# 前言 Linux系统编程要怎么学

跟着老廖教程学习即可 [程序员老廖的个人空间-程序员老廖个人主页-哔哩哔哩视频]

# 第01章 走进Linux系统编程

> **学习目标**：理解Linux系统编程的本质，掌握系统调用与库函数的区别，编写第一个实用工具。

## 引言：为什么学习Linux系统编程?

假设你需要开发一个日志分析工具，每天处理数GB的服务器日志文件。你会发现：

- ❓ 如何高效读取大文件?
- ❓ 如何同时处理多个文件?
- ❓ 如何控制程序的资源占用?
- ❓ 普通的C语言知识够用吗?

这些问题的答案都指向同一个方向：**Linux系统编程**。

### 什么是系统编程?

系统编程是编写**直接与操作系统交互**的软件：

| 对比维度 | 系统编程 | 应用编程 | 嵌入式裸机编程 |
| --- | --- | --- | --- |
| 运行环境 | 操作系统之上 | 高级框架之上 | 直接在硬件上 |
| 主要任务 | 文件/进程/网络操作 | 业务逻辑实现 | 硬件寄存器控制 |
| 典型应用 | 服务器程序、系统工具 | Web应用、桌面软件 | 单片机程序 |
| 关注点 | 资源管理、并发控制 | 用户体验、功能实现 | 硬件时序、中断 |

**本教程聚焦于Linux系统编程**，帮你掌握开发服务器程序、系统工具所需的核心技能。

## 1.1 系统调用：与内核对话的唯一通道

### 1.1.1 什么是系统调用?

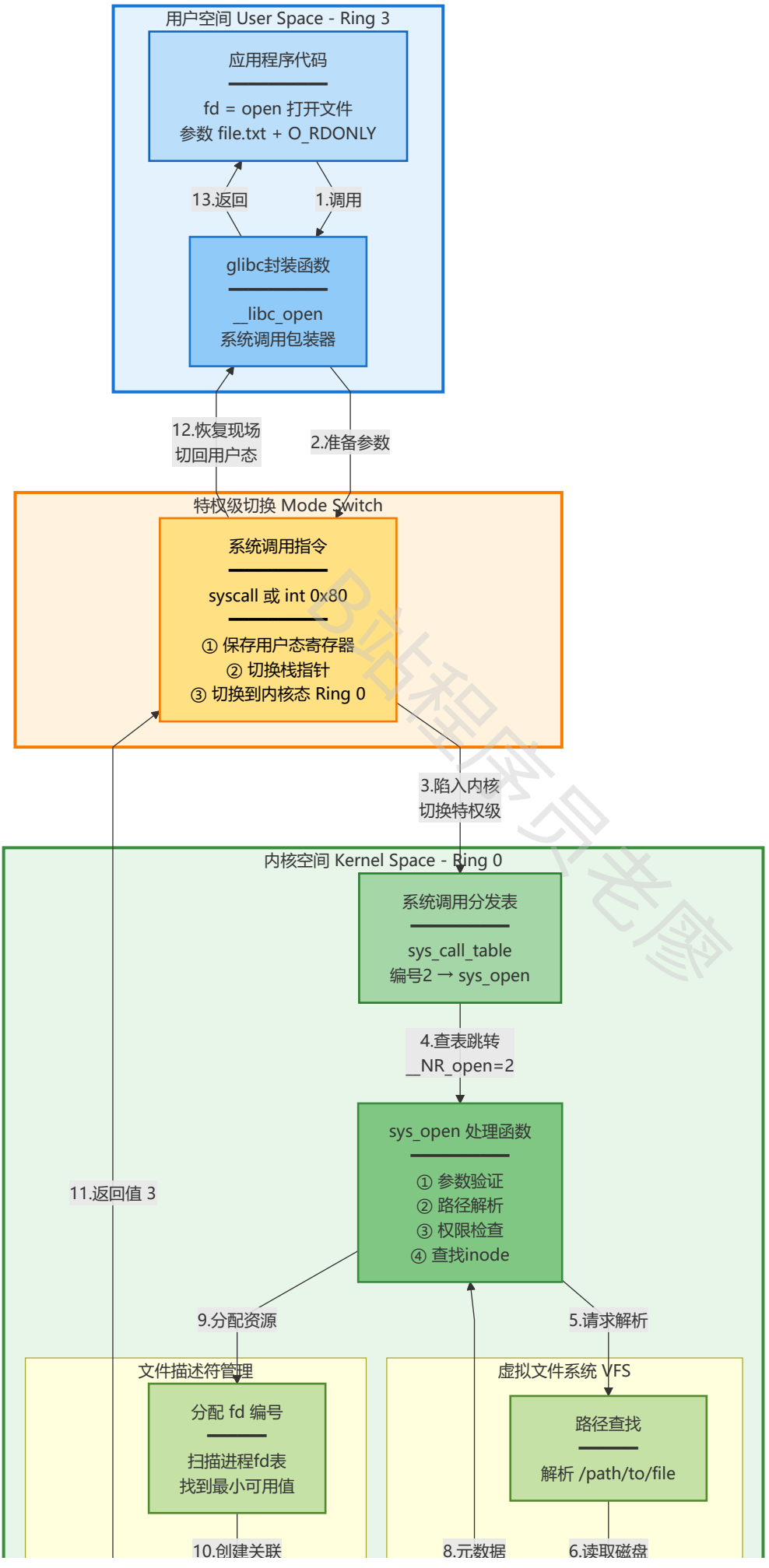
想象你的程序是一个普通员工，而Linux内核是公司老板。员工不能直接访问公司保险柜（硬件资源），必须向老板申请：

```
你的程序：        "我要读取 data.txt 文件"
                        ↓ （系统调用）
Linux内核：       "好的，我帮你读取，返回给你"
                        ↓
你的程序：        "收到文件内容！"
```

**系统调用（System Call）** 就是这个申请通道。它是Linux内核提供给应用程序的API接口，所有涉及硬件访问的操作都必须通过它。

## 1.1.2 系统调用的完整工作流程

下图展示了一个 `open()` 系统调用的完整执行过程：


用户空间 User Space - Ring 3
应用程序代码
fd = open 打开文件
参数 file.txt + O_RDONLY
13.返回
1.调用
glibc封装函数
__libc_open
系统调用包装器
12.恢复现场
切回用户态
2.准备参数
特权级切换 Mode Switch
系统调用指令
syscall 或 int 0x80
① 保存用户态寄存器
② 切换栈指针
③ 切换到内核态 Ring 0
3.陷入内核
切换特权级
内核空间 Kernel Space - Ring 0
系统调用分发表
sys_call_table
编号2 → sys_open
4.查表跳转
__NR_open=2
sys_open 处理函数
① 参数验证
② 路径解析
③ 权限检查
④ 查找inode
11.返回值 3
9.分配资源
5.请求解析
文件描述符管理
分配 fd 编号
扫描进程fd表
找到最小可用值
10.创建关联
虚拟文件系统 VFS
路径查找
解析 /path/to/file
8.元数据
6.读取磁盘

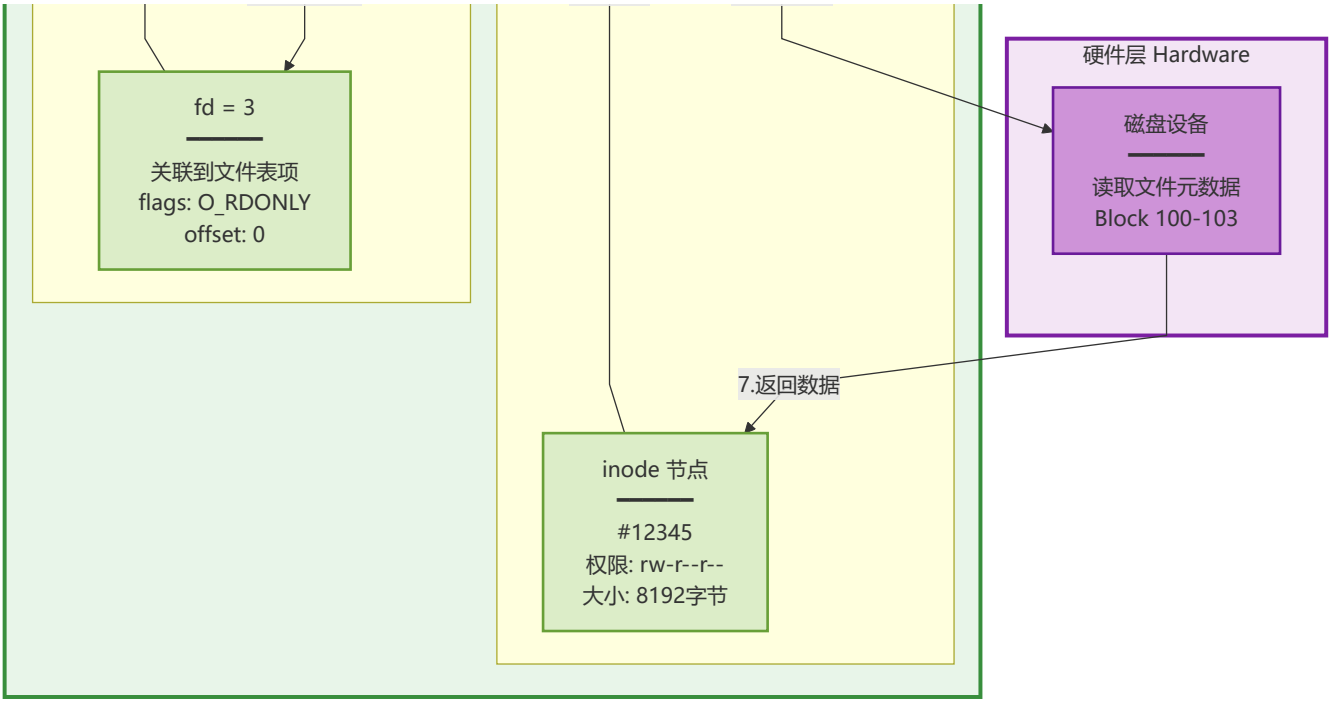



**图 1.1.1 系统调用完整执行流程**

**关键过程解析：**

1. **用户态准备**（蓝色区域）
   - 应用调用 `open()` → glibc包装 → 准备系统调用参数
2. **特权级切换**（橙色区域）
   - 执行 `syscall` 指令 → CPU从Ring 3切换到Ring 0
   - 保存用户态寄存器状态 → 切换到内核栈
3. **内核处理**（绿色区域）
   - 查系统调用表 → 执行 `sys_open()` 处理函数
   - 文件路径解析 → 权限检查 → 查找inode
   - 分配文件描述符 → 创建文件表项
4. **硬件交互**（紫色区域）
   - 从磁盘读取文件元数据（inode信息）
5. **返回用户态**
   - 恢复寄存器 → 切回Ring 3 → 返回 `fd=3`

**性能开销：** 每次系统调用涉及两次特权级切换（用户态→内核态→用户态），这就是为什么要尽量减少系统调用次数。

## 1.1.3 核心系统调用详解

### open() - 打开文件

**函数原型：**

```
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

**参数说明：**

| 参数 | 类型 | 说明 | 示例 |
| --- | --- | --- | --- |
| pathname | const char* | 文件路径（相对或绝对） | "test.txt"<br>"/var/log/app.log" |
| flags | int | 打开模式标志 | O_RDONLY, O_WRONLY, O_RDWR |
| mode | mode_t | 文件权限（创建时需要） | 0644 表示 rw-r--r-- |

**常用flags组合：**

```
// 只读打开（文件必须存在）
int fd = open("data.txt", O_RDONLY);

// 只写打开，不存在则创建
int fd = open("output.txt", O_WRONLY | O_CREAT, 0644);

// 读写打开，存在则清空内容
int fd = open("temp.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);

// 追加写入（数据写到文件末尾）
int fd = open("log.txt", O_WRONLY | O_CREAT | O_APPEND, 0644);
```

**返回值：**

- ✅ **成功**：返回文件描述符（非负整数，通常≥3）
- ❌ **失败**：返回 -1，错误码存储在全局变量 `errno` 中

**文件描述符的特殊值：**

| fd值 | 标准名称 | 用途 |
| --- | --- | --- |
| 0 | STDIN_FILENO | 标准输入（键盘） |
| 1 | STDOUT_FILENO | 标准输出（屏幕） |
| 2 | STDERR_FILENO | 标准错误（屏幕） |
| ≥3 | 自定义 | 用户打开的文件 |

**权限模式说明（mode）：**

```
// 权限位组成：0644 = 所有者可读写，其他人只读
// 采用八进制表示，每3位二进制对应一位八进制

Owner  Group  Others
 rw-    r--    r--
 110    100    100   (二进制)
  6      4      4    (八进制)

// 使用宏定义（更清晰）
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;  // rw-r--r--
//            所有者读  所有者写   组读      其他读
```

**实战示例：**

```
int fd = open("config.txt", O_RDONLY);
if (fd == -1) {
    perror("打开文件失败");  // 自动显示errno对应的错误信息
    return 1;
}
printf("成功打开文件，文件描述符: %d\n", fd);
```

---

## read() - 读取数据

**函数原型：**

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

**参数说明：**

| 参数 | 类型 | 说明 |
| --- | --- | --- |
| fd | int | 文件描述符（由open返回） |
| buf | void* | 数据缓冲区（存储读取的内容） |
| count | size_t | 期望读取的字节数 |

**返回值：**

- `> 0`：实际读取的字节数（可能小于count）
- `= 0`：已到文件末尾（EOF）
- `= -1`：读取失败，检查errno

**为什么读到的可能少于请求的字节数?**

```
场景1: 文件剩余 500 字节，请求 1024 字节 → 返回 500
场景2: 文件剩余 2000 字节，请求 1024 字节 → 返回 1024
场景3: 网络数据未到齐 → 返回已接收的部分（非阻塞模式）
```

**典型用法：**

```c
char buffer[1024];
int n;

// 循环读取直到文件结束
while ((n = read(fd, buffer, 1024)) > 0) {
    // 处理读取到的n个字节
    printf("读取了 %d 字节\n", n);
}

if (n == -1) {
    perror("读取错误");
}
```

---

## write() - 写入数据

**函数原型：**

```c
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

**参数说明：**

| 参数 | 类型 | 说明 |
| --- | --- | --- |
| fd | int | 文件描述符 |
| buf | const void* | 数据缓冲区（要写入的内容） |
| count | size_t | 要写入的字节数 |

**返回值：**

- `> 0`：实际写入的字节数
- `= -1`：写入失败

**注意事项：**

- 写入操作可能被信号中断，返回值小于count
- 磁盘空间不足时会失败
- 如果打开时使用了 `O_APPEND`，数据会追加到文件末尾

---

### close() - 关闭文件

**函数原型：**

```
#include <unistd.h>

int close(int fd);
```

**返回值：**

- `0`：成功关闭
- `-1`：失败（通常不会发生）

**为什么必须关闭文件？**

1. **刷新缓冲区** - 确保数据真正写入磁盘
2. **释放资源** - 文件描述符是有限的（默认进程最多1024个）
3. **解除锁定** - 其他进程才能访问该文件
4. **避免泄漏** - 长期运行的程序会耗尽资源

```
close(fd);
// fd失效，不能再使用！
```

---

## 1.1.4 实战：使用系统调用实现文件复制

让我们编写一个完整的文件复制工具，理解系统调用的实际应用：

**代码文件：** `src/chapter01/syscall_copy.c`

```
/*
 * 文件: syscall_copy.c
 * 功能: 使用系统调用实现文件复制
 *
 * 演示内容:
 *   - open() 打开文件
 *   - read() 读取数据
 *   - write() 写入数据
 *   - close() 关闭文件
 *
 * 编译: gcc -o syscall_copy syscall_copy.c
 * 运行: ./syscall_copy source.txt target.txt
 */

#include <fcntl.h>   // open()
#include <unistd.h>  // read(), write(), close()
#include <stdio.h>   // printf(), perror()

int main(int argc, char *argv[]) {
    // 1. 检查命令行参数
    if (argc != 3) {
```

```
    printf("用法: %s <源文件> <目标文件>\n", argv[0]);
    printf("示例: %s input.txt output.txt\n", argv[0]);
    return 1;
}

// 2. 打开源文件（只读模式）
int src_fd = open(argv[1], O_RDONLY);
if (src_fd == -1) {
    perror("打开源文件失败");
    return 1;
}
printf("✓ 源文件打开成功，fd=%d\n", src_fd);

// 3. 创建目标文件（只写模式，存在则截断）
//    权限设置为 rw-r--r-- (0644)
int dst_fd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (dst_fd == -1) {
    perror("创建目标文件失败");
    close(src_fd);  // 记得关闭已打开的文件
    return 1;
}
printf("✓ 目标文件创建成功，fd=%d\n", dst_fd);

// 4. 循环读取并写入数据
char buffer[1024];  // 1KB缓冲区
int bytes_read;
int total_bytes = 0;

while ((bytes_read = read(src_fd, buffer, sizeof(buffer))) > 0) {
    // 将读取到的数据写入目标文件
    int bytes_written = write(dst_fd, buffer, bytes_read);
    if (bytes_written == -1) {
        perror("写入失败");
        close(src_fd);
        close(dst_fd);
        return 1;
    }
    total_bytes += bytes_written;
}

if (bytes_read == -1) {
    perror("读取失败");
    close(src_fd);
    close(dst_fd);
    return 1;
}

// 5. 关闭文件
close(src_fd);
close(dst_fd);

printf("✓ 文件复制完成！共复制 %d 字节\n", total_bytes);
return 0;
```

```
}
```

**代码逐行解析：**

1. **参数检查**（第18-22行）
    - 确保用户提供了源文件和目标文件名
    - 友好的错误提示
2. **打开源文件**（第25-30行）
    - `O_RDONLY` 表示只读模式
    - 失败时打印错误并退出
3. **创建目标文件**（第33-40行）
    - `O_WRONLY` 只写模式
    - `O_CREAT` 不存在则创建
    - `O_TRUNC` 存在则清空
    - `0644` 设置权限为 rw-r--r--
4. **循环复制**（第43-57行）
    - 每次读取1KB数据
    - 立即写入目标文件
    - 累计复制的字节数
5. **清理资源**（第60-64行）
    - 关闭两个文件描述符
    - 显示复制结果

**编译运行：**

```
# 编译
gcc -o syscall_copy 01_syscall_copy.c

# 测试（先创建一个测试文件）
echo "Hello, System Call!" > test.txt

# 运行
./syscall_copy test.txt copy.txt

# 验证结果
cat copy.txt
# 输出: Hello, System Call!
```

# 1.2 库函数：系统调用的高级封装

## 1.2.1 为什么需要库函数?

直接使用系统调用有性能问题：

```
场景：读取16KB文件，缓冲区4KB

系统调用方式：
read(fd, buf, 4KB)  ← 陷入内核
read(fd, buf, 4KB)  ← 陷入内核
read(fd, buf, 4KB)  ← 陷入内核
read(fd, buf, 4KB)  ← 陷入内核
共4次系统调用，8次特权级切换！
```

**标准C库的解决方案：** 在用户空间增加缓冲层

```
库函数方式：
fread(buf, 4KB, 1, fp)  ← 库内部处理，不陷入内核
fread(buf, 4KB, 1, fp)  ← 库内部处理，不陷入内核
  ↓ 库缓冲区空了，触发一次系统调用
read(fd, lib_buf, 16KB)  ← 陷入内核读取更多数据
fread(buf, 4KB, 1, fp)  ← 从库缓冲区返回
fread(buf, 4KB, 1, fp)  ← 从库缓冲区返回
只需1-2次系统调用！
```

## 1.2.2 系统调用 vs 库函数对比

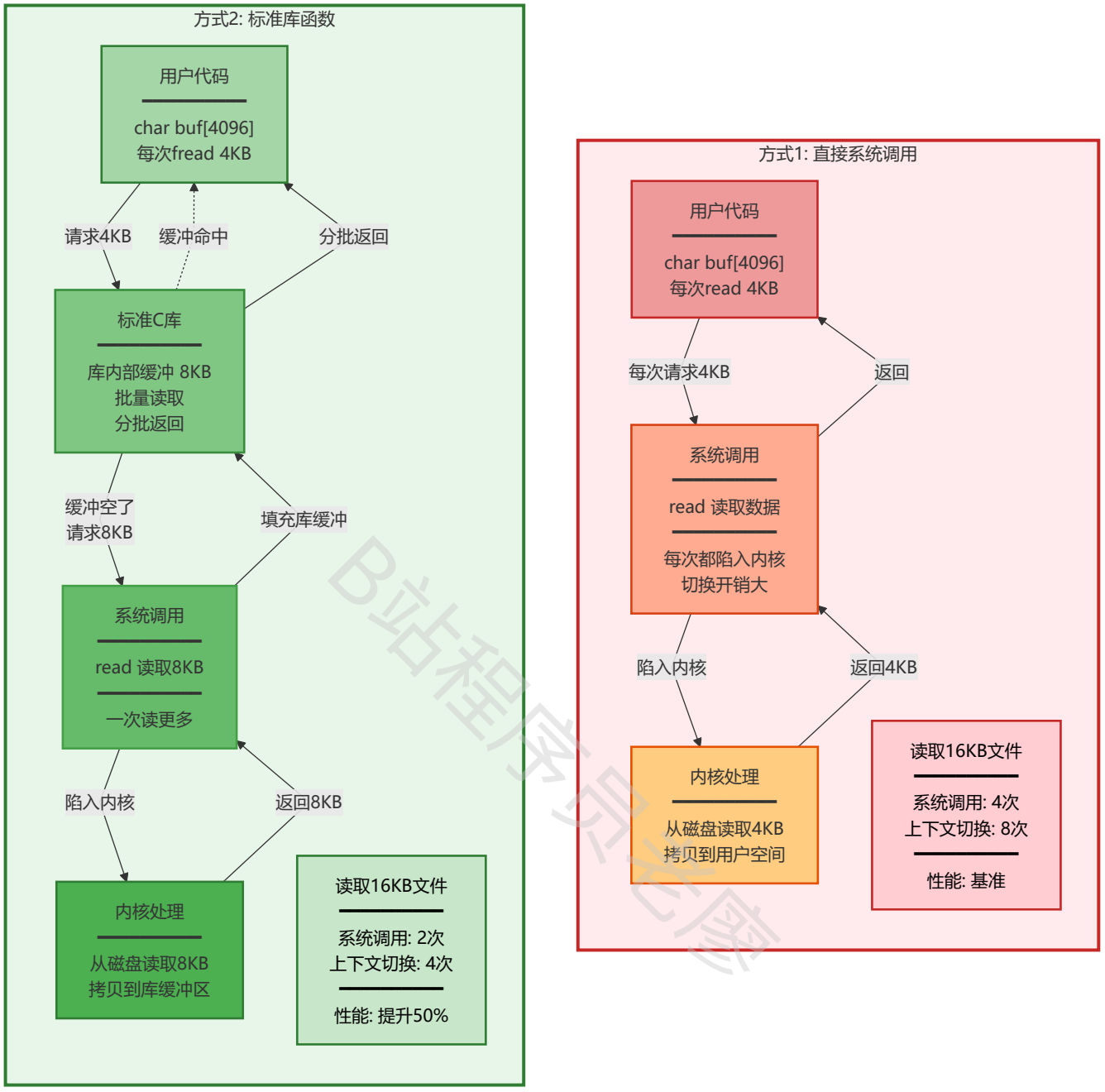



**图 1.2.1 系统调用与库函数性能对比**

## 1.2.3 标准IO库函数

### fopen() - 打开文件流

```
#include <stdio.h>

FILE *fopen(const char *pathname, const char *mode);
```

**模式字符串:**

| 模式 | 说明 | 等效的系统调用flags |
| --- | --- | --- |
| "r" | 只读 | O_RDONLY |
| "w" | 只写（截断） | O_WRONLY \| O_CREAT \| O_TRUNC |
| "a" | 追加写入 | O_WRONLY \| O_CREAT \| O_APPEND |
| "r+" | 读写 | O_RDWR |
| "w+" | 读写（截断） | O_RDWR \| O_CREAT \| O_TRUNC |

**返回值：**

- 成功：返回 `FILE*` 指针
- 失败：返回 `NULL`

### fread() / fwrite() - 读写数据

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

**参数说明：**

- `ptr`：数据缓冲区
- `size`：每个元素的字节数
- `nmemb`：元素个数
- `stream`：文件流指针

**返回值：** 实际读/写的元素个数

### fclose() - 关闭文件流

```
int fclose(FILE *stream);
```

**重要：** `fclose()` 会自动刷新缓冲区，确保数据写入磁盘。

---

## 1.2.4 实战：使用标准IO实现文件复制

**代码文件：** `src/chapter01/stdio_copy.c`

```
/*
 * 文件: stdio_copy.c
 * 功能: 使用标准IO库实现文件复制
 *
 * 演示内容:
 *   - fopen() 打开文件流
 *   - fread() 读取数据（带缓冲）
 *   - fwrite() 写入数据（带缓冲）
 *   - fclose() 关闭文件流
```

```c
 *
 * 编译: gcc -o stdio_copy stdio_copy.c
 * 运行: ./stdio_copy source.txt target.txt
 */

#include <stdio.h>

int main(int argc, char *argv[]) {
    // 1. 检查命令行参数
    if (argc != 3) {
        printf("用法: %s <源文件> <目标文件>\n", argv[0]);
        return 1;
    }

    // 2. 打开源文件（二进制只读）
    //    "rb" 模式适用于任何文件类型
    FILE *src = fopen(argv[1], "rb");
    if (src == NULL) {
        perror("打开源文件失败");
        return 1;
    }
    printf("✓ 源文件打开成功\n");

    // 3. 创建目标文件（二进制只写）
    FILE *dst = fopen(argv[2], "wb");
    if (dst == NULL) {
        perror("创建目标文件失败");
        fclose(src);
        return 1;
    }
    printf("✓ 目标文件创建成功\n");

    // 4. 循环读写数据
    char buffer[1024];
    size_t bytes_read;
    size_t total_bytes = 0;

    while ((bytes_read = fread(buffer, 1, sizeof(buffer), src)) > 0) {
        size_t bytes_written = fwrite(buffer, 1, bytes_read, dst);
        if (bytes_written != bytes_read) {
            printf("写入错误\n");
            fclose(src);
            fclose(dst);
            return 1;
        }
        total_bytes += bytes_written;
    }

    // 5. 关闭文件（自动刷新缓冲区）
    fclose(src);
    fclose(dst);

    printf("✓ 文件复制完成! 共复制 %zu 字节\n", total_bytes);
```

```
    return 0;
}
```

**对比两种方式：**

| 特性 | 系统调用（01） | 标准库（02） |
| --- | --- | --- |
| API | open/read/write/close | fopen/fread/fwrite/fclose |
| 返回值 | int (fd) | FILE* (文件流) |
| 缓冲 | 无（用户自己管理） | 有（库自动管理） |
| 性能 | 频繁系统调用，开销大 | 减少系统调用，性能好 |
| 适用场景 | 底层控制、特殊需求 | 一般应用、推荐使用 |

---

# 1.3 命令行参数处理

## 1.3.1 argc 和 argv 详解

每个C程序的 `main` 函数可以接收命令行参数：

```
int main(int argc, char *argv[]) {
    // argc: 参数个数（argument count）
    // argv: 参数数组（argument vector）
}
```

**示例：** 当执行 `./program hello world` 时：

```
argc = 3
argv[0] = "./program"  (程序名)
argv[1] = "hello"      (第1个参数)
argv[2] = "world"      (第2个参数)
argv[3] = NULL         (数组结束标记)
```

## 1.3.2 实战：命令行参数解析

**代码文件：** `src/chapter01/print_args.c`

```
/*
 * 文件: print_args.c
 * 功能: 演示命令行参数的获取和处理
 *
 * 编译: gcc -o print_args print_args.c
 * 运行: ./print_args arg1 arg2 "arg with spaces"
 */

#include <stdio.h>
```

```c
int main(int argc, char *argv[]) {
    printf("=== 命令行参数解析 ===\n\n");

    // 1. 显示程序名
    printf("程序名称: %s\n", argv[0]);

    // 2. 显示参数个数（不包括程序名）
    printf("参数个数: %d\n", argc - 1);

    // 3. 遍历并显示所有参数
    if (argc > 1) {
        printf("\n参数列表:\n");
        for (int i = 1; i < argc; i++) {
            printf("  [%d] %s\n", i, argv[i]);
        }
    } else {
        printf("\n未提供任何参数\n");
        printf("用法: %s <参数1> <参数2> ...\n", argv[0]);
    }

    return 0;
}
```

**运行示例：**

```
$ ./print_args hello world "Linux programming"
=== 命令行参数解析 ===

程序名称: ./print_args
参数个数: 3

参数列表:
  [1] hello
  [2] world
  [3] Linux programming
```



---

# 1.4 实战项目：文本统计工具

让我们编写一个实用的工具，统计文件的行数、字节数（类似Linux的 `wc` 命令）。

**代码文件：** `src/chapter01/count_lines.c`

```c
/*
 * 文件: count_lines.c
 * 功能: 统计文件的行数和字节数
 *
 * 编译: gcc -o count_lines count_lines.c
 * 运行: ./count_lines file1.txt file2.txt
 */

#include <stdio.h>
```

```c
int main(int argc, char *argv[]) {
    // 1. 检查参数
    if (argc < 2) {
        printf("用法: %s <文件1> [文件2] ...\n", argv[0]);
        return 1;
    }

    // 2. 遍历每个文件
    for (int i = 1; i < argc; i++) {
        FILE *fp = fopen(argv[i], "r");
        if (fp == NULL) {
            printf("无法打开文件: %s\n", argv[i]);
            continue;
        }

        // 3. 统计行数和字节数
        int lines = 0;
        int bytes = 0;
        int ch;

        while ((ch = fgetc(fp)) != EOF) {
            bytes++;
            if (ch == '\n') {
                lines++;
            }
        }

        // 4. 显示结果
        printf("%8d %8d %s\n", lines, bytes, argv[i]);

        fclose(fp);
    }

    return 0;
}
```



**运行示例：**

```
$ ./count_lines test.txt README.md
      10      256 test.txt
      45     1024 README.md
```

# 1.5 开发环境快速搭建

## 1.5.1 安装GCC编译器

```
# Ubuntu/Debian系统
sudo apt update
sudo apt install build-essential

# 验证安装
gcc --version
```

## 1.5.2 编译C程序

```
# 基本编译
gcc -o program source.c

# 带调试信息（用于gdb调试）
gcc -g -o program source.c

# 启用警告信息
gcc -Wall -o program source.c

# 优化编译
gcc -O2 -o program source.c
```



## 1.5.3 GDB调试基础

```
# 编译时加-g选项
gcc -g -o program source.c

# 启动gdb
gdb ./program

# 常用gdb命令
(gdb) break main      # 在main函数设置断点
(gdb) run             # 运行程序
(gdb) next            # 单步执行（不进入函数）
(gdb) step            # 单步执行（进入函数）
(gdb) print var       # 打印变量值
(gdb) continue        # 继续执行
(gdb) quit            # 退出gdb
```

## 1.5.4 使用CMake构建项目

本教程的所有代码都支持CMake构建：

```
# 在项目根目录
mkdir build && cd build

# 生成Makefile
cmake ..

# 编译
make

# 运行
./chapter01/syscall_copy test.txt copy.txt
```

---

# 1.6 API快速参考

## 系统调用函数

| 函数 | 功能 | 返回值 | 头文件 |
| --- | --- | --- | --- |
| open() | 打开文件 | fd（≥0）或-1 | fcntl.h |
| read() | 读取数据 | 字节数（≥0）或-1 | unistd.h |
| write() | 写入数据 | 字节数（≥0）或-1 | unistd.h |
| close() | 关闭文件 | 0或-1 | unistd.h |

## 标准库函数

| 函数 | 功能 | 返回值 | 头文件 |
| --- | --- | --- | --- |
| fopen() | 打开文件流 | FILE* 或NULL | stdio.h |
| fread() | 读取数据（带缓冲） | 元素个数 | stdio.h |
| fwrite() | 写入数据（带缓冲） | 元素个数 | stdio.h |
| fclose() | 关闭文件流 | 0或EOF | stdio.h |
| fgetc() | 读取一个字符 | 字符或EOF | stdio.h |

## open() flags 常用标志

| 标志 | 值 | 说明 |
| --- | --- | --- |
| O_RDONLY | 0 | 只读 |
| O_WRONLY | 1 | 只写 |
| O_RDWR | 2 | 读写 |
| O_CREAT | 0100 | 不存在则创建 |

| 标志 | 值 | 说明 |
| --- | --- | --- |
| O_TRUNC | 01000 | 存在则截断为0 |
| O_APPEND | 02000 | 追加写入 |

## 文件权限模式（mode）

| 宏定义 | 权限 | 八进制 |
| --- | --- | --- |
| S_IRUSR | 所有者可读 | 0400 |
| S_IWUSR | 所有者可写 | 0200 |
| S_IXUSR | 所有者可执行 | 0100 |
| S_IRGRP | 组可读 | 0040 |
| S_IWGRP | 组可写 | 0020 |
| S_IROTH | 其他人可读 | 0004 |
| S_IWOTH | 其他人可写 | 0002 |

**常用组合：**

```
0644  →  rw-r--r--  →  S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH
0755  →  rwxr-xr-x  →  S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH
```



# 1.7 本章总结

## 核心要点

1. **系统调用是与内核交互的唯一方式**
    - 涉及特权级切换（Ring 3 → Ring 0 → Ring 3）
    - 每次系统调用都有性能开销
2. **标准库函数通过缓冲减少系统调用**
    - 在用户空间维护缓冲区
    - 批量调用系统调用
    - 一般应用推荐使用
3. **基本文件操作流程**

    ```
    open() → read()/write() → close()
    ```

4. **命令行参数处理**
    - argc：参数个数

- argv：参数数组
- argv[0]是程序名

## 学习建议

1. **动手实践**
   - 运行本章所有代码示例
   - 尝试修改参数观察效果
   - 用GDB单步调试理解流程
2. **深入理解**
   - 系统调用的完整流程（13个步骤）
   - 缓冲的作用和原理
   - 文件描述符的本质
3. **扩展阅读**
   - `man 2 open` - 查看系统调用手册
   - `man 3 fopen` - 查看C库函数手册
   - Linux内核文档（kernel.org）



# 1.8 知识扩展

## 什么是“Ring”

你提到的 **“Ring 3” 和 “Ring 0”** 中的 “**Ring**”，指的是 **CPU 的特权级（Privilege Level）**，也叫 **保护环（Protection Ring）**。它是现代 CPU（尤其是 x86 架构）中用于实现**操作系统安全与隔离**的一种硬件机制。

---

### Ring简介

“Ring” 字面意思是“环”，它把计算机系统的权限划分为**多个层级（通常是 0 到 3）**，形成一个**由内到外的环形结构**：

```
+------------------+
|     Ring 0       |  ← 最高权限（内核态）
|   （内核空间）     |
+------------------+
|     Ring 1       |  （较少使用）
+------------------+
|     Ring 2       |  （较少使用）
+------------------+
|     Ring 3       |  ← 最低权限（用户态）
|   （用户空间）     |
+------------------+
```

- **Ring 0**：最靠近硬件的核心层，拥有最高权限，可以访问所有硬件资源（如内存、磁盘、网络、CPU 控制寄存器等）。

- **Ring 3**：最外层，权限最低，只能访问自己的内存空间，不能直接操作硬件。

> 💡 类比：就像一个城堡，Ring 0 是国王和禁卫军住的核心区域，Ring 3 是普通百姓居住的外围区域。想进入内层，必须经过严格验证。

---

## 为什么需要 Ring?

主要目的：**保护系统安全和稳定**

1. **防止用户程序乱来**
   - 比如一个浏览器崩溃了，不能让整个系统死机。
   - 如果用户程序可以直接操作内存或关机指令，那太危险了。
2. **实现“最小权限原则”**
   - 普通程序只需要做自己的事（比如计算、读文件），不需要控制硬件。
   - 只有操作系统内核才需要这些高权限操作。
3. **支持多任务和内存隔离**
   - 每个用户进程运行在 Ring 3，彼此隔离。
   - 内核统一管理资源，在 Ring 0 运行。

---

## Ring 0 vs Ring 3 对比

| 特性 | Ring 0（内核态） | Ring 3（用户态） |
| --- | --- | --- |
| 权限等级 | 最高（特权级 0） | 最低（特权级 3） |
| 能否访问硬件 | ✅ 可以（如读写磁盘、中断控制器） | ❌ 不允许 |
| 能否修改页表/内存管理 | ✅ 可以 | ❌ 不允许 |
| 典型运行代码 | 操作系统内核、驱动程序 | 应用程序（浏览器、QQ、游戏） |
| 安全性影响 | 出错会导致系统崩溃（蓝屏） | 出错只影响自己进程 |

---

## 用户程序如何进入 Ring 0?

用户程序不能直接跳到 Ring 0，必须通过**系统调用（System Call）** 或 **中断（Interrupt）** 来“请求”内核帮忙。

**典型流程（比如 `open()` 文件）**

1. 用户程序在 **Ring 3** 调用 `open("file.txt", O_RDONLY)`
2. glibc 封装函数准备参数
3. 执行 `syscall` 指令（x86-64）或 `int 0x80`（旧版）
4. CPU **切换到 Ring 0**，跳转到内核的系统调用处理函数
5. 内核在 Ring 0 执行真正的文件打开操作
6. 完成后，通过 `sysret` 或 `iret` 返回 Ring 3

7. 用户程序继续执行

> 🔁 这个过程叫做 **“模式切换”（Mode Switch）**，不是进程切换，而是**特权级切换。**

---

## 实际例子

| 程序 | 运行在哪个 Ring? | 说明 |
| --- | --- | --- |
| Windows/Linux 内核 | Ring 0 | 管理硬件、调度进程 |
| 显卡驱动、网卡驱动 | Ring 0 | 需要直接操作硬件 |
| Chrome 浏览器 | Ring 3 | 不能直接读硬盘 |
| 你的 C++ 程序 `int main()` | Ring 3 | 所有用户程序默认在此 |
| `printf("hello")` | 先 Ring 3，后进入 Ring 0 | 最终要通过系统调用写屏幕 |

---

## 为什么是 4 个 Ring? （0~3）

Intel 的 x86 架构设计了 **4 个特权级（CPL: Current Privilege Level）**，编号 0~3。

但现代操作系统（如 Linux、Windows）通常只用两个：

- **Ring 0**：内核
- **Ring 3**：用户程序

Ring 1 和 Ring 2 很少使用，有些系统曾用于：

- 虚拟机监控器（VMM）
- 特权较高的服务程序

但最终被简化了。

> 💡 所以你现在看到的“Ring 3 → Ring 0”，其实是“**用户态 → 内核态**”的代名词。

---

## 总结：一句话解释“Ring”

> **“Ring” 是 CPU 提供的硬件级权限隔离机制，Ring 0 是最高权限（内核），Ring 3 是最低权限（用户程序），通过系统调用可以从 Ring 3 进入 Ring 0，但不能反向随意跳转。**

---

如果你画图，可以用这个比喻：

```
+-----------------------------------+
|            用户程序                |
|          (运行在 Ring 3)           |
|    不能碰硬件，不能乱改内存        |
+-----------------+-----------------+
                  |
                  v
          系统调用（syscall）
```

```
                        |
                        v
      +-------------------+-------------+
      |             内核                |
      |          (运行在 Ring 0)        |
      |     可以操作一切：磁盘、内存、CPU |
      +---------------------------------+
```

# 第02章 系统文件IO进阶

> **学习目标**：深入理解文件描述符机制，掌握文件偏移量控制，学会正确的错误处理。

---

## 引言：从问题出发

第01章我们学会了基本的文件读写，但在实际开发中还会遇到这些问题：

```
int fd = open("log.txt", O_WRONLY);
write(fd, "日志1\n", 7);
write(fd, "日志2\n", 7);
// ❓ 日志2会覆盖日志1吗？还是追加？

int fd1 = open("file.txt", O_RDONLY);
int fd2 = open("file.txt", O_RDONLY);
read(fd1, buf, 100);
// ❓ fd2会从哪里开始读？从0还是从100？

if (open("file.txt", O_RDONLY) == -1) {
    // ❓ 如何知道具体什么错误？权限不够？文件不存在？
}
```



本章将回答这些问题，让你真正掌握系统文件IO。

---

## 2.1 文件描述符深入理解

### 2.1.1 文件描述符是什么？

**文件描述符（File Descriptor，fd）** 是一个**非负整数**，是内核给打开文件分配的"编号"。

**生活类比：**

```
文件描述符 = 图书馆的借书卡号
————————————————————
1. 借书 → open()，得到卡号3
2. 用卡号3去借阅 → read(3, ...)
3. 还书 → close(3)
```

**代码示例：**

```c
int fd = open("test.txt", O_RDONLY);
printf("文件描述符: %d\n", fd);  // 通常输出: 3

// fd就是一个整数，用来标识这个打开的文件
```

## 2.1.2 标准的文件描述符

每个程序启动时，系统自动打开3个文件描述符：

| fd值 | 宏定义 | 用途 | 重定向目标 |
| --- | --- | --- | --- |
| 0 | STDIN_FILENO | 标准输入 | 键盘 |
| 1 | STDOUT_FILENO | 标准输出 | 终端 |
| 2 | STDERR_FILENO | 标准错误 | 终端 |

```c
// 这就是为什么printf能直接输出到屏幕
printf("Hello\n");  // 等价于 write(1, "Hello\n", 6);

// 这就是为什么scanf能从键盘读取
scanf("%d", &n);    // 等价于从fd 0读取
```

**因此，你打开的第一个文件通常得到fd=3。**

## 2.1.3 文件描述符的分配规则

**规则：内核总是分配当前最小的未使用文件描述符。**

```c
// 演示最小可用分配
int fd1 = open("file1.txt", O_RDONLY);  // fd1 = 3
int fd2 = open("file2.txt", O_RDONLY);  // fd2 = 4

close(fd1);  // 释放fd 3

int fd3 = open("file3.txt", O_RDONLY);  // fd3 = 3（复用）
```

## 2.1.4 文件描述符的内核结构

当你调用 `open()` 时，内核会做这些事：

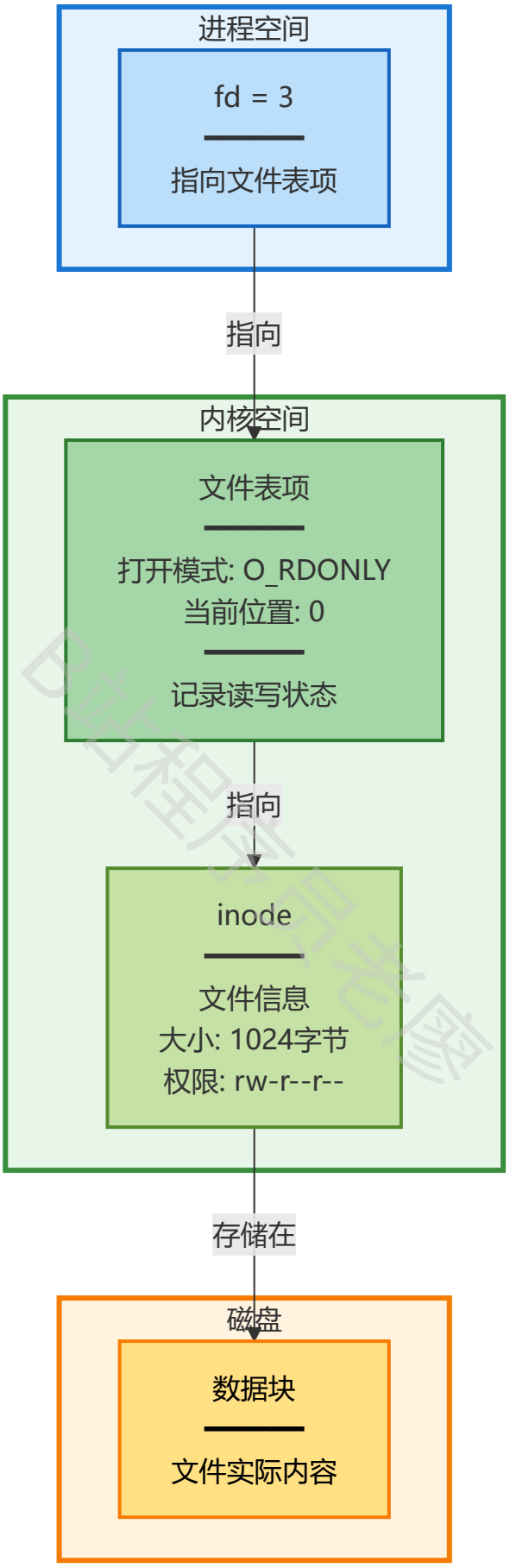



**图 2.1.1 文件描述符与内核结构**

| 概念 | 你的比喻 | 实际含义 | 存储位置 |
| --- | --- | --- | --- |
| **fd (文件描述符)** | 你手里的"钥匙号" | 一个整数，是进程打开文件的**索引**，指向内核中的文件表项 | 进程的文件描述符表（内存） |
| **文件表项 (file table entry)** | "钥匙开到第几页了"（记录offset） | 记录文件的**当前读写位置（offset）**、访问模式（如只读、写）、引用计数等 | **内核内存**（系统级） |
| **inode** | 文件的基本信息（大小、权限、时间戳、数据块指针等） | 文件的**元数据（metadata）**，不包含文件名和实际数据 | **磁盘**（持久存储） |
| **数据块 (data blocks)** | 文件真正的内容 | 文件的实际数据内容，由 inode 中的指针指向 | **磁盘**（持久存储） |

**关键点：每次 `open()` 都会创建新的文件表项！**

---

# 2.2 文件偏移量与lseek

## 2.2.1 什么是文件偏移量?

**文件偏移量（offset）** 记录了下次读写的起始位置。

**可视化：**

```
文件内容:  H  e  l  l  o  ,     W  o  r  l  d  !
字节位置:  0  1  2  3  4  5  6  7  8  9  10 11 12
           ↑
        offset=0
```

**默认行为：**

```
int fd = open("file.txt", O_RDONLY);
// 打开时 offset = 0

read(fd, buf, 5);  // 读取"Hello"
// 现在 offset = 5（自动后移）

read(fd, buf, 2);  // 读取", "
// 现在 offset = 7
```

## 2.2.2 lseek() - 控制文件偏移量

**函数原型：**

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

**参数说明：**

| 参数 | 说明 |
| --- | --- |
| fd | 文件描述符 |
| offset | 偏移量（字节数，可为负） |
| whence | 参考点：SEEK_SET / SEEK_CUR / SEEK_END |

**whence的三种模式：**

| 模式 | 含义 | 新位置计算 |
| --- | --- | --- |
| SEEK_SET | 从文件开头 | offset |
| SEEK_CUR | 从当前位置 | 当前位置 + offset |
| SEEK_END | 从文件末尾 | 文件大小 + offset |

**返回值：**

- 成功：返回新的偏移量
- 失败：返回 -1

**典型用法：**

```
// 1. 移到文件开头
lseek(fd, 0, SEEK_SET);

// 2. 移到文件末尾
lseek(fd, 0, SEEK_END);

// 3. 获取当前位置（不移动）
off_t pos = lseek(fd, 0, SEEK_CUR);

// 4. 往回移动10字节
lseek(fd, -10, SEEK_CUR);

// 5. 跳过文件头100字节
lseek(fd, 100, SEEK_SET);
```



## 2.2.3 实战：获取文件大小

**巧妙技巧：移到文件末尾，返回值就是文件大小！**

**注意**：也可以使用stat函数获取文件大小，见“第04章 文件与目录操作”。

**代码文件：** `src/chapter02/get_file_size.c`

```
/*
 * 功能: 使用lseek获取文件大小
 * 编译: gcc -o get_file_size get_file_size.c
```

```
 * 运行: ./get_file_size test.txt
 */

#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("用法: %s <文件名>\n", argv[0]);
        return 1;
    }

    int fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror("打开失败");
        return 1;
    }

    // 核心技巧: 移到末尾
    off_t size = lseek(fd, 0, SEEK_END);

    printf("文件: %s\n", argv[1]);
    printf("大小: %ld 字节\n", size);

    close(fd);
    return 0;
}
```



## 2.2.4 实战：文件指定位置读取

**代码文件：** `src/chapter02/read_at_offset.c`

```
/*
 * 功能: 从文件指定位置读取内容
 * 编译: gcc -o read_at_offset read_at_offset.c
 * 运行: ./read_at_offset file.txt 100 50
 *       (从字节100开始，读取50字节)
 */

#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc != 4) {
        printf("用法: %s <文件> <起始位置> <字节数>\n", argv[0]);
        return 1;
```

```
    }

    off_t start = atol(argv[2]);
    size_t count = atol(argv[3]);

    int fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror("打开失败");
        return 1;
    }

    // 定位到指定位置
    if (lseek(fd, start, SEEK_SET) == -1) {
        perror("lseek失败");
        close(fd);
        return 1;
    }

    // 读取数据
    char *buffer = malloc(count + 1);
    ssize_t n = read(fd, buffer, count);

    if (n > 0) {
        buffer[n] = '\0';
        printf("读取内容:\n%s\n", buffer);
    }

    free(buffer);
    close(fd);
    return 0;
}
```



# 2.3 错误处理机制

## 2.3.1 为什么需要错误处理?

**错误的代码:**

```
int fd = open("file.txt", O_RDONLY);
read(fd, buf, 100);  // 如果open失败，这里会崩溃!
```

**正确的代码:**

```
int fd = open("file.txt", O_RDONLY);
if (fd == -1) {
    // 处理错误
    return 1;
}
read(fd, buf, 100);
```

## 2.3.2 errno - 全局错误码

当系统调用失败时，内核会设置全局变量 `errno`：

```
#include <errno.h>

int fd = open("file.txt", O_RDONLY);
if (fd == -1) {
    printf("错误码: %d\n", errno);
    // errno == 2  表示 ENOENT（文件不存在）
    // errno == 13 表示 EACCES（权限不足）
}
```

**常见错误码：**

| 错误码 | 宏定义 | 含义 |
| --- | --- | --- |
| 2 | ENOENT | 文件不存在 |
| 13 | EACCES | 权限不足 |
| 17 | EEXIST | 文件已存在 |
| 24 | EMFILE | 打开文件过多 |



## 2.3.3 perror() - 打印错误信息（推荐）

**最简单的错误处理方式：**

```
#include <stdio.h>

int fd = open("file.txt", O_RDONLY);
if (fd == -1) {
    perror("打开文件失败");
    // 自动输出: 打开文件失败: No such file or directory
    return 1;
}
```

**perror() 的优点：**

- ✅ 自动根据errno显示错误信息
- ✅ 可以自定义前缀
- ✅ 线程安全

### 2.3.4 strerror() - 获取错误描述

```
#include <string.h>
#include <errno.h>

int fd = open("file.txt", O_RDONLY);
if (fd == -1) {
    printf("错误: %s\n", strerror(errno));
    // 输出: 错误: No such file or directory
}
```

### 2.3.5 实战：完整的错误处理

**代码文件：** `src/chapter02/error_demo.c`

```
/*
 * 功能: 演示正确的错误处理
 * 编译: gcc -o error_demo error_demo.c
 */

#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main(void) {
    printf("=== 错误处理演示 ===\n\n");

    // 测试1: 打开不存在的文件
    printf("测试1: 打开不存在的文件\n");
    int fd1 = open("不存在.txt", O_RDONLY);
    if (fd1 == -1) {
        printf("  方式1 - perror: ");
        perror("");
        printf("  方式2 - strerror: %s\n", strerror(errno));
        printf("  方式3 - errno值: %d\n\n", errno);
    }

    // 测试2: 创建已存在的文件（排他模式）
    printf("测试2: 排他创建（文件已存在）\n");
    system("touch 已存在.txt");  // 先创建文件
    int fd2 = open("已存在.txt", O_CREAT | O_EXCL, 0644);
    if (fd2 == -1) {
        perror("  创建失败");
        printf("  错误码: %d (EEXIST)\n\n", errno);
    }

    // 测试3: 无权限访问
    printf("测试3: 无权限访问\n");
    system("touch 无权限.txt && chmod 000 无权限.txt");
    int fd3 = open("无权限.txt", O_RDONLY);
```

```
    if (fd3 == -1) {
        perror("  访问失败");
        printf("  错误: %s\n", strerror(errno));
    }

    // 清理
    system("rm -f 已存在.txt 无权限.txt");

    return 0;
}
```

**运行效果：**

```
$ ./error_demo
=== 错误处理演示 ===

测试1: 打开不存在的文件
  方式1 - perror: : No such file or directory
  方式2 - strerror: No such file or directory
  方式3 - errno值: 2

测试2: 排他创建（文件已存在）
  创建失败: File exists
  错误码: 17 (EEXIST)

测试3: 无权限访问
  访问失败: Permission denied
  错误: Permission denied
```



# 2.4 open标志详解

## 2.4.1 常用标志组合

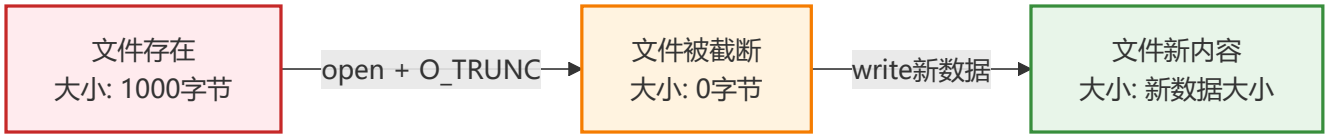
| 标志 | 含义 | 使用场景 |
| --- | --- | --- |
| O_RDONLY | 只读 | 读取文件内容 |
| O_WRONLY | 只写 | 写入或覆盖文件 |
| O_RDWR | 读写 | 需要读写操作 |
| O_CREAT | 不存在则创建 | 创建新文件 |
| O_TRUNC | 截断为0 | 清空文件重新写 |
| O_APPEND | 追加写入 | 日志、追加数据 |
| O_EXCL | 排他创建 | 确保创建新文件 |

## 2.4.2 O_TRUNC - 截断文件

**作用：打开文件时，如果文件存在，将其大小截断为0。**

```
// 场景：覆盖写入
int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
write(fd, "新内容", 9);
close(fd);
// 文件只有"新内容"，旧内容被清空
```

**流程图：**




## 2.4.3 O_APPEND - 追加写入

**作用：每次写入前，自动移动到文件末尾。**

```
// 场景：日志文件
int fd = open("app.log", O_WRONLY | O_CREAT | O_APPEND, 0644);
write(fd, "日志1\n", 7);
write(fd, "日志2\n", 7);
close(fd);
// 两条日志都在文件末尾，不会覆盖
```

**对比：有无O_APPEND**

```
// 没有O_APPEND（会覆盖）
int fd = open("test.txt", O_WRONLY);
write(fd, "AAAA", 4);  // 写在开头
write(fd, "BBBB", 4);  // 从第4字节开始
// 结果: AAAABBBB

// 有O_APPEND（追加）
int fd = open("test.txt", O_WRONLY | O_APPEND);
write(fd, "AAAA", 4);  // 追加到末尾
write(fd, "BBBB", 4);  // 继续追加
// 结果: 原内容AAAABBBB
```

## 2.4.4 O_EXCL - 排他创建

**作用：与O_CREAT一起使用，如果文件已存在则失败。**

```c
// 确保创建新文件，如果存在则报错
int fd = open("new.txt", O_WRONLY | O_CREAT | O_EXCL, 0644);
if (fd == -1) {
    if (errno == EEXIST) {
        printf("文件已存在，拒绝覆盖\n");
    }
    return 1;
}
```

**应用场景：** 防止覆盖重要文件、创建临时文件、实现文件锁。

## 2.4.5 实战：日志追加工具

**代码文件：** `src/chapter02/append_log.c`

```c
/*
 * 功能：向日志文件追加内容
 * 编译：gcc -o append_log append_log.c
 * 运行：./append_log app.log "服务启动"
 */

#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("用法：%s <日志文件> <日志内容>\n", argv[0]);
        return 1;
    }

    // 以追加模式打开日志文件
    int fd = open(argv[1], O_WRONLY | O_CREAT | O_APPEND, 0644);
    if (fd == -1) {
        perror("打开日志文件失败");
        return 1;
    }

    // 获取当前时间
    time_t now = time(NULL);
    char time_str[64];
    strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S",
             localtime(&now));

    // 格式化日志：[时间] 内容\n
    char log_line[256];
    snprintf(log_line, sizeof(log_line), "[%s] %s\n",
             time_str, argv[2]);

    // 写入日志
```

```
    write(fd, log_line, strlen(log_line));

    close(fd);
    printf("日志已追加到 %s\n", argv[1]);

    return 0;
}
```

**使用示例：**

```
$ ./append_log app.log "系统启动"
日志已追加到 app.log

$ ./append_log app.log "用户登录: admin"
日志已追加到 app.log

$ cat app.log
[2025-10-10 14:30:15] 系统启动
[2025-10-10 14:30:20] 用户登录: admin
```

---

# 2.5 多次打开同一文件

## 2.5.1 问题引入

```
int fd1 = open("file.txt", O_RDONLY);
int fd2 = open("file.txt", O_RDONLY);

read(fd1, buf, 100);  // fd1读取100字节
// ❓ fd2会从哪里开始读？
```

**答案：fd2从0开始读！**

## 2.5.2 核心原理

**每次 `open()` 都会创建独立的文件表项（存储在内存里），因此offset是独立的。**

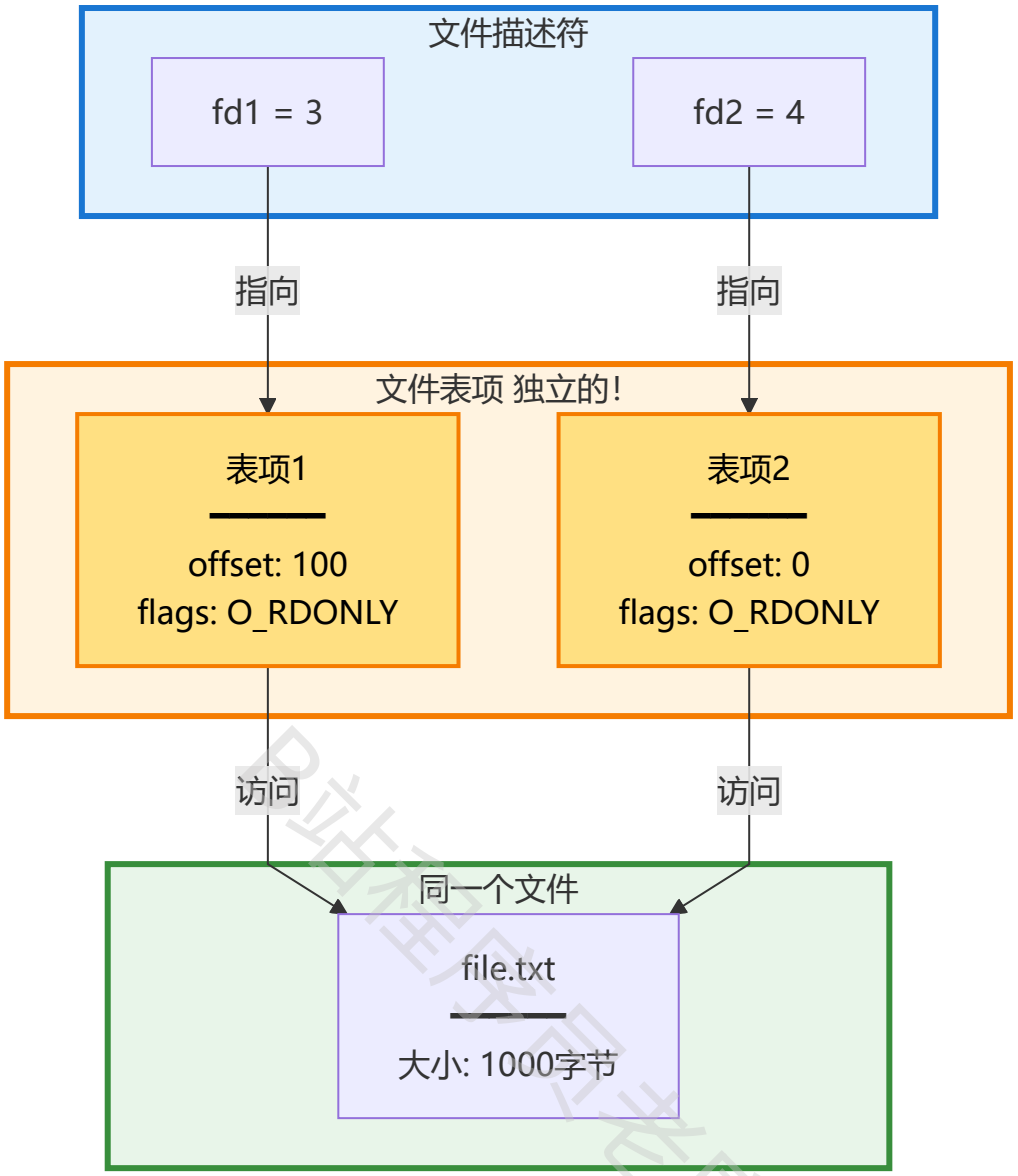



**图 2.5.1 多次打开同一文件**

## 2.5.3 实验验证

**代码文件：** `src/chapter02/multiple_open.c`

```c
/*
 * 功能: 验证多次打开同一文件的offset独立性
 * 编译: gcc -o multiple_open multiple_open.c
 */

#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(void) {
    // 先创建测试文件
    int fd = open("test.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    write(fd, "0123456789ABCDEFGHIJ", 20);
```

```
    close(fd);

    printf("=== 多次打开同一文件测试 ===\n\n");

    // 两次打开同一文件
    int fd1 = open("test.txt", O_RDONLY);
    int fd2 = open("test.txt", O_RDONLY);

    printf("fd1 = %d\n", fd1);
    printf("fd2 = %d\n\n", fd2);

    // fd1读取5字节
    char buf1[10] = {0};
    read(fd1, buf1, 5);
    printf("fd1 读取: %s\n", buf1);
    printf("fd1 当前位置: %ld\n\n", lseek(fd1, 0, SEEK_CUR));

    // fd2读取5字节
    char buf2[10] = {0};
    read(fd2, buf2, 5);
    printf("fd2 读取: %s\n", buf2);
    printf("fd2 当前位置: %ld\n\n", lseek(fd2, 0, SEEK_CUR));

    printf("结论: fd1和fd2的offset是独立的! \n");

    close(fd1);
    close(fd2);

    return 0;
}
```

**运行结果:**

```
$ ./multiple_open
=== 多次打开同一文件测试 ===

fd1 = 3
fd2 = 4

fd1 读取: 01234
fd1 当前位置: 5

fd2 读取: 01234
fd2 当前位置: 5

结论: fd1和fd2的offset是独立的!
```

## 2.5.4 实际应用场景

### 场景1：多线程/多进程读取同一文件

```
// 线程1
int fd1 = open("data.txt", O_RDONLY);
// 每个线程独立读取，不会相互干扰

// 线程2
int fd2 = open("data.txt", O_RDONLY);
```

### 场景2：同时读写同一文件

在某些应用中，需要同时读取和写入同一文件（如日志分析+记录）。

**代码文件：** `src/chapter02/read_write_same.c`

```
/*
 * 功能：演示同时读写同一文件
 * 编译：gcc -o read_write_same read_write_same.c
 */

#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
    // 1. 同时打开读和写
    int fd_read = open("data.txt", O_RDONLY);
    int fd_write = open("data.txt", O_WRONLY | O_APPEND);

    // 2. 先读取原有内容
    char buffer[256];
    ssize_t n = read(fd_read, buffer, sizeof(buffer));
    printf("读取原有内容：%zd 字节\n", n);

    // 3. 追加新内容
    write(fd_write, "新的一行\n", 12);

    // 4. 继续读取（会读到新追加的内容）
    n = read(fd_read, buffer, sizeof(buffer));
    printf("继续读取：%zd 字节\n", n);

    // 5. 重新定位到开头，读取完整文件
    lseek(fd_read, 0, SEEK_SET);
    n = read(fd_read, buffer, sizeof(buffer));
    printf("完整文件：%zd 字节\n", n);

    close(fd_read);
    close(fd_write);

    return 0;
```

```
}
```

**运行效果：**

```
$ ./read_write_same
=== 同时读写同一文件演示 ===

1. 创建初始文件
   初始文件内容已创建

2. 打开两个文件描述符
   fd_read  = 5 (只读)
   fd_write = 6 (追加写)

3. 读取原有内容
   读取到 42 字节:
   ----------------------
初始内容：Line1
初始内容：Line2
   ----------------------

4. 追加新内容（通过 fd_write）
   已追加2行

5. 继续从 fd_read 读取
   又读取到 42 字节:
   ----------------------
追加内容：Line3
追加内容：Line4
   ----------------------

6. 重新定位 fd_read 到文件开头
   读取完整文件（共 84 字节）:
   ======================
初始内容：Line1
初始内容：Line2
追加内容：Line3
追加内容：Line4
   ======================

总结:
  ✓ fd_read 和 fd_write 是独立的文件表项
  ✓ 它们各自维护独立的 offset
  ✓ fd_write 使用 O_APPEND，写入总是在末尾
  ✓ fd_read 可以通过 lseek 重新定位读取新内容
```

**关键理解：**

- ✅ 两个fd独立，各自维护offset
- ✅ `fd_write` 的 O_APPEND 确保总是追加到末尾
- ✅ `fd_read` 可以读到后续追加的内容（需要继续read或lseek重定位）

- ✅ 典型应用：日志文件的同时读取和写入

**并发写入的安全性问题：**

## 场景3：多个进程/线程同时写同一文件会发生什么？

**情况1：不使用O_APPEND（危险！）**

```
// 进程A
int fd = open("log.txt", O_WRONLY);
lseek(fd, 0, SEEK_END);  // 步骤1：移到末尾
write(fd, "A的数据", 9); // 步骤2：写入

// 进程B（同时执行）
int fd = open("log.txt", O_WRONLY);
lseek(fd, 0, SEEK_END);  // 步骤1：移到末尾
write(fd, "B的数据", 9); // 步骤2：写入
```

**可能的问题：**

- ⚠️ A和B的lseek都读到相同的文件末尾位置（假设1000）
- ⚠️ A先写入，文件变成1009字节
- ⚠️ B也从位置1000写入，**覆盖了A的数据**！
- ❌ 结果：数据丢失

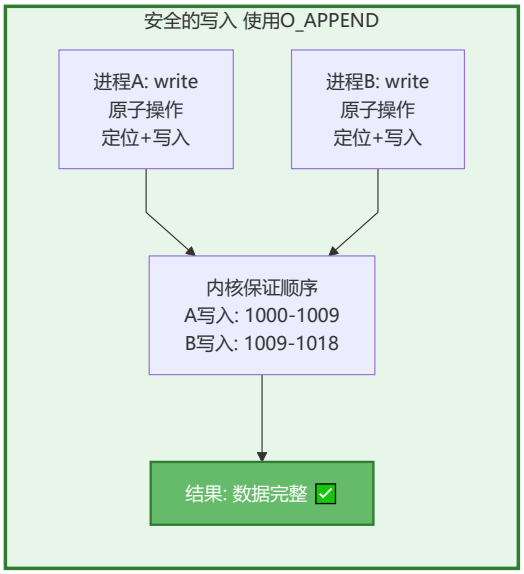
**情况2：使用O_APPEND（安全）**

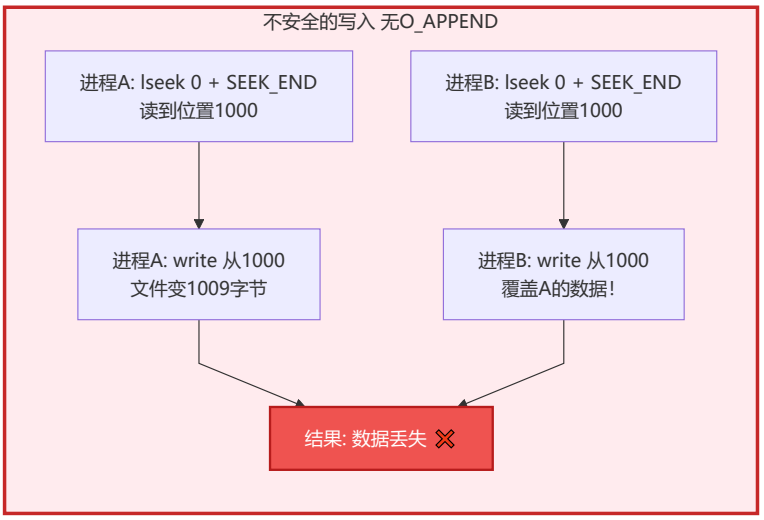
```
// 进程A
int fd = open("log.txt", O_WRONLY | O_APPEND);
write(fd, "A的数据", 9);  // 原子操作：定位+写入

// 进程B（同时执行）
int fd = open("log.txt", O_WRONLY | O_APPEND);
write(fd, "B的数据", 9);  // 原子操作：定位+写入
```

**为什么安全？**

- ✅ O_APPEND使"定位到末尾+写入"成为**原子操作**
- ✅ 内核保证写入时自动定位到**当前**文件末尾
- ✅ 不会发生数据覆盖
- ✅ A和B的数据都会正确追加

**对比图示：**




**实际应用场景：**

| 场景 | 推荐方式 | 原因 |
| --- | --- | --- |
| 多进程写日志 | O_APPEND | 防止数据交错 |
| 单进程覆盖文件 | O_TRUNC | 清空重写 |
| 多线程追加数据 | O_APPEND + 锁 | 更安全 |
| 临时文件写入 | O_EXCL + O_CREAT | 防止冲突 |

**重要结论：**

- 📌 **多进程/线程写同一文件，必须使用O_APPEND**
- 📌 O_APPEND的"定位+写入"是**原子操作**，线程安全
- 📌 对于关键数据，还可以：每个进程写独立文件，最后合并

---

# 2.6 综合实战项目

## 2.6.1 带错误处理的文件复制工具

**代码文件：** `src/chapter02/safe_copy.c`

```
/*
 * 功能: 带完整错误处理的文件复制工具
 * 编译: gcc -o safe_copy safe_copy.c
 * 运行: ./safe_copy source.txt dest.txt
 */

#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
```

```c
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "用法: %s <源文件> <目标文件>\n", argv[0]);
        return 1;
    }

    // 1. 打开源文件
    int src_fd = open(argv[1], O_RDONLY);
    if (src_fd == -1) {
        perror("打开源文件失败");
        return 1;
    }
    printf("✓ 成功打开源文件: %s (fd=%d)\n", argv[1], src_fd);

    // 2. 创建目标文件（排他模式，防止覆盖）
    int dst_fd = open(argv[2], O_WRONLY | O_CREAT | O_EXCL, 0644);
    if (dst_fd == -1) {
        if (errno == EEXIST) {
            fprintf(stderr, "错误: 目标文件已存在，拒绝覆盖\n");
        } else {
            perror("创建目标文件失败");
        }
        close(src_fd);
        return 1;
    }
    printf("✓ 成功创建目标文件: %s (fd=%d)\n", argv[2], dst_fd);

    // 3. 循环复制
    char buffer[4096];
    ssize_t bytes_read, bytes_written;
    size_t total = 0;

    while ((bytes_read = read(src_fd, buffer, sizeof(buffer))) > 0) {
        bytes_written = write(dst_fd, buffer, bytes_read);
        if (bytes_written == -1) {
            perror("写入失败");
            close(src_fd);
            close(dst_fd);
            return 1;
        }
        if (bytes_written != bytes_read) {
            fprintf(stderr, "警告: 写入不完整\n");
        }
        total += bytes_written;
    }

    if (bytes_read == -1) {
        perror("读取失败");
        close(src_fd);
        close(dst_fd);
        return 1;
    }
```

```
    // 4. 关闭文件
    close(src_fd);
    close(dst_fd);

    printf("✓ 复制完成！共复制 %zu 字节\n", total);

    return 0;
}
```

## 2.6.2 文件内容查看工具（类似cat）

**代码文件：** `src/chapter02/simple_cat.c`

```
/*
 * 功能：显示文件内容（类似cat命令）
 * 编译：gcc -o simple_cat simple_cat.c
 * 运行：./simple_cat file.txt
 */

#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "用法：%s <文件名>\n", argv[0]);
        return 1;
    }

    int fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror("打开文件失败");
        return 1;
    }

    char buffer[4096];
    ssize_t n;

    // 循环读取并输出到标准输出（fd=1）
    while ((n = read(fd, buffer, sizeof(buffer))) > 0) {
        if (write(STDOUT_FILENO, buffer, n) != n) {
            perror("写入失败");
            close(fd);
            return 1;
        }
    }

    if (n == -1) {
        perror("读取失败");
        close(fd);
        return 1;
```

```
    }

    close(fd);
    return 0;
}
```

---

# 2.7 API快速参考

## 文件偏移量

| 函数 | 功能 | 返回值 |
| --- | --- | --- |
| lseek(fd, offset, whence) | 设置文件偏移量 | 新偏移量或-1 |

**whence参数：**

- `SEEK_SET` - 从文件开头
- `SEEK_CUR` - 从当前位置
- `SEEK_END` - 从文件末尾

## 错误处理

| 函数/变量 | 说明 |
| --- | --- |
| errno | 全局错误码 |
| perror(msg) | 打印错误信息（推荐） |
| strerror(errno) | 获取错误描述字符串 |

## open标志

| 标志 | 说明 |
| --- | --- |
| O_RDONLY | 只读 |
| O_WRONLY | 只写 |
| O_RDWR | 读写 |
| O_CREAT | 不存在则创建 |
| O_TRUNC | 截断为0 |
| O_APPEND | 追加写入 |
| O_EXCL | 排他创建（需配合O_CREAT） |

---

# 2.8 本章总结

## 核心要点

1. **文件描述符机制**
   - fd是非负整数，标识打开的文件
   - 0/1/2 预留给 stdin/stdout/stderr
   - 每次open创建新的文件表项
2. **文件偏移量控制**
   - lseek()可以随机访问文件
   - 技巧：`lseek(fd, 0, SEEK_END)` 获取文件大小
   - 多次打开同一文件，offset独立
3. **错误处理**
   - 所有系统调用都要检查返回值
   - 使用perror()打印错误信息
   - errno记录错误码
4. **常用open标志**
   - O_TRUNC：清空文件重写
   - O_APPEND：追加到末尾（日志）
   - O_EXCL：防止覆盖已存在文件

## 学习建议

1. **必须养成的习惯**

```c
int fd = open(...);
if (fd == -1) {  // 必须检查!
    perror("open");
    return 1;
}
```

2. **实验验证**
   - 运行本章所有代码示例
   - 尝试修改参数观察效果
   - 用strace跟踪系统调用
3. **常见错误**
   - ❌ 忘记检查返回值
   - ❌ 不关闭文件描述符（资源泄漏）
   - ❌ 打开文件后不检查errno

# 第03章 深入理解I/O缓冲机制

> **本章目标**：理解Linux I/O的多层缓冲机制，掌握性能优化的关键技术

## 引言：一个性能问题引发的思考

### 实际场景

假设你正在编写一个日志收集程序，需要每秒写入10000条日志。你可能会写出这样的代码：

```
// 方案1：每条日志立即写入
for (int i = 0; i < 10000; i++) {
    int fd = open("log.txt", O_WRONLY | O_APPEND);
    write(fd, log_data, strlen(log_data));
    close(fd);
}

// 方案2：使用标准I/O
FILE *fp = fopen("log.txt", "a");
for (int i = 0; i < 10000; i++) {
    fprintf(fp, "%s\n", log_data);
}
fclose(fp);
```



**实测结果（写入10000条记录）**：

| 方案 | 耗时 | 系统调用次数 |
| --- | --- | --- |
| 方案1（系统调用） | ~8.5秒 | 30000次 |
| 方案2（标准I/O） | ~0.3秒 | ~150次 |

**性能相差28倍！为什么？**

答案就在**缓冲机制**。

## 本章解决的核心问题

1. ❓ 为什么需要缓冲？缓冲到底在哪里？
2. ❓ 用户空间缓冲（stdio buffer）和内核缓冲（page cache）有什么区别？
3. ❓ 如何控制缓冲行为？什么时候数据真正写入磁盘？
4. ❓ 什么是直接I/O？何时应该使用？
5. ❓ 混合使用系统调用和标准I/O要注意什么？

## 学完本章你将能够

- ✅ 理解Linux I/O的三层缓冲架构
- ✅ 根据场景选择合适的I/O方式
- ✅ 使用工具测试和优化I/O性能
- ✅ 避免缓冲导致的数据丢失问题
- ✅ 在实际项目中做出正确的技术选型

## 📂 配套代码示例

本章提供5个实践代码，位于 `src/chapter03/`：

| 代码文件 | 对应章节 | 功能说明 |
| --- | --- | --- |
| `buffer_observe.c` | 3.3节 | 观察stdio的三种缓冲模式 |
| `io_performance_test.c` | 3.7节 | 对比不同I/O方式的性能 |
| `direct_io_example.c` | 3.5节 | 演示直接I/O和对齐要求 |
| `fsync_example.c` | 3.4节 | 演示fsync/fdatasync/O_SYNC |
| `mix_io_example.c` | 3.6节 | 演示混合使用I/O的问题 |

**快速开始：**

```
# 编译所有示例
cd src/chapter03 && mkdir -p build && cd build
cmake .. && make

# 运行性能测试（推荐先运行这个）
./io_performance_test

# 观察缓冲行为（配合strace）
strace -e trace=write ./buffer_observe
```

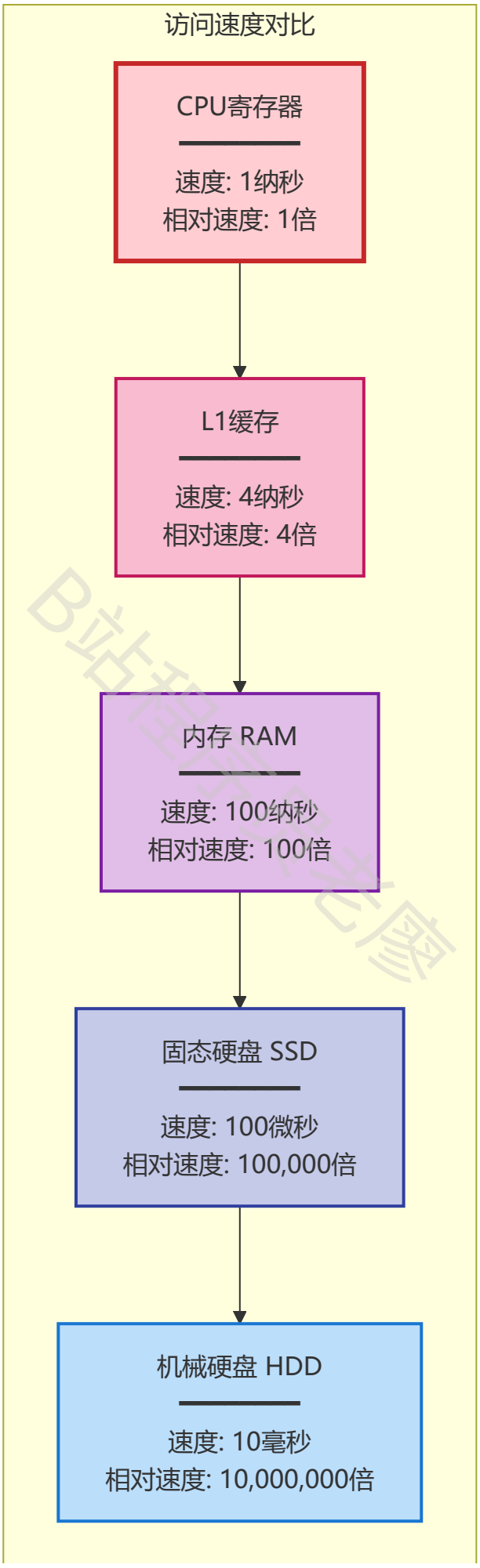


---

# 3.1 为什么需要缓冲？

## 3.1.1 硬件速度的巨大差异

现代计算机系统中，不同层级的硬件速度差异极大：


访问速度对比
CPU寄存器
速度: 1纳秒
相对速度: 1倍
L1缓存
速度: 4纳秒
相对速度: 4倍
内存 RAM
速度: 100纳秒
相对速度: 100倍
固态硬盘 SSD
速度: 100微秒
相对速度: 100,000倍
机械硬盘 HDD
速度: 10毫秒
相对速度: 10,000,000倍

**生活类比：**

假设CPU读取数据用1秒，那么：

- 内存需要 **1.5分钟**
- SSD需要 **1天多**
- 机械硬盘需要 **3个多月**

如果每次写入1字节都直接访问磁盘，程序性能将是灾难性的！

## 3.1.2 系统调用的开销

每次系统调用都需要：

1. **上下文切换**：用户态 → 内核态 → 用户态
2. **保存/恢复寄存器**：保存现场
3. **参数验证**：检查指针合法性
4. **权限检查**：检查文件访问权限

**实测数据（Intel i7）**：

```
// 测试：write 1字节 vs write 4KB
// 写入1MB数据

// 方式1：每次write 1字节
for (int i = 0; i < 1024*1024; i++) {
    write(fd, &byte, 1);  // 100万次系统调用
}
// 耗时：~1.8秒

// 方式2：每次write 4KB
for (int i = 0; i < 256; i++) {
    write(fd, buf, 4096);  // 256次系统调用
}
// 耗时：~0.003秒
```

**性能差距：600倍！**

## 3.1.3 缓冲的核心价值

缓冲机制通过**空间换时间**，实现：

| 优化点 | 具体效果 |
| --- | --- |
| 减少系统调用次数 | 从N次降低到1次 |
| 批量I/O | 一次写入4KB而非1字节 |
| 顺序访问优化 | 磁盘顺序读写比随机快100倍 |

| 优化点 | 具体效果 |
| --- | --- |
| 预读取 | 提前加载可能访问的数据 |

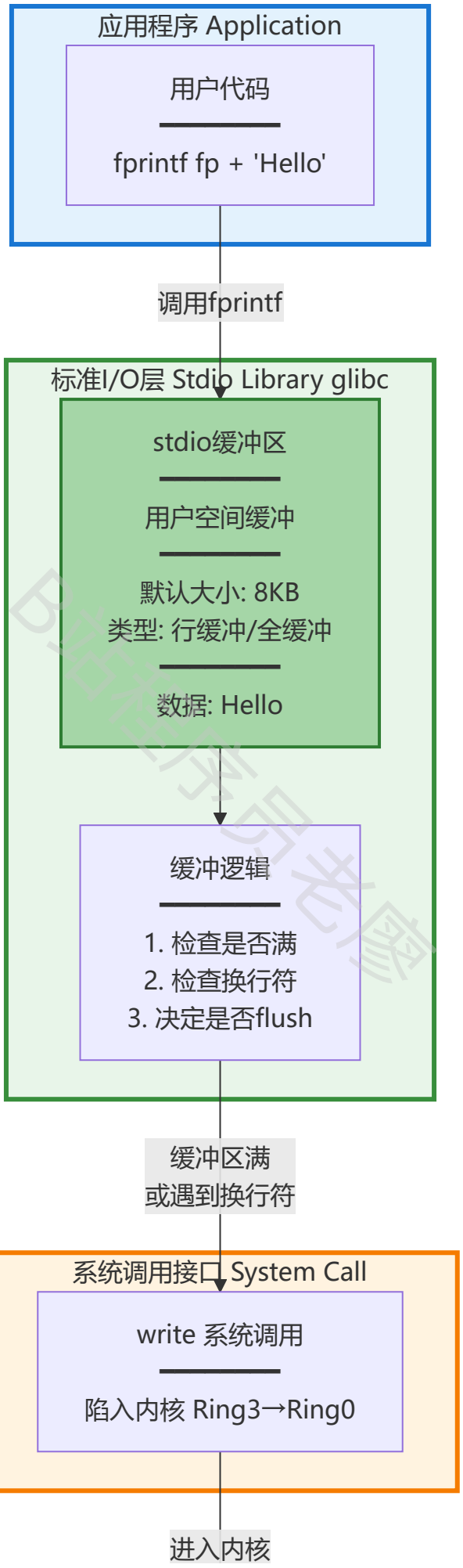
**结论**：缓冲是I/O性能优化的基石。

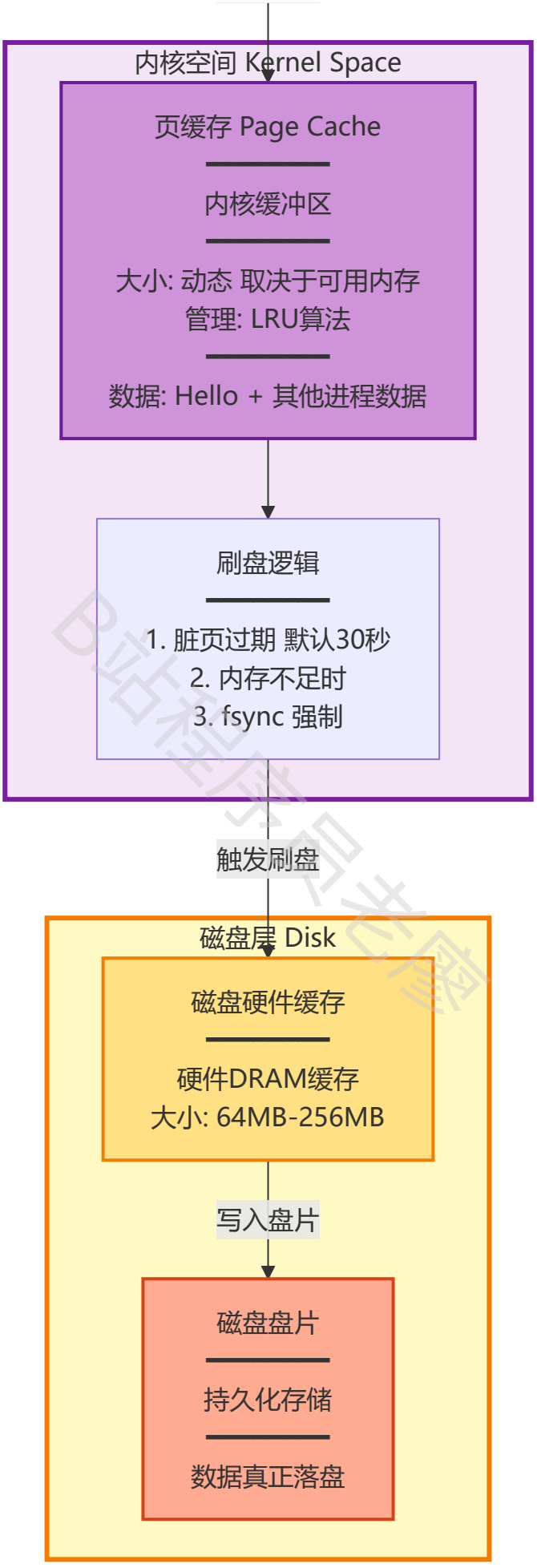
---

# 3.2 Linux I/O的三层缓冲架构

## 3.2.1 完整的数据流转路径

从应用程序写入数据到真正落盘，要经过三层缓冲：


应用程序 Application
用户代码
fprintf fp + 'Hello'
调用fprintf
标准I/O层 Stdio Library glibc
stdio缓冲区
用户空间缓冲
默认大小: 8KB
类型: 行缓冲/全缓冲
数据: Hello
缓冲逻辑
1. 检查是否满
2. 检查换行符
3. 决定是否flush
缓冲区满
或遇到换行符
系统调用接口 System Call
write 系统调用
陷入内核 Ring3→Ring0
进入内核

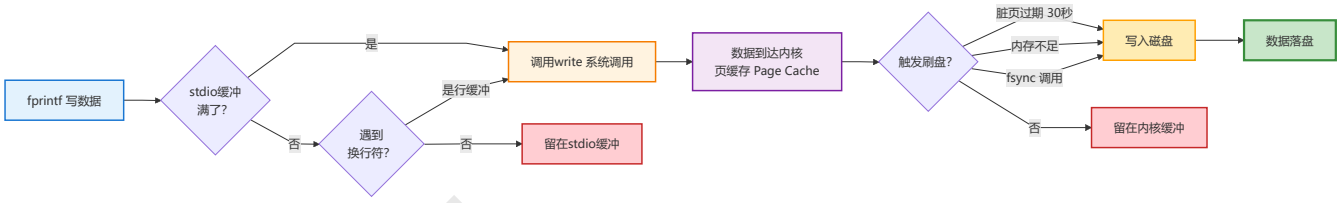

内核空间 Kernel Space
页缓存 Page Cache
内核缓冲区
大小: 动态 取决于可用内存
管理: LRU算法
数据: Hello + 其他进程数据
刷盘逻辑
1. 脏页过期 默认30秒
2. 内存不足时
3. fsync 强制
触发刷盘
磁盘层 Disk
磁盘硬件缓存
硬件DRAM缓存
大小: 64MB-256MB
写入盘片
磁盘盘片
持久化存储
数据真正落盘

### 3.2.2 三层缓冲的职责划分

| 缓冲层 | 位置 | 大小 | 管理者 | 主要作用 |
| --- | --- | --- | --- | --- |
| **stdio缓冲** | 用户空间 | 8KB（默认） | glibc库 | 减少系统调用次数 |
| **页缓存** | 内核空间 | 动态（几MB到几GB） | Linux内核 | 统一缓存、提升整体性能 |
| **磁盘缓存** | 硬件 | 64-256MB | 磁盘控制器 | 加速磁盘读写 |

### 3.2.3 数据何时真正落盘?

从调用 `fprintf()` 到数据真正写入磁盘，需要经过多个条件：




**重要结论**：

- ⚠️ `fprintf()` 返回 ≠ 数据已落盘
- ⚠️ `write()` 返回 ≠ 数据已落盘
- ✅ `fsync()` 返回 = 数据已落盘（大概率，除非硬件缓存未刷）

---

## 3.3 stdio缓冲区详解

### 3.3.1 三种缓冲模式

标准I/O提供三种缓冲模式：

| 模式 | 宏定义 | 刷新时机 | 典型应用 |
| --- | --- | --- | --- |
| **全缓冲** | `_IOFBF` | 缓冲区满 | 普通文件 |
| **行缓冲** | `_IOLBF` | 遇到换行符 | 终端输入输出 |
| **无缓冲** | `_IONBF` | 立即刷新 | 标准错误stderr |

**默认行为**：

```
// 系统自动设置：
// - stdout（终端）：行缓冲
// - stdout（重定向到文件）：全缓冲
// - stderr：无缓冲
// - 普通文件：全缓冲
```

## 3.3.2 setvbuf() - 配置缓冲模式

**函数原型：**

```
#include <stdio.h>

int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

**参数说明：**

| 参数 | 类型 | 说明 | 示例 |
| --- | --- | --- | --- |
| stream | FILE* | 文件指针 | fp |
| buf | char* | 自定义缓冲区（NULL=自动分配） | my_buffer 或 NULL |
| mode | int | 缓冲模式 | `_IOFBF` / `_IOLBF` / `_IONBF` |
| size | size_t | 缓冲区大小 | 4096 |

**返回值：**

- 成功：0
- 失败：非0

**常用示例：**

```
// 示例1：设置全缓冲，使用默认大小
FILE *fp = fopen("test.txt", "w");
setvbuf(fp, NULL, _IOFBF, 0);

// 示例2：设置无缓冲（每次写入立即调用write）
setvbuf(fp, NULL, _IONBF, 0);

// 示例3：自定义缓冲区
char my_buf[4096];
setvbuf(fp, my_buf, _IOFBF, 4096);

// 示例4：将stdout改为全缓冲
setvbuf(stdout, NULL, _IOFBF, 8192);
```



## 3.3.3 fflush() - 手动刷新缓冲

**函数原型：**

```
#include <stdio.h>

int fflush(FILE *stream);
```

**参数说明：**

| 参数 | 说明 |
| --- | --- |
| stream | 文件指针（NULL = 刷新所有打开的输出流） |

**返回值：**

- 成功：0
- 失败：EOF（-1）

**使用场景：**

```
// 场景1：确保日志立即输出
fprintf(log_fp, "[ERROR] Something wrong!\n");
fflush(log_fp);  // 立即刷新，避免程序崩溃丢失日志

// 场景2：交互式提示
printf("请输入密码：");  // 没有换行符
fflush(stdout);          // 强制显示提示
scanf("%s", password);

// 场景3：刷新所有输出
fflush(NULL);  // 刷新stdout、文件等所有输出流
```



## 3.3.4 自动刷新的时机

除了手动`fflush()`，以下情况会自动刷新：

| 触发条件 | 说明 |
| --- | --- |
| 程序正常退出 | exit()、return |
| 调用fclose() | 关闭文件时 |
| 缓冲区满 | 数据填满缓冲区 |
| 行缓冲遇到`\n` | printf("...\n") |
| 程序异常退出 | _exit()、abort()、信号杀死 |

**💻 实践代码：** `src/chapter03/buffer_observe.c`

这个示例演示了三种缓冲模式的实际行为，建议使用`strace`观察：

```
cd src/chapter03/build
strace -e trace=write ./buffer_observe
```

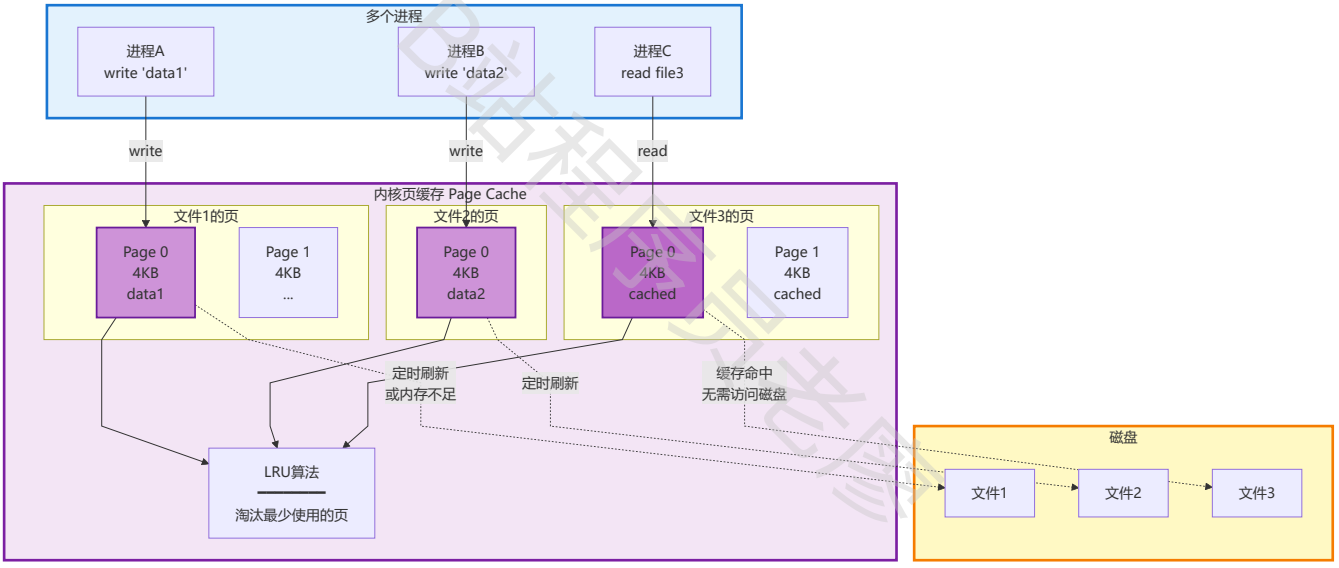
**危险示例**：

```c
// 这段代码可能丢失日志！
FILE *log = fopen("app.log", "w");
fprintf(log, "程序启动\n");
fprintf(log, "正在处理...");  // 没有\n，留在缓冲区

// 程序突然崩溃（段错误、被kill -9）
// "正在处理..." 永远不会写入文件！

// 正确做法：关键日志立即刷新
fprintf(log, "正在处理...");
fflush(log);  // 或者加 \n
```

# 3.4 内核缓冲区（Page Cache）

## 3.4.1 Page Cache的工作原理

内核维护一个统一的**页缓存（Page Cache）**，缓存所有文件I/O：




**Page Cache的优势：**

1. **统一管理**：所有进程共享缓存
2. **预读优化**：顺序读取时自动预读后续页
3. **写合并**：多次小写入合并成一次大写入
4. **缓存命中**：重复读取无需访问磁盘

**内核刷盘参数（可配置）：**

Linux内核通过以下参数控制页缓存的刷新行为：

| 参数 | 默认值 | 说明 |
| --- | --- | --- |
| `/proc/sys/vm/dirty_expire_centisecs` | 3000 (30秒) | 脏页在内存中的最长保留时间 |
| `/proc/sys/vm/dirty_writeback_centisecs` | 500 (5秒) | 内核刷新线程的唤醒间隔 |

| 参数 | 默认值 | 说明 |
| --- | --- | --- |
| `/proc/sys/vm/dirty_ratio` | 20 (20%) | 脏页占总内存的最大比例 |
| `/proc/sys/vm/dirty_background_ratio` | 10 (10%) | 后台开始刷新的脏页比例 |

**查看和修改参数：**

```
# 查看当前配置
cat /proc/sys/vm/dirty_expire_centisecs        # 输出：3000（30秒）
cat /proc/sys/vm/dirty_writeback_centisecs     # 输出：500（5秒）

# 临时修改（重启后失效）
echo 1500 | sudo tee /proc/sys/vm/dirty_expire_centisecs  # 改为15秒

# 永久修改（编辑 /etc/sysctl.conf）
sudo sh -c 'echo "vm.dirty_expire_centisecs = 1500" >> /etc/sysctl.conf'
sudo sysctl -p  # 使配置生效
```

**工作机制：**

1. **pdflush/flush内核线程**每隔`dirty_writeback_centisecs`（5秒）唤醒一次
2. 检查脏页（已修改但未写入磁盘的页）
3. 将超过`dirty_expire_centisecs`（30秒）的脏页写入磁盘
4. 或当脏页超过`dirty_ratio`时强制刷新

## 3.4.2 fsync() - 强制刷入磁盘

**函数原型：**

```
#include <unistd.h>

int fsync(int fd);
int fdatasync(int fd);
void sync(void);
```

**函数对比：**

| 函数 | 刷新内容 | 性能 | 使用场景 |
| --- | --- | --- | --- |
| fsync(fd) | 数据 + 元数据 | 慢 | 数据库事务提交 |
| fdatasync(fd) | 仅数据 | 较快 | 日志文件写入 |
| sync() | 所有文件 | 最慢 | 系统关机前 |

**📖 什么是数据和元数据？**

| 类型 | 含义 | 包含内容 | 举例 |
| --- | --- | --- | --- |
| **数据**<br>(Data) | 文件的实际内容 | 文件中存储的字节流 | • 文本文件的文字<br>• 图片的像素数据<br>• 数据库记录 |
| **元数据**<br>(Metadata) | 描述文件的信息 | • 文件大小 (size)<br>• 修改时间 (mtime)<br>• 访问时间 (atime)<br>• 权限 (mode)<br>• 所有者 (uid/gid)<br>• inode信息 | 执行`ls -l`看到的信息就是元数据 |

**为什么区分数据和元数据?**

```
// 示例：写入文件
int fd = open("log.txt", O_WRONLY | O_APPEND);
write(fd, "新日志\n", 9);  // 修改了数据
```

执行这个操作后:

- ✅ **数据改变**：文件内容增加了"新日志\n"
- ✅ **元数据改变**：文件大小增加了9字节，修改时间更新

**fsync vs fdatasync 的区别:**

```
// 情况1：使用fsync（慢，但更安全）
write(fd, data, size);
fsync(fd);
// ✓ 数据落盘
// ✓ 元数据落盘（文件大小、修改时间等）

// 情况2：使用fdatasync（快）
write(fd, data, size);
fdatasync(fd);
// ✓ 数据落盘
// X 元数据可能不落盘（除非影响数据读取）
```

**何时元数据很重要?**

| 场景 | 数据丢失影响 | 元数据丢失影响 | 推荐 |
| --- | --- | --- | --- |
| 数据库事务 | ❌ 致命 | ❌ 可能导致数据不一致 | fsync |
| 追加日志 | ❌ 日志丢失 | ⚠️ 文件大小错误但数据在 | fdatasync |
| 创建新文件 | ❌ 文件丢失 | ❌ 文件可能"不存在" | fsync |
| 覆盖写入 | ❌ 数据错误 | ✅ 元数据不变，影响小 | fdatasync |

**性能差异:**

fdatasync通常比fsync快**10-30%**，因为：

- 磁盘写入次数更少（不写inode）
- 元数据通常在不同的磁盘位置，需要额外的磁盘寻道

**参数说明：**

| 参数 | 类型 | 说明 |
| --- | --- | --- |
| fd | int | 文件描述符 |

**返回值：**

- 成功：0
- 失败：-1

**使用示例：**

```
// 示例1：确保数据库事务已落盘
int fd = open("db.dat", O_WRONLY);
write(fd, transaction_data, size);
fsync(fd);  // 阻塞，直到数据真正写入磁盘
close(fd);

// 示例2：日志写入（不关心元数据）
int log_fd = open("app.log", O_WRONLY | O_APPEND);
write(log_fd, log_msg, strlen(log_msg));
fdatasync(log_fd);  // 比fsync快，因为不刷新元数据
close(log_fd);

// 示例3：拔U盘前
sync();  // 刷新所有未写入的数据
// 现在可以安全拔出U盘
```



💻 **实践代码：** `src/chapter03/fsync_example.c`

这个示例对比了不同同步方式的性能和安全性：

```
cd src/chapter03/build
./fsync_example  # 查看不同同步方式的耗时对比
```

## 3.4.3 O_SYNC 和 O_DSYNC 标志

通过`open()`标志控制每次`write()`的同步行为：

**标志说明：**

| 标志 | 等价于 | 性能影响 | 使用场景 |
| --- | --- | --- | --- |
| O_SYNC | 每次write后调用fsync | 极大降低性能 | 数据绝对不能丢失 |
| O_DSYNC | 每次write后调用fdatasync | 降低性能 | 数据重要但元数据可延迟 |

**示例代码：**

```
// 普通写入：数据停留在内核缓冲区
int fd = open("file.txt", O_WRONLY | O_CREAT, 0644);
write(fd, data, size);  // 快速返回
close(fd);

// O_SYNC写入：每次write都等待落盘
int fd = open("file.txt", O_WRONLY | O_CREAT | O_SYNC, 0644);
write(fd, data, size);  // 慢，等待磁盘I/O完成
close(fd);
```

**性能测试对比**（写入1MB）：

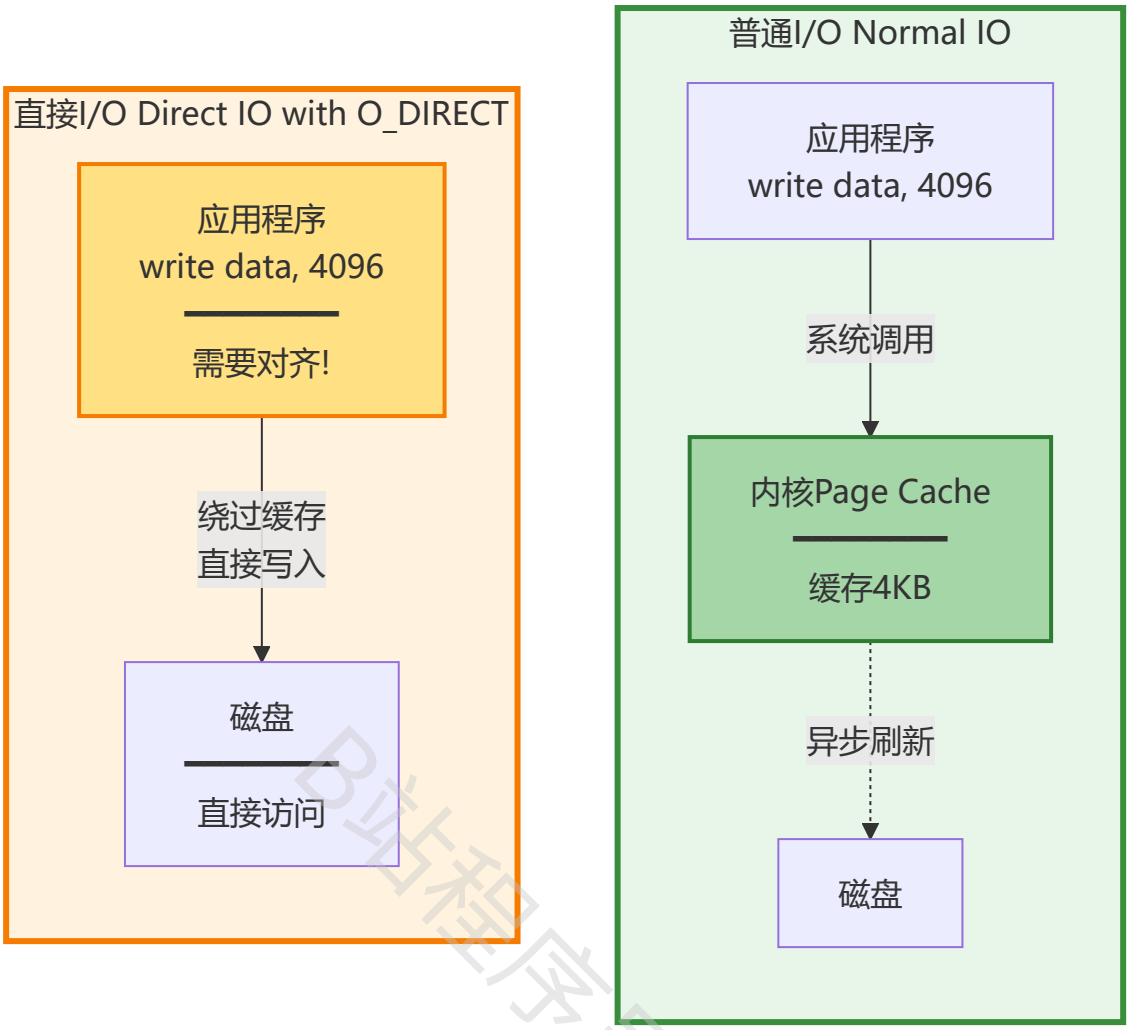
| 方式 | 耗时 | 说明 |
| --- | --- | --- |
| 普通write | 0.003秒 | 仅写入内核缓冲 |
| write + fsync | 0.15秒 | 手动刷盘 |
| O_SYNC | 0.15秒 | 每次write自动刷盘 |



# 3.5 直接I/O（O_DIRECT）

## 3.5.1 什么是直接I/O?

**直接I/O**绕过内核页缓存，直接在用户空间缓冲区和磁盘之间传输数据：






**直接I/O的特点：**

| 特点 | 说明 |
| --- | --- |
| ✅ 避免内核缓存 | 数据直接到磁盘 |
| ✅ 节省内存 | 不占用页缓存 |
| ✅ 适合大文件顺序访问 | 数据库、视频流 |
| ❌ 性能可能更差 | 失去缓存优化 |
| ❌ 有对齐限制 | 必须按扇区对齐 |

## 3.5.2 对齐限制

使用`O_DIRECT`必须满足：

| 限制项 | 要求 |
| --- | --- |
| 缓冲区地址 | 必须按扇区大小对齐（通常512或4096字节） |
| 写入大小 | 必须是扇区大小的整数倍 |

| 限制项 | 要求 |
| --- | --- |
| 文件偏移量 | 必须是扇区大小的整数倍 |

**错误示例**:

```
int fd = open("file.txt", O_WRONLY | O_CREAT | O_DIRECT, 0644);

char buf[1000];  // ❌ 大小不是4096的倍数
write(fd, buf, 1000);  // 失败：Invalid argument

char *ptr = malloc(4096);  // ❌ malloc返回的地址可能未对齐
write(fd, ptr, 4096);  // 可能失败
```

**正确示例**:

```
// 方法1：使用posix_memalign分配对齐内存
char *buf;
posix_memalign((void**)&buf, 4096, 4096);  // 4096字节对齐

int fd = open("file.txt", O_WRONLY | O_CREAT | O_DIRECT, 0644);
write(fd, buf, 4096);  // ✅ 成功
free(buf);

// 方法2：使用__attribute__对齐
char buf[4096] __attribute__((aligned(4096)));
write(fd, buf, 4096);  // ✅ 成功
```

💻 **实践代码**: `src/chapter03/direct_io_example.c`

这个示例演示了直接I/O的对齐要求和性能对比:

```
cd src/chapter03/build
./direct_io_example  # 对比普通I/O和直接I/O
```

## 3.5.3 何时使用直接I/O?

| 场景 | 是否适合 | 原因 |
| --- | --- | --- |
| 数据库管理系统 | ✅ 适合 | 自己管理缓存，避免双重缓存 |
| 视频流服务 | ✅ 适合 | 大文件顺序读取，缓存无益 |
| 磁盘性能测试 | ✅ 适合 | 需要真实的磁盘I/O性能 |
| 小文件随机访问 | ❌ 不适合 | 页缓存能大幅提升性能 |
| 日志文件 | ❌ 不适合 | 小数据频繁写入 |

# 3.6 混合使用系统调用和标准I/O

## 3.6.1 fileno() 和 fdopen()

在同一文件上混合使用系统调用和标准I/O：

**函数原型：**

```
#include <stdio.h>

int fileno(FILE *stream);        // FILE* → fd
FILE *fdopen(int fd, const char *mode);  // fd → FILE*
```

**参数说明：**

| 函数 | 参数 | 返回值 |
| --- | --- | --- |
| fileno() | FILE* | 文件描述符fd（-1表示失败） |
| fdopen() | fd + mode | FILE*（NULL表示失败） |

**示例代码：**

```
// FILE* → fd
FILE *fp = fopen("test.txt", "r");
int fd = fileno(fp);
read(fd, buf, 100);  // 用系统调用读取

// fd → FILE*
int fd = open("test.txt", O_RDONLY);
FILE *fp = fdopen(fd, "r");
fgets(buf, 100, fp);  // 用标准I/O读取
```

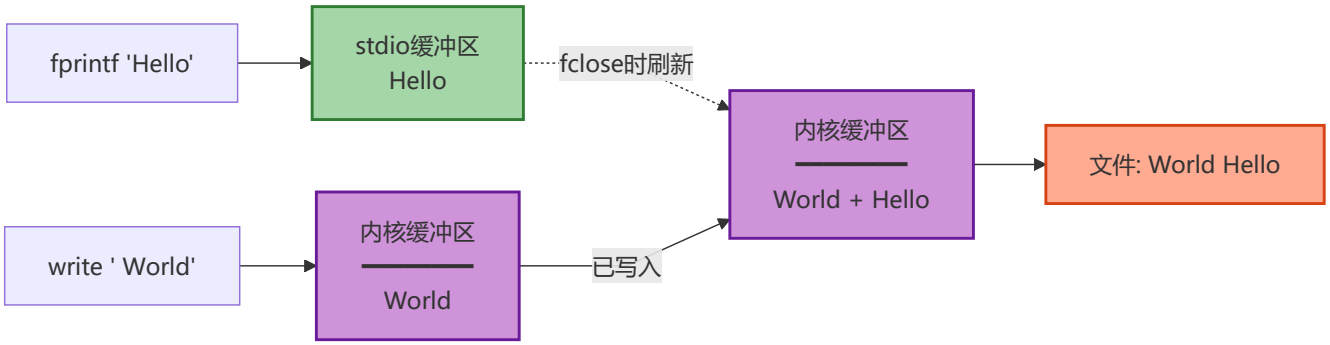


## 3.6.2 缓冲同步问题

**危险示例**：

```
FILE *fp = fopen("test.txt", "w");
int fd = fileno(fp);

fprintf(fp, "Hello");   // 数据在stdio缓冲区
write(fd, " World", 6); // 直接写入内核缓冲区

fclose(fp);
// 文件内容可能是 " WorldHello" 而非 "Hello World"!
```

**问题原因**：




**正确做法**:

```
FILE *fp = fopen("test.txt", "w");
int fd = fileno(fp);

fprintf(fp, "Hello");
fflush(fp);             // ⚠️ 先刷新stdio缓冲
write(fd, " World", 6);

fclose(fp);
// 现在文件内容正确：Hello World
```

**最佳实践**:

| 建议 | 说明 |
| --- | --- |
| ✅ 尽量不混合使用 | 选择一种I/O方式并坚持 |
| ✅ 混合时手动fflush | 确保stdio缓冲已刷新 |
| ✅ 理解两层缓冲 | stdio缓冲 + 内核缓冲 |
| ❌ 不要假设写入顺序 | 缓冲可能打乱顺序 |

**💻 实践代码：** `src/chapter03/mix_io_example.c`

这个示例演示了混合使用的正确和错误方式:

```
cd src/chapter03/build
./mix_io_example  # 观察混合I/O时的顺序问题
cat test1.txt test2.txt  # 查看文件内容对比
```

---

# 3.7 综合实验：性能对比

***注意：这里的测试结果只是作为参考，以自己实际测试为准。***

## 3.7.1 实验设计

我们将测试6种I/O方式写入10MB数据的性能：

| 方式 | 代码特征 | 预期性能 |
| --- | --- | --- |
| 1. 系统调用 1字节/次 | write(fd, &byte, 1) | 最慢 |
| 2. 系统调用 4KB/次 | write(fd, buf, 4096) | 较快 |
| 3. 系统调用 + O_SYNC | O_SYNC标志 | 很慢 |
| 4. 标准I/O fprintf | fprintf(fp, ...) | 快 |
| 5. 标准I/O fwrite | fwrite(buf, 1, 4096, fp) | 最快 |
| 6. 直接I/O | O_DIRECT标志 | 慢（但不占内存） |

💻 **实践代码：** `src/chapter03/io_performance_test.c`

运行性能测试：

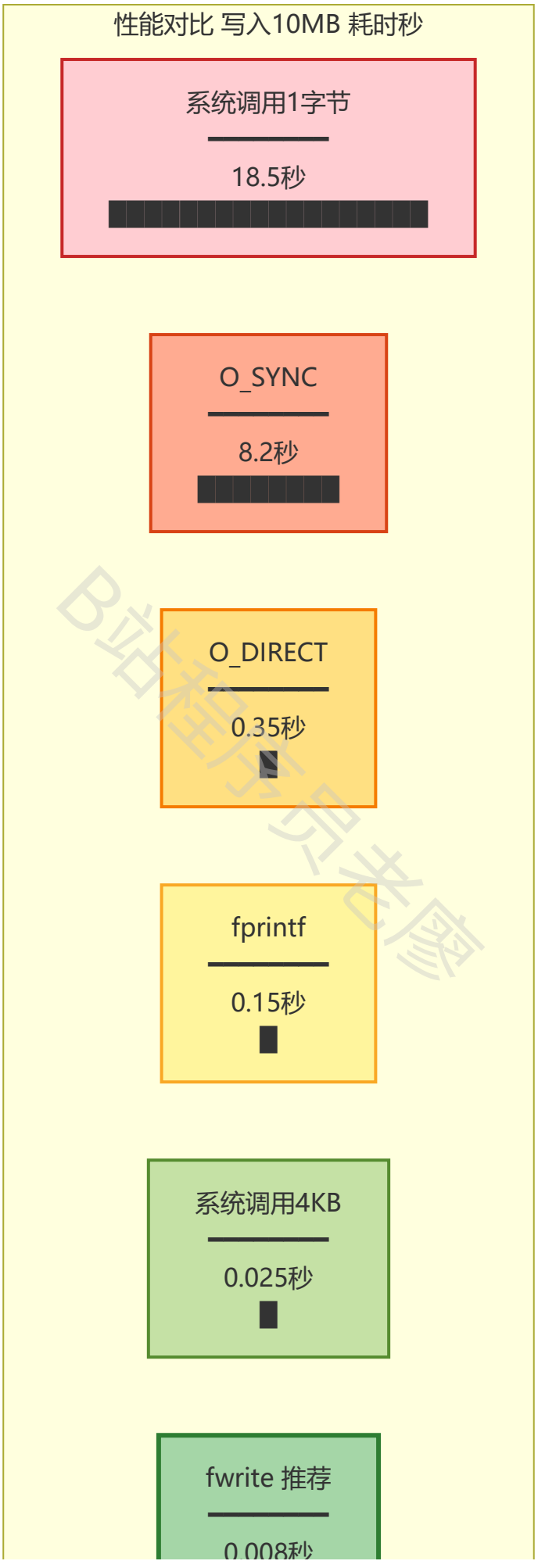
```
cd src/chapter03/build
./io_performance_test  # 查看完整性能对比
```

## 3.7.2 实测结果

**测试环境**：Ubuntu 22.04 + SSD + 16GB RAM

| 方式 | 耗时 | 系统调用次数 | 相对性能 |
| --- | --- | --- | --- |
| 系统调用 1字节 | 18.5秒 | 10,485,760 | 基准×1 |
| 系统调用 4KB | 0.025秒 | 2,560 | 740倍 ⬆️ |
| O_SYNC | 8.2秒 | 2,560 | 2.3倍 ⬆️ |
| fprintf | 0.15秒 | ~150 | 123倍 ⬆️ |
| fwrite | 0.008秒 | ~1,280 | 2,312倍 ⬆️ |
| O_DIRECT | 0.35秒 | 2,560 | 53倍 ⬆️ |

**可视化对比**（仅作为性能区别的参考）：

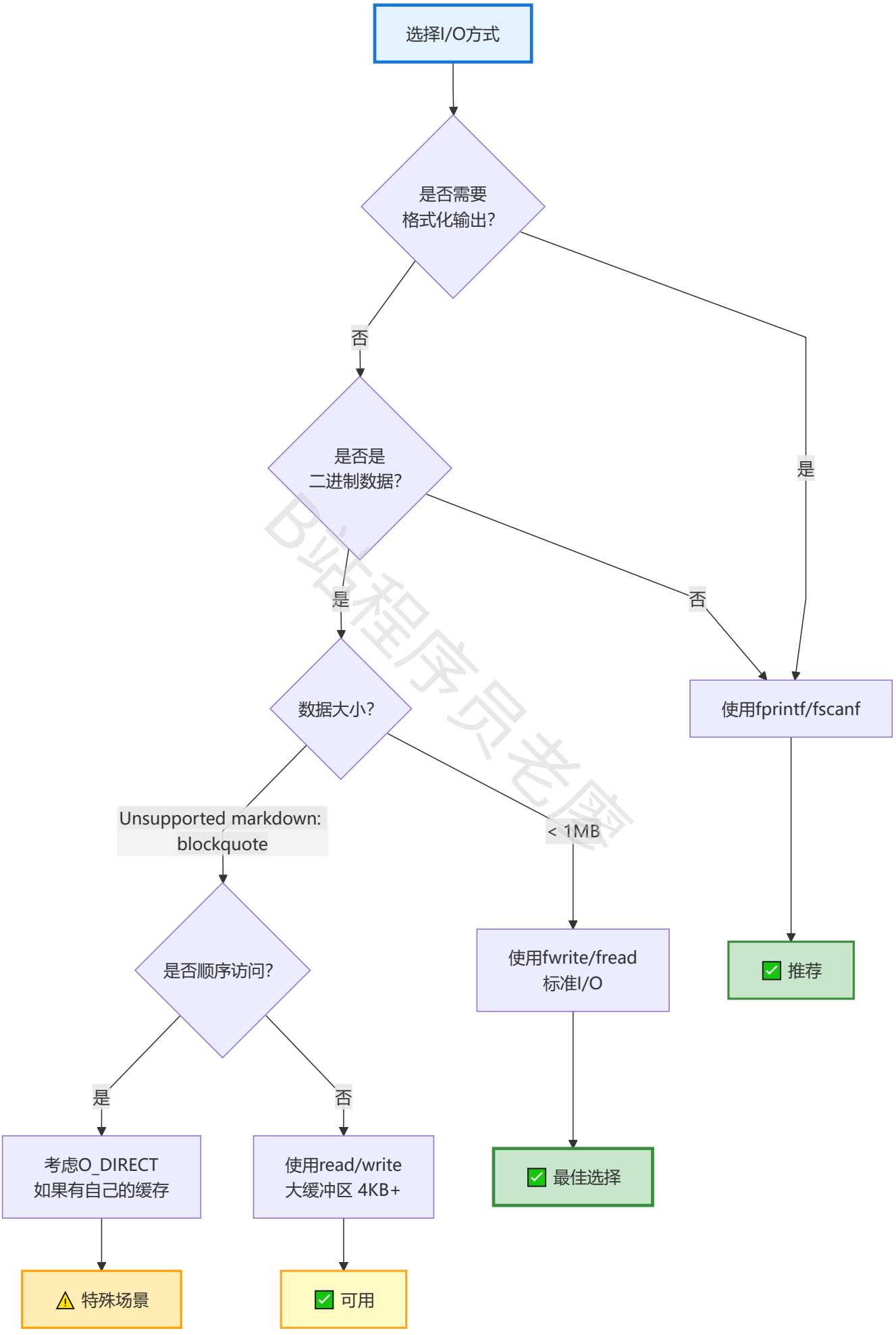

性能对比 写入10MB 耗时秒
系统调用1字节
18.5秒
O_SYNC
8.2秒
O_DIRECT
0.35秒
fprintf
0.15秒
系统调用4KB
0.025秒
fwrite 推荐
0.008秒

## 3.7.3 结论与建议

**性能排名**（快 → 慢）：

1. 🥇 **fwrite（标准I/O + 全缓冲）** - 推荐用于普通文件
2. 🥈 **系统调用 + 大缓冲区（4KB+）** - 推荐用于特殊需求
3. 🥉 **fprintf（标准I/O）** - 推荐用于格式化输出
4. ⚠️ **O_DIRECT** - 仅用于数据库、大文件流
5. ⚠️ **O_SYNC** - 仅用于关键数据
6. ❌ **小缓冲区系统调用** - 避免使用

**选择指南**：


选择I/O方式
是否需要
格式化输出?
否
是
是否是
二进制数据?
是
否
数据大小?
使用fprintf/fscanf
Unsupported markdown: blockquote
< 1MB
是否顺序访问?
使用fwrite/fread
标准I/O
✅ 推荐
是
否
考虑O_DIRECT
如果有自己的缓存
使用read/write
大缓冲区 4KB+
✅ 最佳选择
⚠ 特殊场景
✅ 可用

# 3.8 使用strace分析I/O行为

## 3.8.1 strace工具介绍

`strace`可以跟踪程序的系统调用：

```
# 基本用法
strace ./my_program

# 只显示文件I/O相关调用
strace -e trace=open,read,write,close ./my_program

# 统计系统调用次数
strace -c ./my_program

# 输出到文件
strace -o trace.log ./my_program
```

## 3.8.2 实战分析

**示例程序**（使用fprintf）：

```
// test_fprintf.c
#include <stdio.h>

int main() {
    FILE *fp = fopen("test.txt", "w");

    for (int i = 0; i < 10; i++) {
        fprintf(fp, "Line %d\n", i);
    }

    fclose(fp);
    return 0;
}
```

**strace输出分析**：

```
$ strace -e trace=open,write,close ./test_fprintf

openat(AT_FDCWD, "test.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
write(3, "Line 0\nLine 1\nLine 2\nLine 3\nLi"..., 70) = 70
close(3) = 0
```

**关键发现**：

- ✅ 调用了10次`fprintf()`
- ✅ 但只有**1次**`write()`系统调用

- ✅ 数据被合并成一次写入

**对比：使用write()**

```
int fd = open("test.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
for (int i = 0; i < 10; i++) {
    char buf[20];
    int len = sprintf(buf, "Line %d\n", i);
    write(fd, buf, len);
}
close(fd);
$ strace -e trace=write ./test_write

write(3, "Line 0\n", 7) = 7
write(3, "Line 1\n", 7) = 7
write(3, "Line 2\n", 7) = 7
... (共10次write)
```

- ❌ 10次`write()`系统调用
- ❌ 性能比fprintf差

💡 **实践建议：**

使用strace分析本章的所有示例代码，观察实际的系统调用行为：

```
cd src/chapter03/build

# 分析buffer_observe
strace -e trace=write -c ./buffer_observe

# 分析性能测试
strace -e trace=write,open,close -o trace.log ./io_performance_test
grep "write(" trace.log | wc -l  # 统计write调用次数

# 分析混合I/O
strace -e trace=write ./mix_io_example
```



---

# 3.9 本章总结

## 3.9.1 核心知识点回顾

**三层缓冲架构**：

```
应用程序
    ↓
stdio缓冲（8KB，glibc管理）
    ↓
系统调用（用户态 → 内核态）
    ↓
页缓存（动态大小，内核管理）
    ↓
磁盘I/O
```

**关键API**：

| API | 作用 | 层次 |
| --- | --- | --- |
| setvbuf() | 配置stdio缓冲 | 用户空间 |
| fflush() | 刷新stdio缓冲 | 用户空间 |
| fsync() | 刷新页缓存 | 内核空间 |
| O_SYNC | 每次write自动刷盘 | 内核空间 |
| O_DIRECT | 绕过页缓存 | 直接I/O |

**性能优化建议**：

1. ✅ 优先使用标准I/O（fread/fwrite/fprintf）
2. ✅ 系统调用使用大缓冲区（≥4KB）
3. ✅ 关键数据及时fflush()或fsync()
4. ✅ 避免频繁的小I/O
5. ⚠️ 特殊场景才考虑O_DIRECT

## 3.9.2 常见陷阱

| 陷阱 | 后果 | 解决方法 |
| --- | --- | --- |
| 程序崩溃丢失缓冲数据 | 日志/数据丢失 | 关键数据立即fflush() |
| 混合使用fd和FILE* | 数据顺序错乱 | 手动fflush()同步 |
| 忘记fsync() | 断电数据丢失 | 事务提交后fsync() |
| O_DIRECT未对齐 | write()失败 | 使用posix_memalign() |

### 3.9.3 调试技巧

```
# 1. 查看系统调用
strace -e trace=file ./my_program

# 2. 统计I/O次数
strace -c ./my_program | grep "write\|read"

# 3. 查看页缓存使用情况
free -h  # 查看cached列

# 4. 清空页缓存（需要root）
echo 3 > /proc/sys/vm/drop_caches
```

---

## 3.10 API快速参考

### stdio缓冲控制

| 函数 | 功能 | 返回值 | 头文件 |
| --- | --- | --- | --- |
| setvbuf(stream, buf, mode, size) | 设置缓冲模式 | 0或非0 | stdio.h |
| fflush(stream) | 刷新stdio缓冲 | 0或EOF | stdio.h |
| setbuf(stream, buf) | 简化版setvbuf | void | stdio.h |

**mode取值**:

- `_IOFBF` - 全缓冲
- `_IOLBF` - 行缓冲
- `_IONBF` - 无缓冲

### 内核缓冲控制

| 函数 | 功能 | 返回值 | 头文件 |
| --- | --- | --- | --- |
| fsync(fd) | 刷新数据+元数据 | 0或-1 | unistd.h |
| fdatasync(fd) | 仅刷新数据 | 0或-1 | unistd.h |
| sync() | 刷新所有文件 | void | unistd.h |

### open标志

| 标志 | 说明 | 性能影响 |
| --- | --- | --- |
| O_SYNC | 每次write同步 | 极大降低 |
| O_DSYNC | 数据同步（不含元数据） | 降低 |

| 标志 | 说明 | 性能影响 |
| --- | --- | --- |
| O_DIRECT | 绕过页缓存 | 可能降低 |

### 文件描述符与FILE*互转

| 函数 | 转换方向 | 返回值 | 头文件 |
| --- | --- | --- | --- |
| fileno(stream) | FILE* → fd | fd或-1 | stdio.h |
| fdopen(fd, mode) | fd → FILE* | FILE*或NULL | stdio.h |

### 内存对齐

| 函数 | 功能 | 头文件 |
| --- | --- | --- |
| posix_memalign(ptr, alignment, size) | 分配对齐内存 | stdlib.h |

## 3.11 实战练习

### 练习1：观察缓冲行为 ⭐

**任务：** 修改`src/chapter03/buffer_observe.c`，观察不同缓冲模式的输出时机。

```
cd src/chapter03/build

# 1. 正常运行观察
./buffer_observe

# 2. 使用strace观察write系统调用
strace -e trace=write ./buffer_observe 2>&1 | grep "write(3"

# 3. 修改代码：将缓冲区大小从16改为64，观察变化
# 编辑buffer_observe.c，重新编译运行
```

**思考题：**

- 全缓冲模式下，为什么第三次fprintf才触发write？
- 行缓冲模式下，不输出`\n`会怎样？

### 练习2：性能测试 ⭐⭐

**任务：** 运行`src/chapter03/io_performance_test.c`，对比不同I/O方式的性能。

```
cd src/chapter03/build

# 1. 基本测试
./io_performance_test

# 2. 修改TEST_SIZE，测试不同数据量
# 编辑io_performance_test.c: #define TEST_SIZE (10 * 1024 * 1024)
# 重新编译运行

# 3. 清空缓存后测试（需要root）
sudo sync
echo 3 | sudo tee /proc/sys/vm/drop_caches
./io_performance_test
```

**思考题：**

- 为什么fwrite比系统调用快？
- O_SYNC为什么这么慢？

## 练习3：strace分析 ⭐⭐⭐

**任务：** 使用strace分析程序的I/O行为。

```
cd src/chapter03/build

# 1. 统计系统调用次数
strace -c ./io_performance_test

# 2. 只看文件I/O相关调用
strace -e trace=open,read,write,close,fsync ./fsync_example

# 3. 输出到文件分析
strace -o trace.log ./mix_io_example
less trace.log
```

**进阶任务：**

- 分析自己之前写的程序
- 找出I/O瓶颈
- 尝试优化

## 练习4：实现安全日志 ⭐⭐⭐

**任务：** 编写一个日志函数，确保关键日志不会因程序崩溃而丢失。

```
// 在src/chapter03/创建safe_log.c

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdarg.h>
```

```c
#include <time.h>

// TODO: 实现一个安全的日志函数
void safe_log(const char *level, const char *format, ...) {
    // 提示：
    // 1. 使用O_APPEND打开文件
    // 2. 格式化消息（包含时间戳、级别）
    // 3. 如果level是"CRITICAL"，立即fsync
    // 4. 否则，定期fflush
}

int main() {
    safe_log("INFO", "程序启动");
    safe_log("CRITICAL", "发现严重错误: %d", 404);
    // 故意崩溃测试
    // int *p = NULL; *p = 1;
}
```

**验证：**

- 程序正常退出，日志是否完整？
- 程序异常退出（注释掉的崩溃代码），CRITICAL日志是否保存？

---

# 3.12 扩展阅读

## 3.12.1 推荐资源

1. 《Linux内核设计与实现》- 第16章：页缓存和页回写
2. Linux Man Pages:
    - `man 2 fsync`
    - `man 2 open` (搜索O_DIRECT)
    - `man 3 setvbuf`
3. 在线资源：
    - Linux I/O原理与优化

## 3.12.2 什么是脏页？

“脏页”（Dirty Page）是 Linux 内存管理和文件 I/O 系统中的一个核心概念。我们来用通俗易懂的方式解释它。

> **脏页 = 被修改过、但尚未写入磁盘的内存页**

我们一步步拆解：

### Linux 读写文件时，并不直接操作磁盘

当你读写一个文件时，Linux 会先把数据加载到**内存（RAM）**中，这个过程叫做 **Page Cache（页面缓存）**。

- **读文件**：先从磁盘读到内存 → 后续再读，直接从内存拿（更快）
- **写文件**：先写到内存的缓存中 → 稍后统一写回磁盘

### 一旦你“写”了数据，对应的内存页就“变脏”了

假设你执行了：

```
write(fd, "hello", 5);
```

操作系统会把这 5 个字节写入内存中的“页面缓存”（Page Cache），但**此时磁盘上的文件还没更新。**

👉 这个内存页就被标记为 **“脏页”（dirty page）** —— 因为它和磁盘上的数据不一致，是“脏”的。

---

### 举个生活中的比喻

想象你在用 **铅笔在草稿纸上写字**：

- 草稿纸 = 内存（Page Cache）
- 最终要抄到正式本上 = 磁盘文件
- 你写下的内容 = 脏页
- 抄到正式本 = 脏页写回磁盘

你可以在草稿纸上随便改（快），但最终必须抄一遍到正式本上，才算真正保存。

---

### 脏页什么时候写回磁盘？

Linux 不会每写一次就立刻刷盘（那样太慢），而是根据以下条件**异步写回**：

1. 脏页占内存比例超过 `dirty_background_ratio`（如 10%）→ 后台悄悄写
2. 脏页超过 `dirty_ratio`（如 20%）→ 所有写操作被阻塞，强制写回
3. 脏页在内存中存在超过 `dirty_expire_centisecs`（如 30秒）→ 必须写回
4. 手动执行 `sync` 命令 → 强制所有脏页落盘

---

### 为什么需要脏页？

| 好处 | 说明 |
| --- | --- |
| ✅ 提高写入性能 | 写内存比写磁盘快千倍，应用不卡顿 |
| ✅ 减少磁盘 I/O | 多次小写合并成一次大写，提升效率 |
| ✅ 支持缓存读取 | 读过的文件还在内存，下次读更快 |

---

### 脏页的风险

- **断电丢失数据**：如果系统突然断电，脏页中的数据就没了（没写到磁盘）
- **内存压力**：脏页太多会占用内存，影响其他程序
- **写卡顿**：当脏页达到 `dirty_ratio`，所有写操作会被阻塞，导致程序“卡住”

---

### 如何查看脏页?

```
# 查看当前脏页占用的内存（单位：KB）
grep -E "Dirty|Writeback" /proc/meminfo

# 示例输出：
MemTotal:       16384000 kB
MemFree:         8000000 kB
Dirty:            200000 kB     ← 当前脏页占用 200MB
Writeback:         50000 kB     ← 正在写回磁盘的页
```

---

### 小结

| 术语 | 解释 |
| --- | --- |
| **页（Page）** | 内存管理的基本单位，通常 4KB |
| **Page Cache** | 文件数据在内存中的缓存 |
| **脏页（Dirty Page）** | 被修改过、但未写入磁盘的 Page Cache 页 |
| **写回（Writeback）** | 将脏页数据写入磁盘的过程 |

---

简单一句话：

> **脏页 = 内存中已修改、但还没保存到硬盘的数据块**

它是性能与数据安全之间的平衡机制。理解它，有助于优化系统 I/O 行为。

## 3.12.3 控制页面缓存写回行为的核心配置

`dirty_expire_centisecs`，`dirty_writeback_centisecs`，`dirty_ratio`，`dirty_background_ratio`是 Linux 内核中 **控制页面缓存（page cache）写回（writeback）行为** 的核心配置，它们共同决定了 **何时、如何、多快地将内存中的“脏页”（dirty pages）写入磁盘。**

### `/proc/sys/vm/dirty_background_ratio`（后台写回触发比例）

- **默认值**：10（即 10%）
- **单位**：占**系统总物理内存**的百分比
- **含义**：
  - 当系统中**脏页总量**超过这个比例时，**内核会自动启动一个后台线程（`pdflush` 或 `writeback` 线程）**，开始异步地将脏页写入磁盘。
  - 这是一个“软阈值”，目的是**提前清理脏页**，避免积压过多。
- ✅ **特点**：
  - 后台进行，不影响应用程序写操作。
  - 写入速度较慢、平滑。

> 📌 举例：如果系统有 16GB 内存，当脏页超过 1.6GB 时，后台开始写盘。

### `/proc/sys/vm/dirty_ratio`（强制写回触发比例）

- **默认值**：20（即 20%）
- **单位**：占**系统总物理内存**的百分比
- **含义**：
  - 当脏页总量超过这个比例时，**所有新的写操作（如 `write()`）会被阻塞**，直到脏页被写回磁盘，降到该阈值以下。
  - 这是一个“硬阈值”，防止内存被脏页耗尽。
- ⚠️ **特点**：
  - 会导致应用程序**写操作卡顿（stall）**，延迟显著增加。
  - 是一种“最后防线”机制。

> 📌 举例：16GB 内存下，脏页超过 3.2GB 时，**程序写文件会“卡住”**。

---

### 对比：`dirty_background_ratio` vs `dirty_ratio`

| 参数 | 触发时机 | 是否阻塞应用 | 目的 |
| --- | --- | --- | --- |
| `dirty_background_ratio` | 脏页 > 10% | ❌ 不阻塞 | 提前清理，平滑写入 |
| `dirty_ratio` | 脏页 > 20% | ✅ 阻塞所有写操作 | 防止内存耗尽 |

👉 **关系**：`dirty_background_ratio < dirty_ratio`，必须满足，否则后台机制失效。

---

### `/proc/sys/vm/dirty_expire_centisecs`（脏页过期时间）

- **默认值**：3000（即 30 秒）
- **单位**：厘秒（centiseconds），100 厘秒 = 1 秒
- **含义**：
  - 一个脏页从**被修改（变脏）到必须被写回磁盘的最长生命周期。**
  - 如果一个脏页在内存中存在时间超过这个值，即使没达到 `dirty_ratio`，也会被强制写回。
- ✅ **目的**：
  - 保证数据不会在内存中“太久”，提高数据落盘的及时性，降低意外断电时的数据丢失风险。

> 📌 举例：某个文件被写入内存后 30 秒内必须落盘。

---

### `/proc/sys/vm/dirty_writeback_centisecs`（写回线程唤醒间隔）

- **默认值**：500（即 5 秒）
- **单位**：厘秒
- **含义**：
  - 内核的 `writeback` 内核线程（如 `kworker/u:xx`）**每隔多久被唤醒一次**，去检查是否有脏页需要写

回。
    - 它决定了**后台写回的检查频率。**
- ⚠️ 注意：
    - 这个线程每次唤醒后，会检查是否满足 `dirty_background_ratio` 或 `dirty_expire_centisecs`，然后决定写多少数据。

> 📌 举例：每 5 秒检查一次"有没有脏页该写了"。

---

## 四者协同工作流程

1. 应用程序写文件 → 数据写入 page cache（变成脏页）
2. 脏页积累，达到 `dirty_background_ratio`（如 10%）→ **后台写回线程被唤醒**（每 `dirty_writeback_centisecs` 秒检查一次）
3. 后台线程开始将脏页写入磁盘
4. 如果写入速度慢，脏页继续增长，达到 `dirty_ratio`（如 20%）→ **所有新写操作被阻塞**
5. 同时，任何脏页在内存中存活超过 `dirty_expire_centisecs`（30秒）→ 无论比例，**必须写回**

---

## 调优建议

| 场景 | 建议配置 |
| --- | --- |
| **高吞吐写入（如日志服务器）** | ↑ `dirty_background_ratio` (15~30%)，↑ `dirty_ratio` (40~60%)，避免频繁阻塞 |
| **低延迟要求（如数据库）** | ↓ `dirty_expire_centisecs` (1000~1500)，让数据更快落盘 |
| **防止写卡顿** | ↓ `dirty_background_ratio` (5~8%)，提前触发后台写 |
| **SSD 环境** | 可适当提高比例，SSD 写性能高 |
| **机械硬盘** | 避免突发大量写，建议平滑写入，降低 `dirty_writeback_centisecs` 到 100（1秒）更频繁检查 |

---

## 查看当前脏页状态（另一种方式）

```
# 查看当前脏页统计
cat /proc/vmstat | grep -E "dirty|writeback"

# 输出示例:
nr_dirty 123456          # 当前脏页数量（页）
nr_writeback 789         # 正在写回的页数
```

结合 `free` 和页大小（通常 4KB），可计算脏页占用内存。

---

### 小结：一句话区分

| 参数 | 一句话解释 |
| --- | --- |
| `dirty_background_ratio` | “脏页超 10% 了，后台悄悄开始写” |
| `dirty_ratio` | “脏页超 20% 了，所有人等我写完！” |
| `dirty_expire_centisecs` | “脏页最多在内存待 30 秒” |
| `dirty_writeback_centisecs` | “每 5 秒醒来一次，看看有没有脏页要写” |

---

合理调整这些参数，可以显著提升系统的 I/O 平滑性和响应速度。

## 3.12.4 O_SYNC 能否保证磁盘里的缓存也刷新

**这里大致理解即可，不需要强记。**

> **通常可以，但不能 100% 保证，取决于磁盘是否“诚实”地实现 `SYNC` 语义。**

更准确地说：

- `O_SYNC` 或 `O_DSYNC` 标志会要求 **操作系统等待数据被“持久化”。**这包括：
  - **内存页缓存 → 磁盘控制器缓存 → 实际写入物理介质（如 NAND 或磁盘）。**
- 但如果**磁盘固件谎报完成**（例如声称已落盘，**实际还在自己的缓存中**），`O_SYNC` 也无法强制它真正落盘。

---

### 详细解释

**1. `O_SYNC` 的作用**

当你用 `open()` 打开文件时指定 `O_SYNC`：

```
int fd = open("file.txt", O_WRONLY | O_SYNC);
```

这意味着每次 `write()` 调用都会：

- 等待数据**同步写入底层存储设备**
- 包括：**文件数据 + 元数据（如 mtime）**

这与以下函数类似效果（但由内核自动完成）：

- `write()` + `fsync()` 或 `fdatasync()`

> ⚠️ 注意：`O_SYNC` 是“每次 write 都同步”，性能较低，适用于关键日志等场景。

---

**2. 数据路径上的缓存层级**

数据从用户程序到真正"落盘"要经过多个缓存：

```
用户缓冲区
    ↓
内核页缓存 (Page Cache) ← O_SYNC 会绕过/刷新此层
    ↓
块设备层
    ↓
磁盘控制器缓存（如 SATA/NVMe 缓存）
    ↓
磁盘内部缓存（DRAM 缓存，可能带电容保护）
    ↓
✅ 物理存储介质（HDD 磁道 / SSD NAND）
```

`O_SYNC` 的目标是确保数据走到最后一步——**写入物理介质。**

---

**3. `O_SYNC` 是否能刷新"磁盘里的缓存"？**

答案分两种情况：

| 情况 | 是否真正落盘 | 说明 |
| --- | --- | --- |
| ✅ **磁盘开启了写缓存但支持 FUA (Force Unit Access)** | 是 | `O_SYNC` 可触发 FUA 命令，要求绕过缓存，直接写入介质（常见于现代 SSD/HDD） |
| ❌ **磁盘谎报"已完成"** | 否 | 有些廉价磁盘或配置错误的 RAID 卡会"假装"写完了，实际上数据还在断电易失的 DRAM 缓存中 |
| ⚠️ **磁盘写缓存被禁用** | 是（但慢） | 此时所有写操作必须直接写入介质，`O_SYNC` 自然也落盘 |

> 📌 关键点：`O_SYNC` **依赖硬件正确实现同步语义。**如果硬件不守信，操作系统也无能为力。

## 3.12.5 保证数据"真正落盘"（持久化写入物理磁盘

**这里大致理解即可，不需要强记。**

要彻底避免"数据停留在缓存（RAID卡缓存、磁盘缓存、系统页缓存）而未写入盘片"的风险，需从**硬件层、文件系统层、应用层三维协同**，以下是经过细节修正的完整方案，含关键操作、作用及注意事项：

## 硬件层：筑牢缓存数据的“断电安全网”

硬件是数据落盘的基础，核心是解决“缓存断电丢失”问题，同时控制磁盘缓存的干扰：

1. **必选：使用带BBU/超级电容的RAID卡**
    - 作用：RAID卡缓存（提升IO性能的关键）在断电后，由BBU（电池备份单元）或超级电容供电（维持几秒至几分钟），强制将缓存内数据刷入物理磁盘，避免“RAID卡缓存数据丢失”。
    - 注意：仅保护RAID卡自身缓存，不覆盖系统页缓存、磁盘内置缓存；需定期通过RAID管理工具（如MegaCLI）检查BBU健康状态，防止电容失效。
2. **按需：禁用磁盘内置写缓存**
    - 作用：磁盘默认启用的写缓存（数据先存磁盘DRAM，再异步刷盘）是“伪落盘”重灾区，禁用后数据需直接写入盘片，彻底消除磁盘级缓存风险。
    - 操作：Linux用`hdparm -W 0 /dev/sda`（或`smartctl -s wcache,off /dev/sda`），Windows通过“磁盘属性-策略”关闭“启用设备写缓存”；
    - 代价：写入性能下降50%以上，仅适合金融交易、关键日志等“安全优先于性能”的场景。

## 文件系统层：强化数据与元数据的同步规则

选择支持持久性的文件系统，并配置关键参数，避免“文件系统级数据错乱”：

1. **选对文件系统**
    - 推荐：ext4（主流）、XFS（大文件场景）；
    - 禁用：ext2（无日志，崩溃易丢数据）、tmpfs（内存文件系统，断电即失）。
2. **配置强持久性参数**

| 文件系统 | 核心配置（挂载/格式化参数） | 作用 |
| --- | --- | --- |
| ext4 | `data=journal`（格式化时指定）+ `barrier=1`（挂载时指定） | `data=journal`：数据先写日志区，日志刷盘后再写数据区，持久性最强；`barrier=1`：防止磁盘重排序IO导致“元数据已写、数据丢失” |
| ext4 | （平衡方案）`data=ordered`（默认）+ `barrier=1` | 先写元数据、再按顺序写数据，性能优于`data=journal`，满足多数业务安全需求 |
| XFS | （默认已优化）`logbufs=8` + `logbsize=256k` | 强化元数据日志的刷盘效率，若需强数据持久性，需配合应用层`fsync` |

## 应用层：主动控制落盘时机，绕开冗余缓存

应用层通过IO标志或同步函数，直接定义“数据何时落盘”，减少中间层不确定性：

1. **强落盘组合：`O_DIRECT` + `O_SYNC`（Linux系统）**
    - `O_DIRECT`：绕开操作系统“页缓存”（数据直接在用户空间与磁盘IO控制器传输），避免“页缓存未刷盘”的风险；
    - `O_SYNC`：强制“每次写操作返回时，数据已物理写入盘片”（而非停留在任何缓存）；
    - 适用场景：需“写入即落盘”的关键场景（如数据库redo日志），代价是IO延迟显著增加。

2. **灵活落盘：调用 `fsync()` / `fdatasync()`**
   无需 `O_SYNC` 时，通过主动调用同步函数控制落盘时机，平衡安全与性能：
   - `fsync(int fd)`：同步指定文件（`fd`）的**所有数据+元数据**（如文件名、修改时间），确保文件完整一致；
   - `fdatasync(int fd)`：仅同步**数据**，不同步"非必要元数据"（如访问时间），性能比 `fsync` 略优；
   - 最佳实践：避免"每次写都调用"（性能骤降），采用"批量写入+定期调用"（如每写1MB或每100ms调用一次）。

### 不同场景的"落盘方案组合"

1. **极高安全场景（金融交易、医疗数据）**
   RAID卡BBU + 禁用磁盘写缓存 + ext4（`data=journal`） + 应用层 `O_DIRECT+O_SYNC`。
2. **平衡场景（普通业务数据、用户订单）**
   RAID卡BBU + ext4（`data=ordered+barrier=1`） + 应用层"批量写+ `fdatasync`（每50ms一次）"。
3. **性能优先、安全次场景（非关键日志、临时文件）**
   RAID卡BBU + 启用磁盘写缓存 + XFS（默认配置） + 应用层"批量写+ `fsync`（每1秒一次）"。

# 第04章 文件与目录操作

## 引言：为什么需要操作文件和目录?

在实际的Linux应用开发中，我们经常需要处理以下场景：

**场景1：日志系统**

```
# 应用程序需要按日期组织日志
logs/
├── 2025-10-10/
|   ├── error.log
|   └── access.log
├── 2025-10-11/
|   ├── error.log
|   └── access.log
```

程序需要自动创建日期目录、检查目录是否存在、写入日志文件。

**场景2：文件浏览器**

- 用户点击某个目录，程序需要列出该目录下的所有文件
- 显示文件类型（是文件还是目录）
- 显示文件大小、修改时间等信息

**场景3：备份工具**

- 遍历源目录下的所有文件和子目录
- 检查文件的修改时间，决定是否需要备份
- 在目标位置重建目录结构

**本章解决的问题：**

- ✅ 如何获取文件的类型、大小、修改时间等信息?
- ✅ 如何创建、删除、遍历目录?
- ✅ 软链接和硬链接有什么区别?如何创建?
- ✅ 如何删除文件和重命名文件?
- ✅ 如何递归遍历整个目录树?

**学完本章你能做什么:**

- 实现一个简单的`ls`命令
- 开发日志目录管理工具
- 编写文件备份脚本
- 实现目录树遍历功能

---

# 4.1 Linux文件类型

## 4.1.1 文件类型概述

在Windows系统中,通过文件扩展名来识别文件类型(`.txt`、`.exe`、`.zip`等)。但在Linux系统中,**文件类型由文件系统记录在inode中**,而不依赖于文件名。

Linux系统有**7种**文件类型:

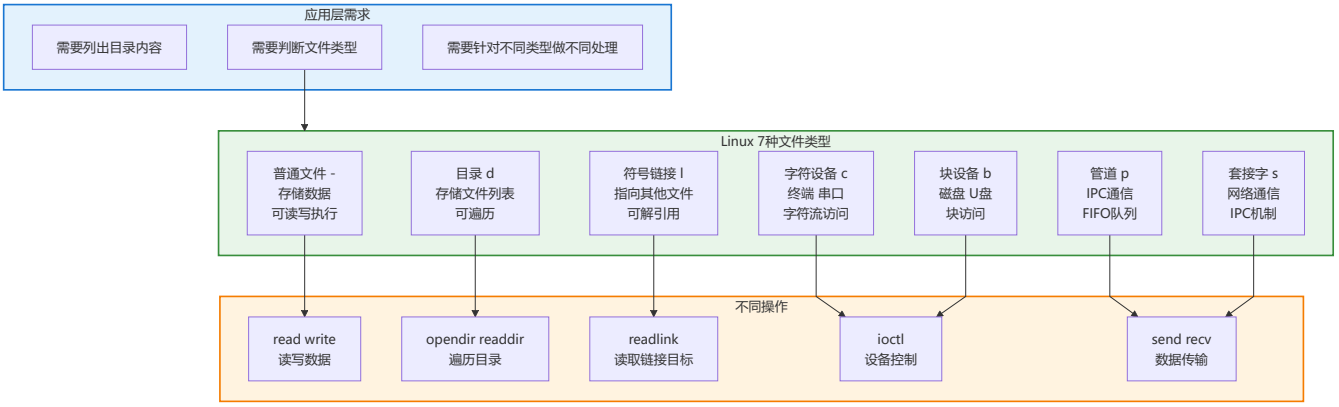
| 类型符号 | 类型名称 | 说明 | 举例 |
|---|---|---|---|
| `-` | 普通文件 | 文本文件、二进制文件等 | `test.txt`, `a.out` |
| `d` | 目录 | 文件夹,存储文件列表 | `/home`, `/etc` |
| `l` | 符号链接 | 类似Windows快捷方式 | 软链接文件 |
| `c` | 字符设备 | 按字符流访问的硬件设备 | `/dev/tty`, `/dev/null` |
| `b` | 块设备 | 按数据块访问的硬件设备 | `/dev/sda`, `/dev/loop0` |
| `p` | 管道 | 进程间通信的FIFO队列 | 命名管道 |
| `s` | 套接字 | 网络通信或本地IPC | Unix domain socket |

**查看文件类型:**

```
$ ls -l
-rw-r--r-- 1 ubuntu ubuntu  1024 Oct 10 10:30 test.txt      # 普通文件
drwxr-xr-x 2 ubuntu ubuntu  4096 Oct 10 10:31 my_dir        # 目录
lrwxrwxrwx 1 ubuntu ubuntu    10 Oct 10 10:32 link -> test.txt  # 符号链接
```

第一个字符就是文件类型标识。

### 4.1.2 文件类型与实际应用




**为什么需要判断文件类型?**

在编写工具时，需要根据文件类型采取不同的处理策略：

```
// 伪代码示例
if (是普通文件) {
    // 可以读取、复制、备份
    read_and_backup(file);
} else if (是目录) {
    // 需要递归遍历
    traverse_directory(file);
} else if (是符号链接) {
    // 需要读取链接目标
    readlink(file);
}
```



---

## 4.2 获取文件信息 - stat函数族

### 4.2.1 stat函数介绍

`stat()`是Linux提供的系统调用，用于获取文件的详细信息（元数据）。

**函数原型:**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

**参数说明:**

| 参数 | 类型 | 说明 |
| --- | --- | --- |
| pathname | const char* | 文件路径（绝对路径或相对路径） |
| fd | int | 文件描述符（fstat使用） |
| statbuf | struct stat* | 指向stat结构体的指针，用于存储文件信息 |

**返回值：**

- 成功：返回 `0`
- 失败：返回 `-1`，并设置 `errno`

**三个函数的区别：**

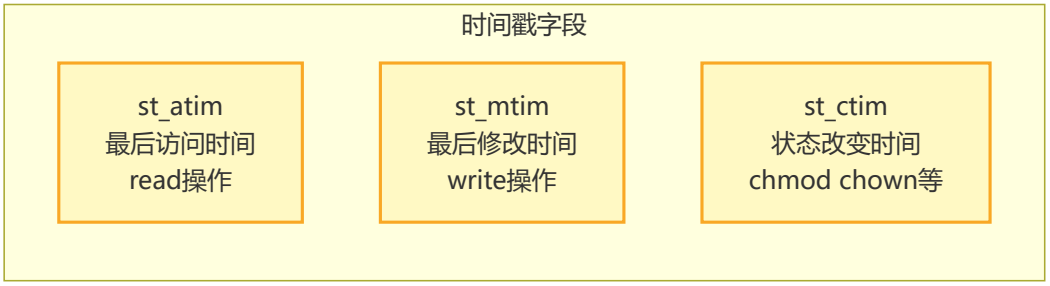
| 函数 | 区别 |
| --- | --- |
| stat | 如果路径是符号链接，返回**链接指向的文件**的信息 |
| lstat | 如果路径是符号链接，返回**符号链接本身**的信息 |
| fstat | 通过文件描述符获取信息（需要先用open打开文件） |

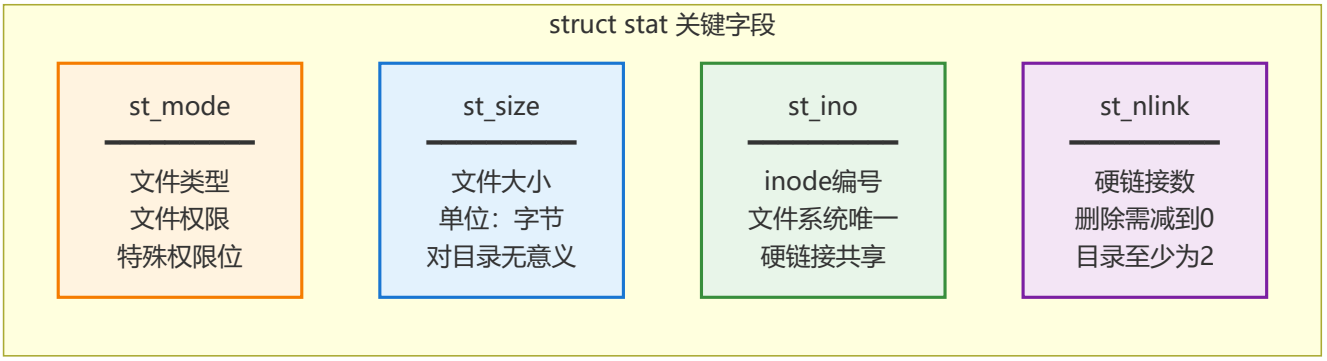
## 4.2.2 struct stat结构体详解

`struct stat`结构体包含了文件的所有元数据信息：

```
struct stat {
    dev_t     st_dev;         /* 文件所在设备的ID */
    ino_t     st_ino;         /* inode编号 */
    mode_t    st_mode;        /* 文件类型 + 权限 */
    nlink_t   st_nlink;       /* 硬链接数 */
    uid_t     st_uid;         /* 所有者用户ID */
    gid_t     st_gid;         /* 所有者组ID */
    dev_t     st_rdev;        /* 设备号（针对设备文件） */
    off_t     st_size;        /* 文件大小（字节） */
    blksize_t st_blksize;     /* 文件系统块大小 */
    blkcnt_t  st_blocks;      /* 占用的块数 */
    struct timespec st_atim;  /* 最后访问时间 */
    struct timespec st_mtim;  /* 最后修改时间 */
    struct timespec st_ctim;  /* 最后状态改变时间 */
};
```

**重要字段说明：**






## 4.2.3 判断文件类型

**st_mode字段的组成：**

`st_mode`是一个16位的整数，包含了文件类型和权限信息：

```
 15  14  13  12 | 11  10   9 | 8   7   6 | 5   4   3 | 2   1   0
-------------------------------------------------------------------
文件类型 (4位)  | S(特殊3位)  | U(所有者)  | G(同组)   | O(其他)
```

**判断文件类型的宏：**

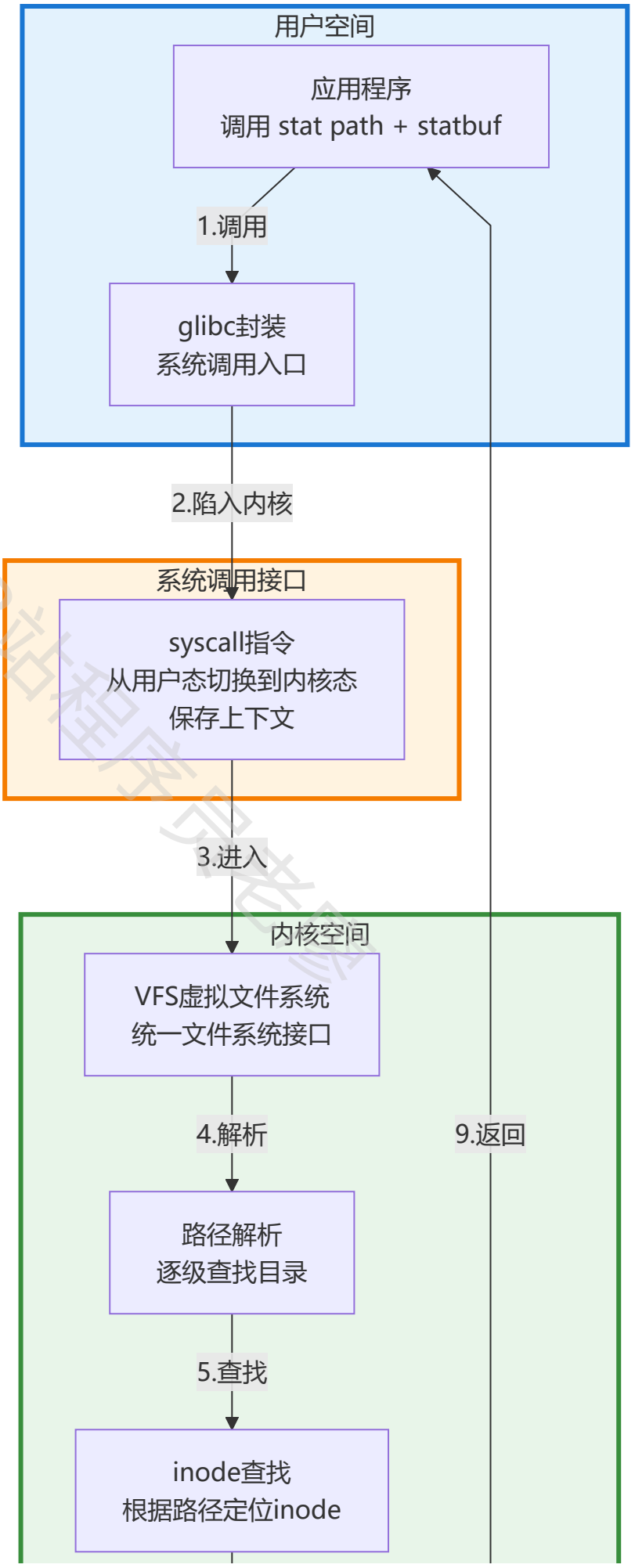
Linux提供了一组宏来判断文件类型：

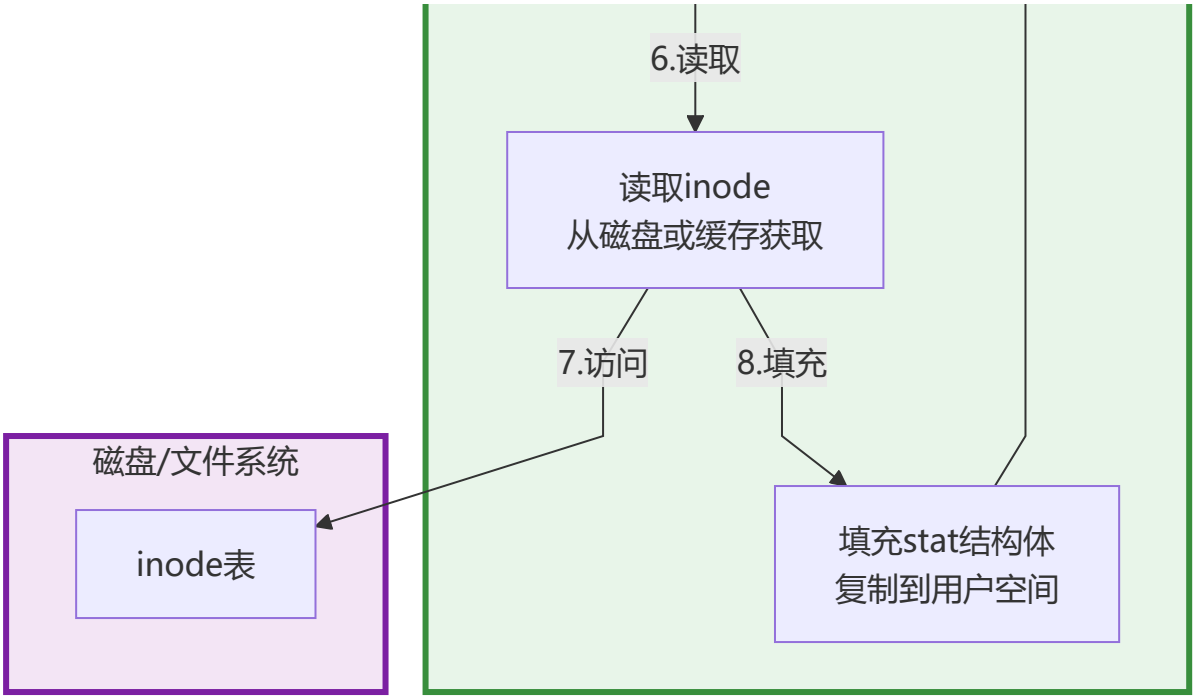
| 宏 | 功能 | 示例 |
| --- | --- | --- |
| S_ISREG(m) | 是否为普通文件 | `if (S_ISREG(st.st_mode))...` |
| S_ISDIR(m) | 是否为目录 | `if (S_ISDIR(st.st_mode))...` |
| S_ISLNK(m) | 是否为符号链接 | 需要使用lstat |
| S_ISCHR(m) | 是否为字符设备 | `/dev/tty`等 |
| S_ISBLK(m) | 是否为块设备 | `/dev/sda`等 |
| S_ISFIFO(m) | 是否为管道 | 命名管道 |
| S_ISSOCK(m) | 是否为套接字 | Unix域套接字 |

**使用示例：**

```c
struct stat st;

if (stat("test.txt", &st) == -1) {
    perror("stat");
    return -1;
}

// 判断文件类型
if (S_ISREG(st.st_mode)) {
    printf("这是一个普通文件\n");
    printf("文件大小: %ld 字节\n", st.st_size);
} else if (S_ISDIR(st.st_mode)) {
    printf("这是一个目录\n");
}
```

### 4.2.4 stat函数调用流程


用户空间
应用程序
调用 stat path + statbuf
1.调用
glibc封装
系统调用入口
2.陷入内核
系统调用接口
syscall指令
从用户态切换到内核态
保存上下文
3.进入
内核空间
VFS虚拟文件系统
统一文件系统接口
4.解析
9.返回
路径解析
逐级查找目录
5.查找
inode查找
根据路径定位inode




**流程说明：**

1. 应用程序调用 `stat()` 函数
2. glibc将调用转换为系统调用
3. CPU从用户态切换到内核态（Ring 3 → Ring 0）
4. 内核VFS层进行路径解析（如 `/home/user/test.txt`）
5. 根据路径逐级查找，最终定位到文件的inode
6. 从磁盘（或页缓存）读取inode信息
7. 将inode信息转换为 `struct stat` 格式
8. 复制到用户空间的statbuf
9. 返回用户程序



## 4.2.5 完整示例代码

💻 **实践代码：** `src/chapter04/file_info.c`

```
/*
 * 文件: file_info.c
 * 功能: 获取并显示文件的详细信息
 *
 * 演示内容:
 *   - stat()获取文件信息
 *   - 判断文件类型
 *   - 显示文件大小、inode、时间戳
 *
 * 编译: gcc -o file_info file_info.c
 * 运行: ./file_info <文件路径>
 */

#include <stdio.h>
```

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>
#include <string.h>

// 获取文件类型字符串
const char* get_file_type(mode_t mode) {
    if (S_ISREG(mode))  return "普通文件";
    if (S_ISDIR(mode))  return "目录";
    if (S_ISLNK(mode))  return "符号链接";
    if (S_ISCHR(mode))  return "字符设备";
    if (S_ISBLK(mode))  return "块设备";
    if (S_ISFIFO(mode)) return "管道";
    if (S_ISSOCK(mode)) return "套接字";
    return "未知类型";
}

// 格式化时间
void format_time(time_t t, char *buf, size_t size) {
    struct tm *tm_info = localtime(&t);
    strftime(buf, size, "%Y-%m-%d %H:%M:%S", tm_info);
}

int main(int argc, char *argv[]) {
    struct stat st;
    char time_buf[64];

    // 1. 检查参数
    if (argc != 2) {
        fprintf(stderr, "用法: %s <文件路径>\n", argv[0]);
        return 1;
    }

    // 2. 调用stat获取文件信息
    if (stat(argv[1], &st) == -1) {
        perror("stat失败");
        return 1;
    }

    // 3. 显示文件信息
    printf("----------------------------\n");
    printf("文件路径: %s\n", argv[1]);
    printf("----------------------------\n");

    // 文件类型
    printf("文件类型: %s\n", get_file_type(st.st_mode));

    // 文件大小（仅对普通文件有意义）
    if (S_ISREG(st.st_mode)) {
        printf("文件大小: %ld 字节\n", st.st_size);
    }
```

```c
    // inode编号
    printf("inode编号: %ld\n", st.st_ino);

    // 硬链接数
    printf("硬链接数: %ld\n", st.st_nlink);

    // 时间戳
    format_time(st.st_atim.tv_sec, time_buf, sizeof(time_buf));
    printf("最后访问时间: %s\n", time_buf);

    format_time(st.st_mtim.tv_sec, time_buf, sizeof(time_buf));
    printf("最后修改时间: %s\n", time_buf);

    format_time(st.st_ctim.tv_sec, time_buf, sizeof(time_buf));
    printf("状态改变时间: %s\n", time_buf);

    printf("————————————————————————\n");

    return 0;
}
```

**运行示例：**

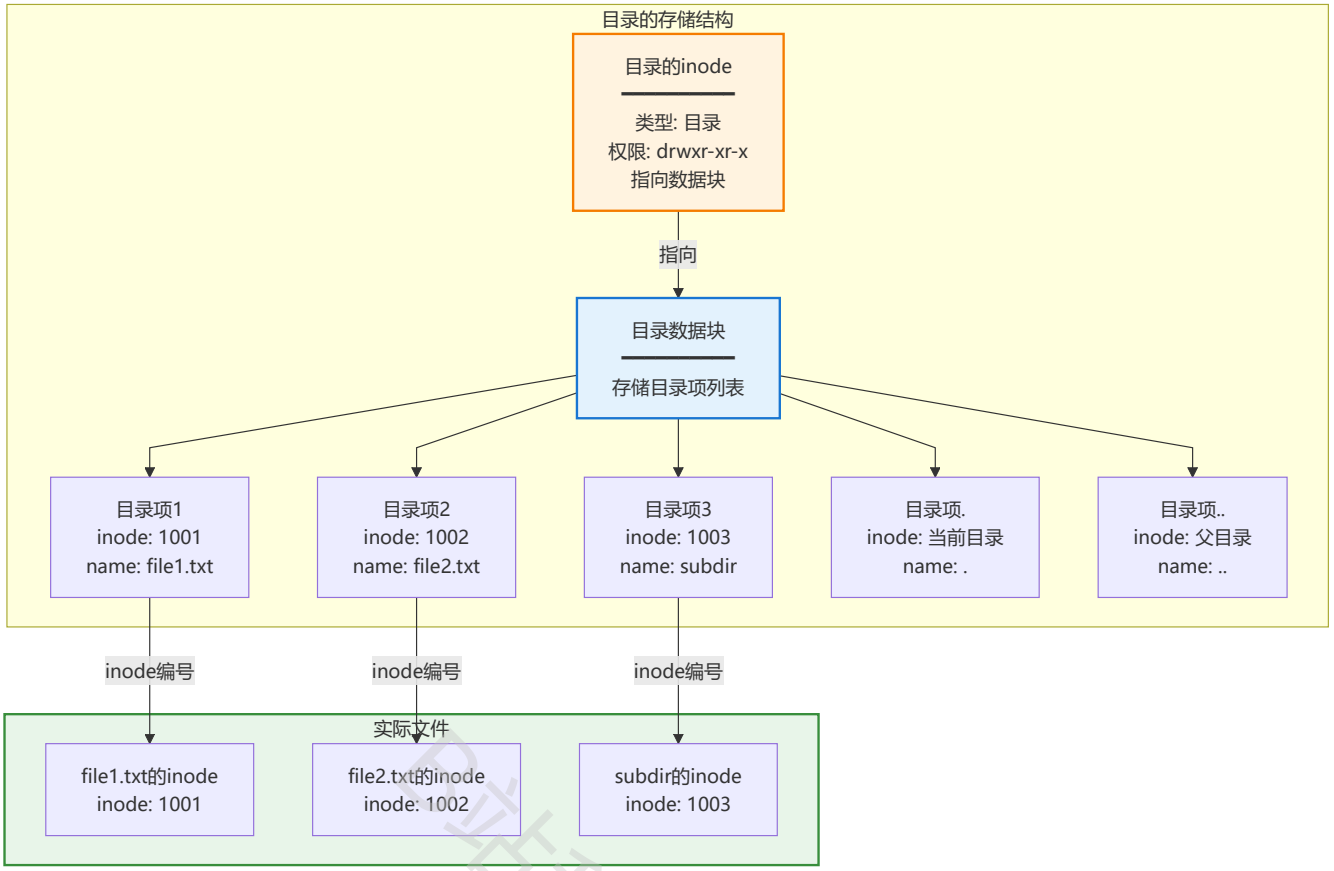
```
$ gcc -o file_info file_info.c
$ ./file_info /etc/passwd
————————————————————————
文件路径: /etc/passwd
————————————————————————
文件类型: 普通文件
文件大小: 2845 字节
inode编号: 1234567
硬链接数: 1
最后访问时间: 2025-10-10 10:30:15
最后修改时间: 2025-09-20 08:15:30
状态改变时间: 2025-09-20 08:15:30
————————————————————————
```



# 4.3 目录操作

## 4.3.1 目录的存储结构

在文件系统中，目录也是一种特殊的文件，它的数据块中存储的是**目录项列表**（directory entries）。




**关键概念：**

- 目录本身也有inode
- 目录的数据块存储的是目录项（文件名 + inode编号）
- 每个目录都有`.`（当前目录）和`..`（父目录）两个特殊目录项

## 4.3.2 创建和删除目录

### mkdir() - 创建目录

**函数原型：**

```
#include <sys/stat.h>
#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode);
```

**参数说明：**

| 参数 | 类型 | 说明 |
| --- | --- | --- |
| pathname | const char* | 要创建的目录路径 |
| mode | mode_t | 目录权限（受umask影响，实际权限为 mode & ~umask） |

**返回值：**

- 成功：返回 `0`
- 失败：返回 `-1`，并设置 `errno`

**常见错误：**

- `EEXIST`：目录已存在
- `ENOENT`：父目录不存在
- `EACCES`：权限不足

**rmdir() - 删除空目录**

**函数原型：**

```
#include <unistd.h>

int rmdir(const char *pathname);
```

**注意：**

- 只能删除**空目录**（只包含 `.` 和 `..`）
- 如果目录非空，返回 `ENOTEMPTY` 错误

💻 **实践代码：** `src/chapter04/dir_create_remove.c`

```
/*
 * 文件: dir_create_remove.c
 * 功能: 演示目录的创建和删除
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    const char *dir_path = "./test_dir";

    // 1. 创建目录（权限: rwxr-xr-x = 0755）
    printf("创建目录: %s\n", dir_path);
    if (mkdir(dir_path, 0755) == -1) {
        perror("mkdir失败");
        return 1;
    }
    printf("✓ 目录创建成功\n\n");

    // 2. 验证目录是否存在
    struct stat st;
    if (stat(dir_path, &st) == 0 && S_ISDIR(st.st_mode)) {
        printf("✓ 验证: 目录存在且类型正确\n\n");
    }
```

```
    // 3. 删除目录
    printf("删除目录: %s\n", dir_path);
    if (rmdir(dir_path) == -1) {
        perror("rmdir失败");
        return 1;
    }
    printf("✓ 目录删除成功\n");

    return 0;
}
```

## 4.3.3 打开和读取目录

**opendir() - 打开目录**

**函数原型：**

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
```

**返回值：**

- 成功：返回指向 `DIR` 结构体的指针（目录流）
- 失败：返回 `NULL`

**readdir() - 读取目录项**

**函数原型：**

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

**返回值：**

- 成功：返回指向 `struct dirent` 的指针
- 到达目录末尾或出错：返回 `NULL`（通过errno区分）

**struct dirent结构体：**

```
struct dirent {
    ino_t          d_ino;       /* inode编号 */
    off_t          d_off;       /* 偏移量（不常用） */
    unsigned short d_reclen;    /* 记录长度 */
    unsigned char  d_type;      /* 文件类型 */
    char           d_name[256]; /* 文件名 */
};
```

**d_type的值：**

| 宏定义 | 值 | 含义 |
| --- | --- | --- |
| DT_REG | 8 | 普通文件 |
| DT_DIR | 4 | 目录 |
| DT_LNK | 10 | 符号链接 |
| DT_FIFO | 1 | 管道 |
| DT_CHR | 2 | 字符设备 |
| DT_BLK | 6 | 块设备 |
| DT_SOCK | 12 | 套接字 |
| DT_UNKNOWN | 0 | 未知类型 |

**closedir() - 关闭目录**

```
int closedir(DIR *dirp);
```



## 4.3.4 遍历目录示例

💻 **实践代码：** `src/chapter04/list_dir.c`

```
/*
 * 文件: list_dir.c
 * 功能: 列出目录下的所有文件（类似ls命令）
 *
 * 编译: gcc -o list_dir list_dir.c
 * 运行: ./list_dir <目录路径>
 */

#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/types.h>
#include <errno.h>
#include <string.h>

// 获取文件类型字符
char get_type_char(unsigned char d_type) {
    switch(d_type) {
        case DT_REG:  return '-';
        case DT_DIR:  return 'd';
        case DT_LNK:  return 'l';
        case DT_CHR:  return 'c';
        case DT_BLK:  return 'b';
        case DT_FIFO: return 'p';
        case DT_SOCK: return 's';
        default:      return '?';
    }
```

```c
}

int main(int argc, char *argv[]) {
    DIR *dirp;
    struct dirent *entry;
    const char *dir_path;
    int count = 0;

    // 1. 参数检查
    if (argc < 2) {
        dir_path = ".";  // 默认当前目录
    } else {
        dir_path = argv[1];
    }

    // 2. 打开目录
    dirp = opendir(dir_path);
    if (dirp == NULL) {
        fprintf(stderr, "无法打开目录 %s: %s\n",
                dir_path, strerror(errno));
        return 1;
    }

    // 3. 读取并显示目录内容
    printf("目录内容: %s\n", dir_path);
    printf("-----------------------------------\n");
    printf("类型  inode编号       文件名\n");
    printf("-----------------------------------\n");

    errno = 0;
    while ((entry = readdir(dirp)) != NULL) {
        printf(" %c   %-12ld   %s\n",
               get_type_char(entry->d_type),
               entry->d_ino,
               entry->d_name);
        count++;
    }

    // 4. 检查是否出错
    if (errno != 0) {
        perror("readdir出错");
        closedir(dirp);
        return 1;
    }

    printf("-----------------------------------\n");
    printf("总计: %d 个文件/目录\n", count);

    // 5. 关闭目录
    closedir(dirp);

    return 0;
}
```

**运行示例：**

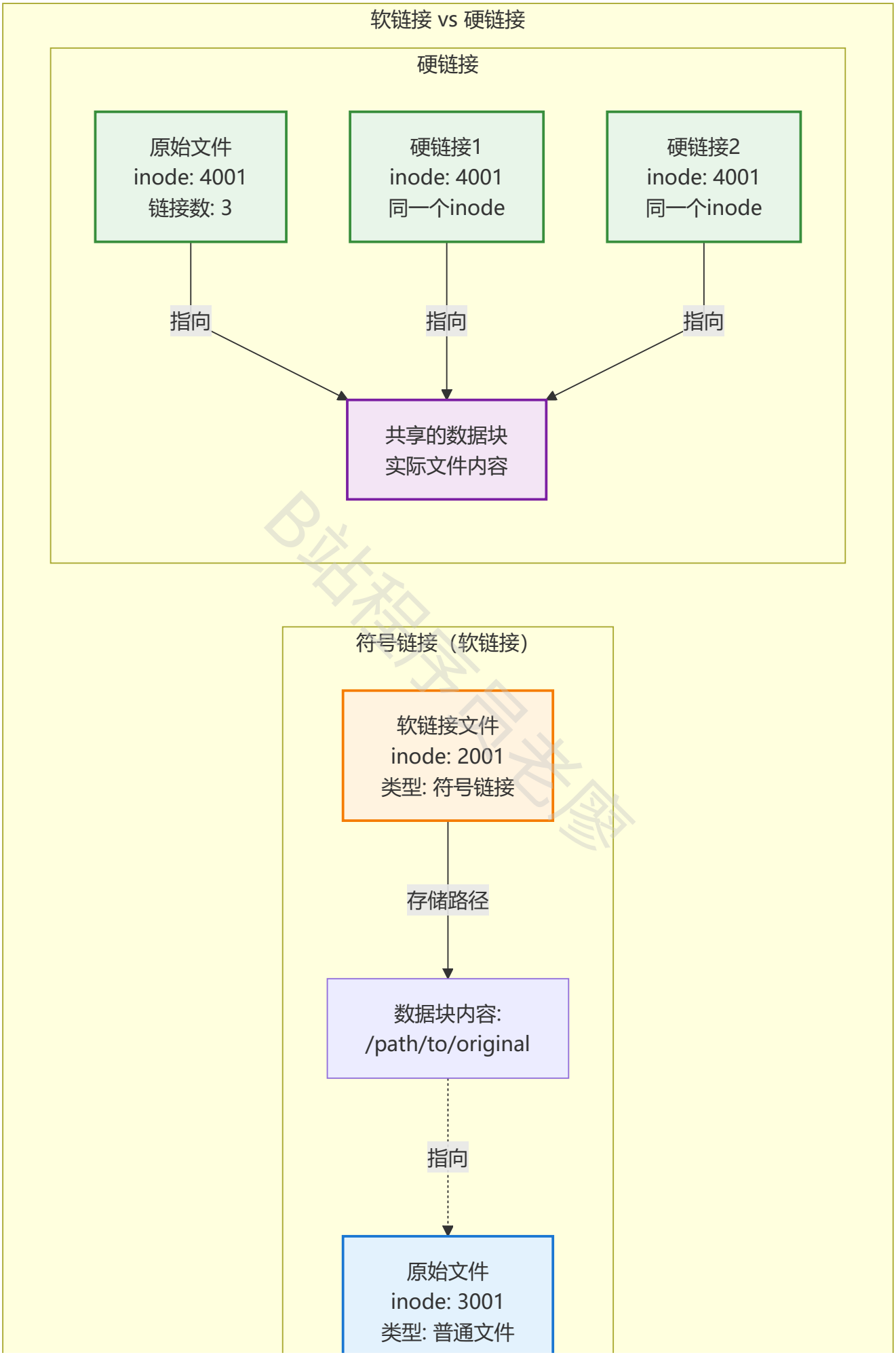
```
$ gcc -o list_dir list_dir.c
$ ./list_dir /tmp
目录内容: /tmp
--------------------------------------
类型   inode编号       文件名
--------------------------------------
 d   123456        .
 d   789012        ..
 -   345678        test.txt
 d   901234        my_dir
 l   567890        link_file
--------------------------------------
总计: 5 个文件/目录
```

---

# 4.4 符号链接与硬链接

## 4.4.1 两种链接的原理

在Linux中，有两种链接文件：**符号链接（软链接）**和**硬链接**。


软链接 vs 硬链接
硬链接
原始文件
inode: 4001
链接数: 3
硬链接1
inode: 4001
同一个inode
硬链接2
inode: 4001
同一个inode
指向
指向
指向
共享的数据块
实际文件内容
符号链接（软链接）
软链接文件
inode: 2001
类型: 符号链接
存储路径
数据块内容:
/path/to/original
指向
原始文件
inode: 3001
类型: 普通文件

**关键区别对比：**

| 特性 | 符号链接（软链接） | 硬链接 |
| --- | --- | --- |
| inode | 拥有自己的inode | 与源文件共享同一个inode |
| 存储内容 | 存储源文件的路径字符串 | 直接指向源文件的数据块 |
| 删除源文件 | 链接变成"悬空链接"，无法访问 | 其他硬链接仍可正常访问 |
| 跨文件系统 | ✅ 可以 | ❌ 不可以 |
| 链接目录 | ✅ 可以 | ❌ 不可以（普通用户） |
| 文件大小 | 链接文件大小 = 路径字符串长度 | 所有硬链接大小相同 |
| 修改内容 | 修改会影响源文件 | 修改任何一个链接都会影响其他的 |

**形象比喻：**

- **符号链接**：像是一张"便签"，上面写着"真正的文件在那里"
- **硬链接**：像是给同一个文件起了多个名字，每个名字都是"真身"

## 4.4.2 创建链接

**link() - 创建硬链接**

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
```

- `oldpath`：源文件路径
- `newpath`：硬链接文件路径
- 返回值：成功返回0，失败返回-1

**symlink() - 创建符号链接**

```
#include <unistd.h>

int symlink(const char *target, const char *linkpath);
```

- `target`：目标文件路径（可以不存在）
- `linkpath`：符号链接文件路径
- 返回值：成功返回0，失败返回-1

**readlink() - 读取符号链接**

```
#include <unistd.h>

ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
```

- `pathname`: 符号链接路径
- `buf`: 存储目标路径的缓冲区
- `bufsiz`: 缓冲区大小
- 返回值：成功返回读取的字节数，失败返回-1

**注意：** `readlink()`不会在buf末尾添加`\0`，需要手动添加。

## 4.4.3 链接操作示例

💻 **实践代码：** `src/chapter04/link_demo.c`

```
/*
 * 文件: link_demo.c
 * 功能: 演示硬链接和符号链接的创建与使用
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>

int main(void) {
    struct stat st;
    char buf[256];
    ssize_t len;

    // 1. 创建一个测试文件
    FILE *fp = fopen("original.txt", "w");
    if (fp == NULL) {
        perror("创建文件失败");
        return 1;
    }
    fprintf(fp, "这是原始文件的内容\n");
    fclose(fp);
    printf("✓ 创建原始文件: original.txt\n");

    // 2. 创建硬链接
    if (link("original.txt", "hardlink.txt") == -1) {
        perror("创建硬链接失败");
        return 1;
    }
    printf("✓ 创建硬链接: hardlink.txt\n");

    // 3. 创建符号链接
```

```
    if (symlink("original.txt", "softlink.txt") == -1) {
        perror("创建符号链接失败");
        return 1;
    }
    printf("✓ 创建符号链接: softlink.txt\n\n");

    // 4. 查看inode信息
    printf("──────────────────────────────────────\n");
    printf("文件信息对比:\n");
    printf("──────────────────────────────────────\n");

    stat("original.txt", &st);
    printf("original.txt  - inode: %ld, 链接数: %ld\n",
           st.st_ino, st.st_nlink);

    stat("hardlink.txt", &st);
    printf("hardlink.txt  - inode: %ld, 链接数: %ld\n",
           st.st_ino, st.st_nlink);

    lstat("softlink.txt", &st);  // 使用lstat获取链接本身信息
    printf("softlink.txt  - inode: %ld, 链接数: %ld (符号链接)\n",
           st.st_ino, st.st_nlink);

    // 5. 读取符号链接的目标
    memset(buf, 0, sizeof(buf));
    len = readlink("softlink.txt", buf, sizeof(buf) - 1);
    if (len != -1) {
        buf[len] = '\0';  // 手动添加结束符
        printf("\n符号链接指向: %s\n", buf);
    }

    printf("──────────────────────────────────────\n");

    // 6. 清理
    printf("\n提示: 可以手动删除测试文件\n");
    printf("rm original.txt hardlink.txt softlink.txt\n");

    return 0;
}
```

**运行效果:**

```
$ gcc -o link_demo link_demo.c
$ ./link_demo
✓ 创建原始文件: original.txt
✓ 创建硬链接: hardlink.txt
✓ 创建符号链接: softlink.txt

──────────────────────────────────────
文件信息对比:
──────────────────────────────────────
original.txt  - inode: 123456, 链接数: 2
hardlink.txt  - inode: 123456, 链接数: 2
```

```
softlink.txt  - inode: 789012, 链接数: 1 (符号链接)

符号链接指向: original.txt
————————————————————————————————

$ ls -li
123456 -rw-r--r-- 2 ubuntu ubuntu 27 Oct 10 10:30 original.txt
123456 -rw-r--r-- 2 ubuntu ubuntu 27 Oct 10 10:30 hardlink.txt
789012 lrwxrwxrwx 1 ubuntu ubuntu 12 Oct 10 10:30 softlink.txt -> original.txt
```

**观察要点：**

- `original.txt` 和 `hardlink.txt` 的inode相同，都是 `123456`
- 两者的链接数都是 `2`
- `softlink.txt` 有自己的inode `789012`
- 符号链接的大小是 `12` 字节（"original.txt"的长度）

---

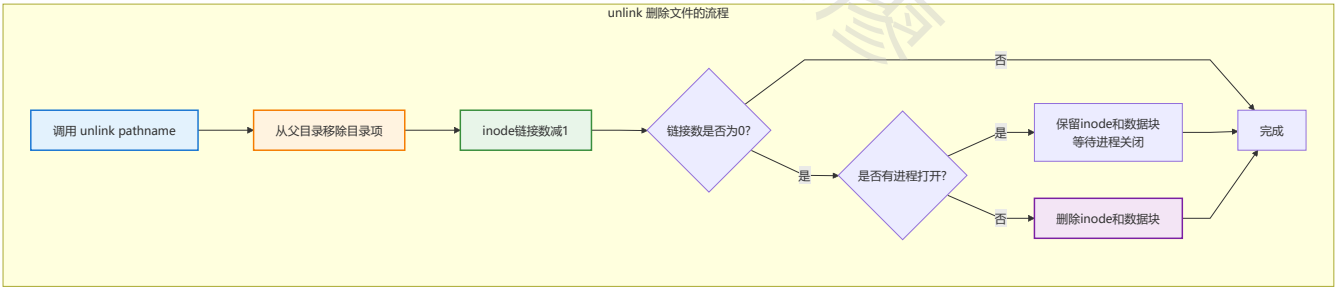
# 4.5 删除文件和重命名

## 4.5.1 删除文件 - unlink()

**函数原型：**

```c
#include <unistd.h>

int unlink(const char *pathname);
```

**工作原理：**




**关键点：**

1. `unlink()` 只是移除目录项，将链接数减1
2. 只有当链接数为0**且**没有进程打开该文件时，文件才真正删除
3. 如果文件被打开，删除会延迟到所有进程关闭该文件

**remove() - 更通用的删除函数**

```c
#include <stdio.h>

int remove(const char *pathname);
```

- 如果是文件，调用`unlink()`
- 如果是目录，调用`rmdir()`
- 更推荐使用（更通用）

## 4.5.2 重命名 - rename()

**函数原型：**

```
#include <stdio.h>

int rename(const char *oldpath, const char *newpath);
```

**功能：**

- 重命名文件或目录
- 移动文件到同一文件系统的另一个目录
- **原子操作**：不会出现中间状态

**特点：**

- 不改变inode编号
- 不移动文件数据
- 只修改目录项
- 如果`newpath`已存在，会被覆盖

💻 **实践代码：** `src/chapter04/file_ops.c`

```
/*
 * 文件: file_ops.c
 * 功能: 演示文件的删除和重命名操作
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>

int main(void) {
    struct stat st;
    FILE *fp;

    // 1. 创建测试文件
    printf("1. 创建测试文件...\n");
    fp = fopen("test.txt", "w");
    if (fp == NULL) {
        perror("创建文件失败");
        return 1;
    }
    fprintf(fp, "测试内容\n");
    fclose(fp);
```

```c
    // 获取inode
    stat("test.txt", &st);
    ino_t orig_ino = st.st_ino;
    printf("   文件: test.txt, inode: %ld\n\n", orig_ino);

    // 2. 重命名文件
    printf("2. 重命名 test.txt -> new_test.txt\n");
    if (rename("test.txt", "new_test.txt") == -1) {
        perror("重命名失败");
        return 1;
    }

    // 验证inode未变
    stat("new_test.txt", &st);
    printf("   文件: new_test.txt, inode: %ld\n", st.st_ino);
    if (st.st_ino == orig_ino) {
        printf("   ✓ inode未改变，只修改了文件名\n\n");
    }

    // 3. 删除文件
    printf("3. 删除文件 new_test.txt\n");
    if (unlink("new_test.txt") == -1) {
        perror("删除失败");
        return 1;
    }
    printf("   ✓ 文件已删除\n\n");

    // 4. 验证文件已不存在
    if (access("new_test.txt", F_OK) == -1) {
        printf("4. 验证：文件已不存在\n");
    }

    return 0;
}
```



# 4.6 实战案例：递归遍历目录树

## 4.6.1 需求分析

实现一个类似`tree`命令的程序，递归显示目录结构：

```
my_project/
├── src/
│   ├── main.c
│   └── util.c
├── include/
│   └── util.h
└── README.md
```

### 4.6.2 实现代码

💻 **实践代码：** `src/chapter04/tree.c`

```c
/*
 * 文件: tree.c
 * 功能: 递归遍历目录树（类似tree命令）
 *
 * 编译: gcc -o tree tree.c
 * 运行: ./tree <目录路径>
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#define MAX_PATH 1024

// 递归遍历目录
void traverse_directory(const char *dir_path, int depth) {
    DIR *dirp;
    struct dirent *entry;
    struct stat st;
    char full_path[MAX_PATH];

    // 打开目录
    dirp = opendir(dir_path);
    if (dirp == NULL) {
        perror("opendir");
        return;
    }

    // 读取目录项
    while ((entry = readdir(dirp)) != NULL) {
        // 跳过 . 和 ..
        if (strcmp(entry->d_name, ".") == 0 ||
            strcmp(entry->d_name, "..") == 0) {
            continue;
        }

        // 打印缩进
        for (int i = 0; i < depth; i++) {
            printf("|   ");
        }

        // 构造完整路径
        snprintf(full_path, sizeof(full_path), "%s/%s",
                 dir_path, entry->d_name);
```

```
        // 获取文件信息
        if (lstat(full_path, &st) == -1) {
            printf("├── %s (无法访问)\n", entry->d_name);
            continue;
        }

        // 根据文件类型打印
        if (S_ISDIR(st.st_mode)) {
            printf("├── %s/\n", entry->d_name);
            // 递归进入子目录
            traverse_directory(full_path, depth + 1);
        } else if (S_ISLNK(st.st_mode)) {
            char link_target[256];
            ssize_t len = readlink(full_path, link_target, sizeof(link_target) - 1);
            if (len != -1) {
                link_target[len] = '\0';
                printf("├── %s -> %s\n", entry->d_name, link_target);
            } else {
                printf("├── %s (符号链接)\n", entry->d_name);
            }
        } else {
            printf("├── %s (%ld bytes)\n", entry->d_name, st.st_size);
        }
    }

    closedir(dirp);
}

int main(int argc, char *argv[]) {
    const char *dir_path;

    // 获取目录路径
    if (argc < 2) {
        dir_path = ".";
    } else {
        dir_path = argv[1];
    }

    // 打印根目录
    printf("%s/\n", dir_path);

    // 递归遍历
    traverse_directory(dir_path, 0);

    return 0;
}
```



**运行示例：**

```
$ gcc -o tree tree.c
$ ./tree /home/ubuntu/project
/home/ubuntu/project/
|   ├── src/
|   |   ├── main.c (1024 bytes)
|   |   ├── util.c (2048 bytes)
|   ├── include/
|   |   ├── util.h (512 bytes)
|   ├── build/
|   |   ├── a.out (8192 bytes)
|   ├── README.md (256 bytes)
|   ├── link -> /tmp/test (符号链接)
```

---

# 4.7 本章总结

## 4.7.1 知识点回顾

本章学习了Linux文件和目录操作的核心知识：

**1. 文件类型（7种）**

- 普通文件、目录、符号链接、字符设备、块设备、管道、套接字
- 使用`stat()`和`S_IS*`宏判断文件类型

**2. 获取文件信息**

- `stat()`, `lstat()`, `fstat()`三个函数
- `struct stat`结构体：inode、大小、时间戳、链接数等

**3. 目录操作**

- `mkdir()`: 创建目录
- `rmdir()`: 删除空目录
- `opendir()`, `readdir()`, `closedir()`: 遍历目录
- `struct dirent`: 目录项信息

**4. 链接文件**

- **硬链接**：共享inode，不能跨文件系统
- **符号链接**：独立inode，存储路径字符串
- `link()`, `symlink()`, `readlink()`

**5. 文件操作**

- `unlink()`, `remove()`: 删除文件
- `rename()`: 重命名/移动文件

## 4.7.2 API快速参考

**文件信息获取：**

| 函数 | 功能 | 头文件 |
| --- | --- | --- |
| stat | 获取文件信息 | sys/stat.h |
| lstat | 获取链接本身信息 | sys/stat.h |
| fstat | 通过fd获取信息 | sys/stat.h |

**目录操作：**

| 函数 | 功能 | 头文件 |
| --- | --- | --- |
| mkdir | 创建目录 | sys/stat.h |
| rmdir | 删除空目录 | unistd.h |
| opendir | 打开目录 | dirent.h |
| readdir | 读取目录项 | dirent.h |
| closedir | 关闭目录 | dirent.h |

**链接操作：**

| 函数 | 功能 | 头文件 |
| --- | --- | --- |
| link | 创建硬链接 | unistd.h |
| symlink | 创建符号链接 | unistd.h |
| readlink | 读取符号链接 | unistd.h |

**文件操作：**

| 函数 | 功能 | 头文件 |
| --- | --- | --- |
| unlink | 删除文件 | unistd.h |
| remove | 删除文件/目录 | stdio.h |
| rename | 重命名/移动 | stdio.h |



## 4.7.3 常见错误与注意事项

**1. 忘记检查返回值**

```
// ❌ 错误
opendir("/some/path");

// ✅ 正确
DIR *dirp = opendir("/some/path");
if (dirp == NULL) {
    perror("opendir");
    return -1;
}
```

**2. 目录遍历时忘记跳过. 和..**

```
// ✅ 必须跳过
if (strcmp(entry->d_name, ".") == 0 ||
    strcmp(entry->d_name, "..") == 0) {
    continue;
}
```

**3. 使用stat而非lstat读取符号链接**

```
// 如果要获取符号链接本身的信息
lstat("link_file", &st);  // ✅ 正确
stat("link_file", &st);   // ❌ 会解引用
```

**4. readlink不添加结束符**

```
char buf[256];
ssize_t len = readlink("link", buf, sizeof(buf) - 1);
if (len != -1) {
    buf[len] = '\0';  // ✅ 必须手动添加
}
```



## 4.7.4 进阶学习建议

1. **实践项目**
    - 实现一个简单的文件管理器
    - 编写目录同步工具
    - 开发日志轮转脚本
2. **深入理解**
    - inode和目录项的关系
    - VFS虚拟文件系统层
    - 不同文件系统的实现差异
3. **性能优化**
    - 使用`fstatat()`避免路径拼接
    - 批量操作时的优化策略

- 大目录遍历的性能考虑

## 配套代码清单

本章提供了以下示例代码（位于 `src/chapter04/`）：

| 文件 | 功能说明 |
| --- | --- |
| file_info.c | 获取和显示文件详细信息 |
| dir_create_remove.c | 目录的创建和删除 |
| list_dir.c | 列出目录内容（类似ls） |
| link_demo.c | 硬链接和符号链接演示 |
| file_ops.c | 文件删除和重命名操作 |
| tree.c | 递归遍历目录树 |

**编译运行说明请参考：** `src/chapter04/README.md`



# 第05章 进程控制

## 引言：为什么需要进程控制?

在实际的Linux应用开发中，我们经常需要创建和管理多个进程来完成复杂的任务。

**场景1：Shell命令执行**

当你在终端输入命令时，Shell会创建一个新进程来执行：

```
$ ls -l
# Shell进程 fork → 子进程 → exec执行/bin/ls
```

**场景2：后台任务处理**

应用需要执行耗时的任务（如视频转码、数据备份）而不阻塞主程序：

```
主进程（用户界面）
└── 子进程（后台转码，完成后通知主进程）
```

**本章解决的问题：**

- ✅ 如何创建新进程？（fork）
- ✅ 如何在新进程中执行其他程序？（exec族）
- ✅ 如何等待子进程结束并获取退出状态？（wait/waitpid）
- ✅ 什么是僵尸进程和孤儿进程？如何避免？

- ✅ 父子进程如何协作完成任务?

**学完本章你能做什么:**

- 实现一个简单的Shell
- 开发多进程服务器
- 编写进程管理工具
- 理解Linux系统的进程模型

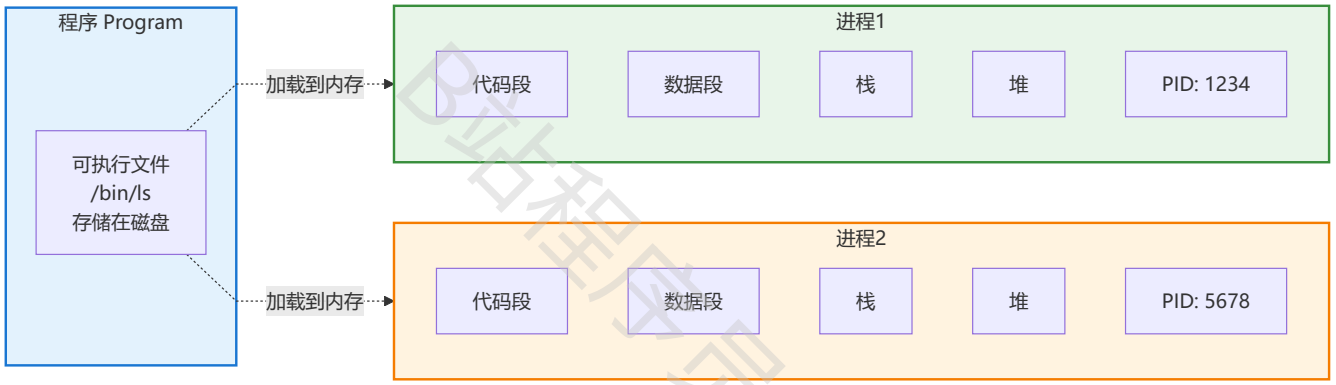
---

# 5.1 进程的基本概念

## 5.1.1 什么是进程?

**进程（Process）** 是正在运行的程序的实例。一个程序可以同时运行多个进程。

**程序 vs 进程:**





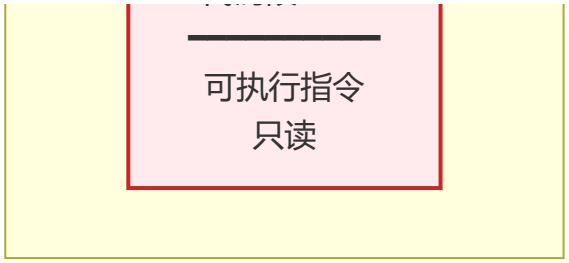

**关键概念:**

- **程序**：静态的可执行文件（存储在磁盘上）
- **进程**：动态的执行实例（加载到内存中运行）
- **PID**：进程ID，系统中每个进程的唯一标识符

## 5.1.2 进程的内存布局

Linux进程在内存中的典型布局:


进程内存空间 0 到 4GB
内核空间 3GB-4GB
操作系统内核
用户不可直接访问
栈 Stack
局部变量
函数调用栈
向下增长
堆 Heap
动态分配内存
malloc分配
向上增长
BSS段
未初始化的全局变量
自动初始化为0
数据段 Data
已初始化的全局变量
静态变量
代码段 Text




### 5.1.3 进程ID

Linux系统中，每个进程都有几个重要的ID：

| ID类型 | 函数 | 说明 |
| --- | --- | --- |
| PID | `getpid()` | 当前进程的进程ID |
| PPID | `getppid()` | 父进程的进程ID |
| UID | `getuid()` | 当前进程的用户ID |
| GID | `getgid()` | 当前进程的组ID |

**示例代码：**

```
#include <stdio.h>
#include <unistd.h>

int main(void) {
    printf("当前进程PID: %d\n", getpid());
    printf("父进程PPID: %d\n", getppid());
    printf("用户ID: %d\n", getuid());
    printf("组ID: %d\n", getgid());
    return 0;
}
```



## 5.2 创建进程 - fork()

### 5.2.1 fork() 函数介绍

`fork()`是Linux中创建新进程的系统调用。它会创建一个与父进程几乎完全相同的子进程。

**函数原型：**

```
#include <unistd.h>

pid_t fork(void);
```

**返回值（非常重要）：**

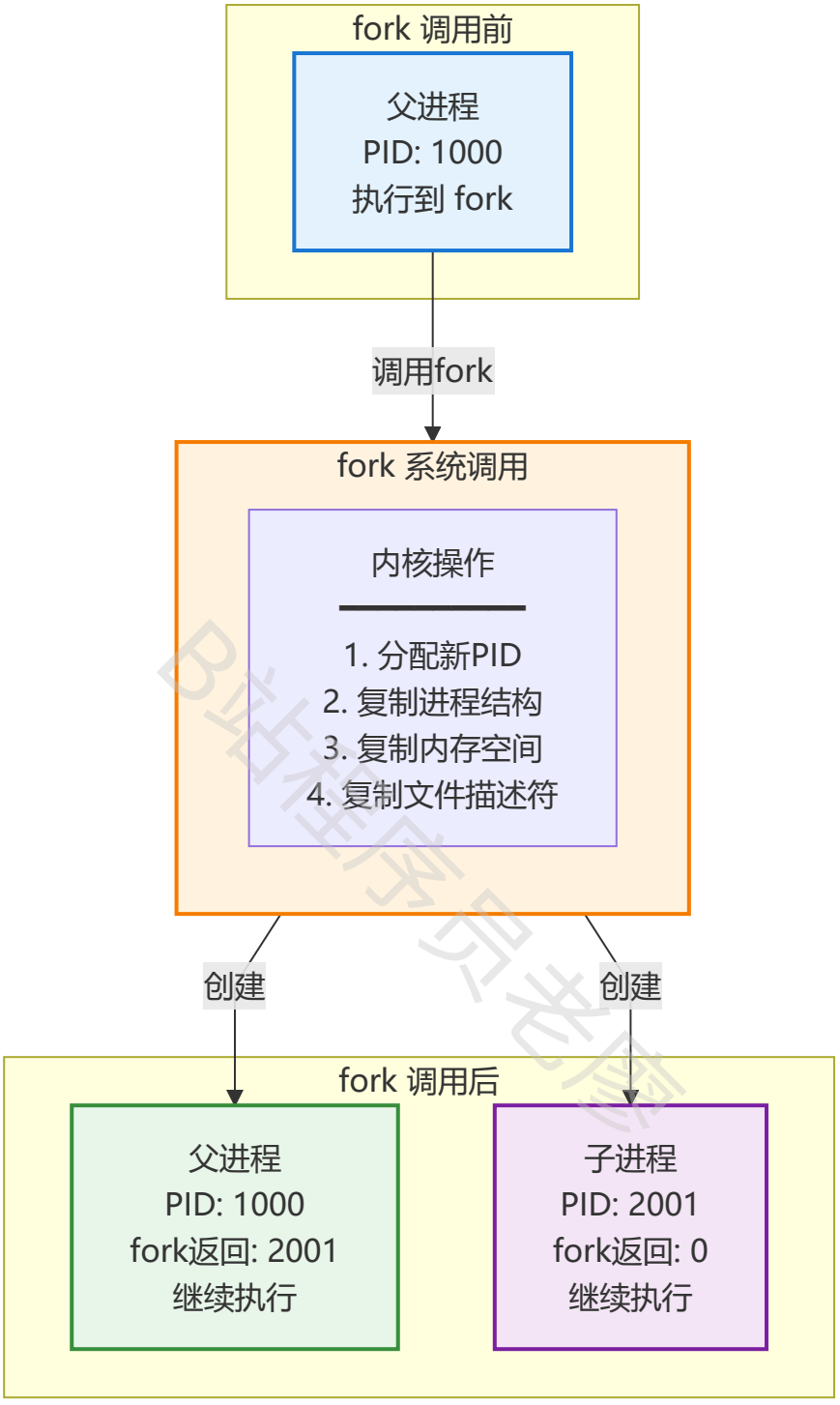
`fork()`函数有**三个**可能的返回值：

| 返回值 | 说明 |
| --- | --- |
| `> 0` | 在**父进程**中返回，值为子进程的PID |
| `0` | 在**子进程**中返回 |
| `-1` | 创建失败（如系统资源不足），设置`errno` |

**核心特点：**

> **"一次调用，两次返回"** —— fork()调用一次，但会返回两次（父进程和子进程各返回一次）

### 5.2.2 fork() 工作原理




**fork后的复制内容：**

✅ **会复制：**

- 代码段（共享，写时复制）
- 数据段、堆、栈（完全复制）
- 打开的文件描述符
- 当前工作目录
- 信号处理设置

- 环境变量

❌ **不会复制：**

- PID（子进程获得新的PID）
- 父进程ID（子进程的PPID是父进程的PID）
- 文件锁
- 挂起的信号

## 5.2.3 fork() 基本使用

💻 **实践代码：** `src/chapter05/fork_basic.c`

```c
/*
 * 文件: fork_basic.c
 * 功能: 演示fork()的基本使用
 *
 * 编译: gcc -o fork_basic fork_basic.c
 * 运行: ./fork_basic
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main(void) {
    pid_t pid;
    int count = 0;  // 测试变量

    printf("程序开始，当前进程PID: %d\n", getpid());
    printf("准备调用fork()...\n\n");

    // 创建子进程
    pid = fork();

    if (pid < 0) {
        // fork失败
        perror("fork失败");
        return 1;
    } else if (pid == 0) {
        // 子进程
        printf("【子进程】我是子进程\n");
        printf("【子进程】我的PID: %d\n", getpid());
        printf("【子进程】我的父进程PPID: %d\n", getppid());
        printf("【子进程】fork()返回值: %d\n", pid);

        // 修改count变量
        count = 100;
        printf("【子进程】count = %d\n", count);

        printf("【子进程】子进程结束\n");
```

```
    } else {
        // 父进程（pid > 0，pid是子进程的PID）
        printf("【父进程】我是父进程\n");
        printf("【父进程】我的PID: %d\n", getpid());
        printf("【父进程】fork()返回值（子进程PID）: %d\n", pid);

        // 等待子进程结束（简单sleep，后续会学wait）
        sleep(1);

        // count在父进程中不受影响
        printf("【父进程】count = %d\n", count);

        printf("【父进程】父进程结束\n");
    }

    // 父子进程都会执行这里
    printf("进程 %d 到达公共代码段\n", getpid());

    return 0;
}
```

**运行效果：**

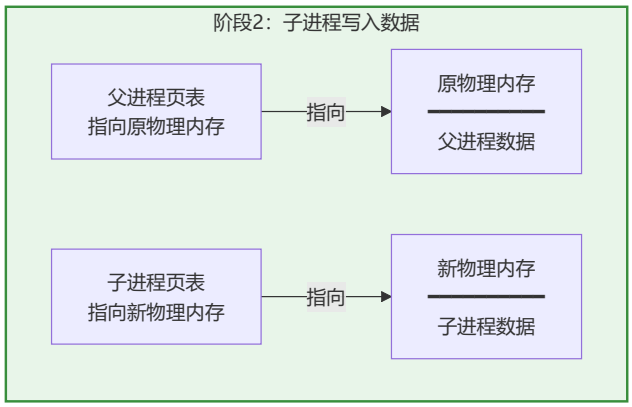
```
$ ./fork_basic
程序开始，当前进程PID: 12345
准备调用fork()...

【父进程】我是父进程
【父进程】我的PID: 12345
【父进程】fork()返回值（子进程PID）: 12346
【子进程】我是子进程
【子进程】我的PID: 12346
【子进程】我的父进程PPID: 12345
【子进程】fork()返回值: 0
【子进程】count = 100
【子进程】子进程结束
进程 12346 到达公共代码段
【父进程】count = 0
【父进程】父进程结束
进程 12345 到达公共代码段
```

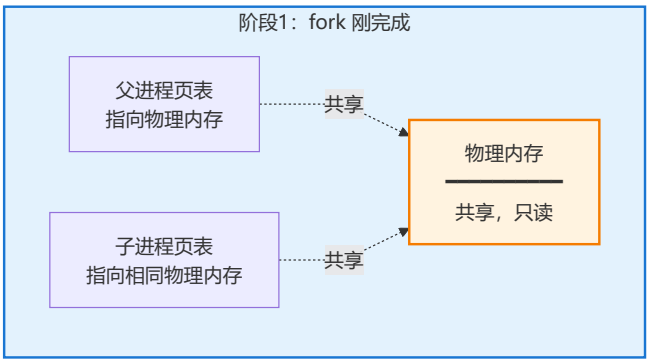
**关键观察：**

1. **fork()返回两次**：父进程中返回子进程PID，子进程中返回0
2. **变量独立**：子进程修改`count`不影响父进程
3. **执行顺序不确定**：父子进程谁先执行取决于调度器

### 5.2.4 写时复制（Copy-on-Write, COW）

为了提高性能，Linux使用**写时复制**技术：




**优势：**

- fork时只复制页表，不复制实际数据
- 只有在写入时才真正复制内存
- 如果子进程只读取数据，永远不需要复制

---

## 5.3 执行新程序 - exec族函数

### 5.3.1 为什么需要exec?

`fork()`创建的子进程是父进程的副本。如果我们想在子进程中运行**完全不同的程序**（如`ls`命令），就需要使用`exec`族函数。

**典型应用场景：Shell执行命令**

```
$ ls -l
# Shell进程：fork() → exec("/bin/ls", "ls", "-l")
```

### 5.3.2 exec族函数

Linux提供了一族exec函数，它们的功能相同，只是参数形式不同：

```
#include <unistd.h>

int execl(const char *path, const char *arg, ..., NULL);
int execlp(const char *file, const char *arg, ..., NULL);
int execle(const char *path, const char *arg, ..., NULL, char *const envp[]);

int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

**命名规则：**

| 后缀 | 含义 |
| --- | --- |
| `l` | list - 参数以列表形式传递（可变参数） |
| `v` | vector - 参数以数组形式传递 |
| `p` | path - 在PATH环境变量中搜索可执行文件 |
| `e` | environment - 可以指定环境变量 |

**常用的两个：**

1. **execl()** - 简单场景，参数少

```
execl("/bin/ls", "ls", "-l", NULL);
```

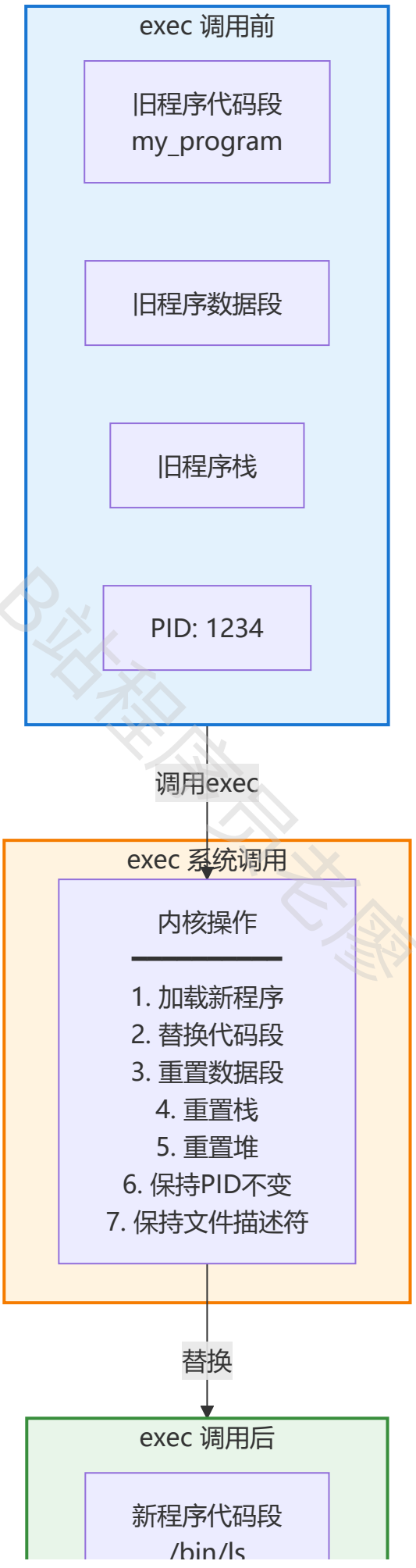
2. **execvp()** - 复杂场景，参数以数组形式，自动搜索PATH

```
char *args[] = {"ls", "-l", NULL};
execvp("ls", args);
```

**返回值：**

- **成功**：不返回（因为当前进程的代码已被替换）
- **失败**：返回`-1`，设置`errno`

### 5.3.3 exec() 工作原理


exec 调用前
旧程序代码段
my_program
旧程序数据段
旧程序栈
PID: 1234
调用exec
exec 系统调用
内核操作
1. 加载新程序
2. 替换代码段
3. 重置数据段
4. 重置栈
5. 重置堆
6. 保持PID不变
7. 保持文件描述符
替换
exec 调用后
新程序代码段
/bin/ls

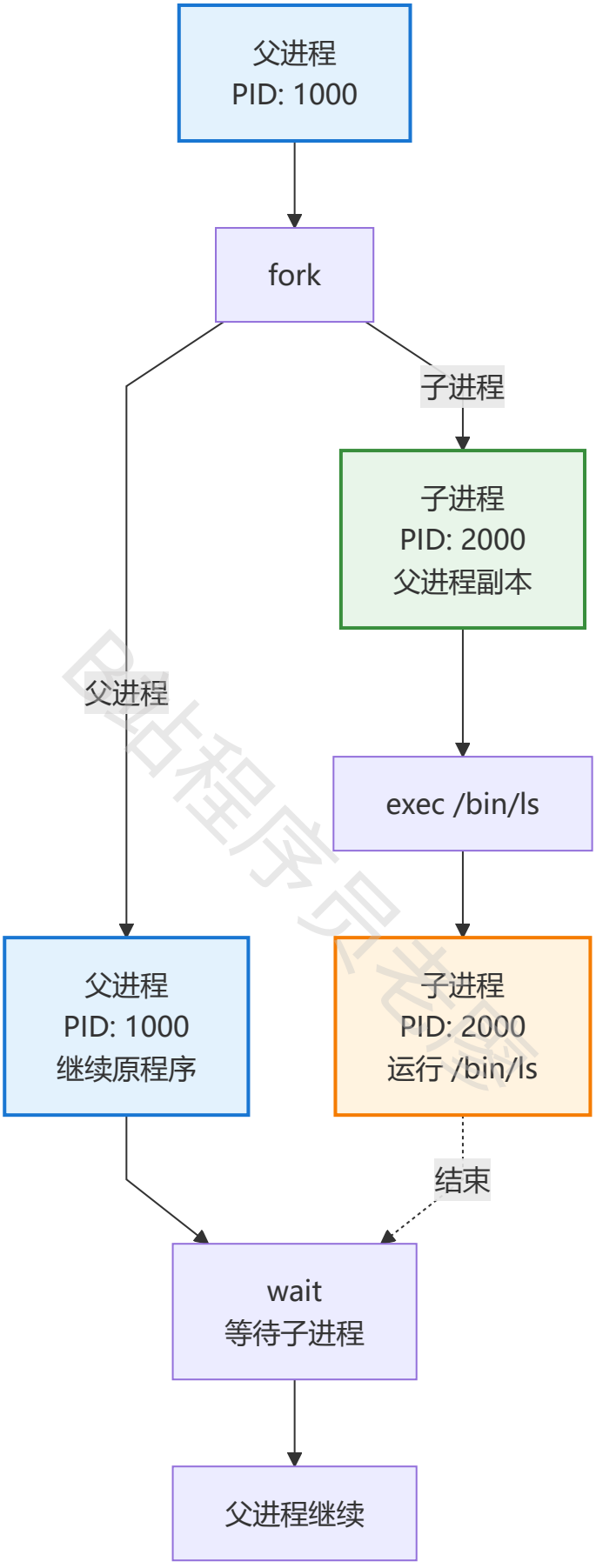



**关键点：**

- ✅ **PID不变**：进程ID保持不变
- ✅ **文件描述符保持**：已打开的文件仍然打开（除非设置了close-on-exec标志）
- ❌ **代码被替换**：旧程序的代码完全被新程序替换
- ❌ **内存被清空**：堆、栈、数据段都被重置

## 5.3.4 fork + exec 组合使用

实际应用中，通常是**fork + exec**的组合：




**💻 实践代码：** `src/chapter05/fork_exec.c`

```
/*
```

```c
 * 文件: fork_exec.c
 * 功能: 演示fork + exec组合使用
 *
 * 编译: gcc -o fork_exec fork_exec.c
 * 运行: ./fork_exec
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid;

    printf("父进程开始，PID: %d\n", getpid());

    // 创建子进程
    pid = fork();

    if (pid < 0) {
        perror("fork失败");
        return 1;
    } else if (pid == 0) {
        // 子进程：执行ls命令
        printf("\n【子进程】准备执行 ls -l 命令\n");
        printf("【子进程】PID: %d\n", getpid());
        printf("【子进程】即将被 /bin/ls 替换...\n\n");

        // 执行ls命令（下面的代码不会执行）
        execl("/bin/ls", "ls", "-l", NULL);

        // 如果exec失败，才会执行到这里
        perror("exec失败");
        return 1;
    } else {
        // 父进程：等待子进程结束
        printf("【父进程】子进程PID: %d\n", pid);
        printf("【父进程】等待子进程执行完毕...\n");

        int status;
        wait(&status);  // 等待子进程

        printf("\n【父进程】子进程已结束\n");
        printf("【父进程】父进程继续执行其他任务\n");
    }

    return 0;
}
```



**运行效果：**

```
$ ./fork_exec
父进程开始，PID: 12345

【子进程】准备执行 ls -l 命令
【子进程】PID: 12346
【子进程】即将被 /bin/ls 替换...

【父进程】子进程PID: 12346
【父进程】等待子进程执行完毕...

total 48
-rwxr-xr-x 1 ubuntu ubuntu 16704 Oct 11 10:30 fork_basic
-rw-r--r-- 1 ubuntu ubuntu  2048 Oct 11 10:30 fork_basic.c
-rwxr-xr-x 1 ubuntu ubuntu 16856 Oct 11 10:35 fork_exec
-rw-r--r-- 1 ubuntu ubuntu  1856 Oct 11 10:35 fork_exec.c

【父进程】子进程已结束
【父进程】父进程继续执行其他任务
```

---

# 5.4 等待子进程 - wait/waitpid

## 5.4.1 为什么需要wait?

当父进程创建子进程后，有两种选择：

1. **不关心子进程**：继续执行自己的任务
2. **等待子进程结束**：获取子进程的退出状态

如果父进程不等待子进程，可能导致**僵尸进程**（稍后讲解）。

## 5.4.2 wait() 函数

**函数原型：**

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

**功能：**

- 阻塞等待**任意一个**子进程结束
- 获取子进程的退出状态

**返回值：**

- 成功：返回结束的子进程PID
- 失败：返回`-1`（如没有子进程）

**status参数：**

`status`是一个输出参数，用于存储子进程的退出状态。需要使用宏来解析：

| 宏 | 功能 |
| --- | --- |
| `WIFEXITED(status)` | 子进程是否正常退出 |
| `WEXITSTATUS(status)` | 获取子进程的退出码（0-255） |
| `WIFSIGNALED(status)` | 子进程是否被信号终止 |
| `WTERMSIG(status)` | 获取终止子进程的信号编号 |

## 5.4.3 waitpid() 函数

`waitpid()` 比 `wait()` 更灵活，可以指定等待哪个子进程。

**函数原型：**

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

**参数说明：**

| 参数 | 说明 |
| --- | --- |
| pid | 要等待的子进程PID<br>`> 0`：等待指定PID<br>`-1`：等待任意子进程<br>`0`：等待同组的任意子进程 |
| status | 存储退出状态 |
| options | 选项标志<br>`0`：阻塞等待<br>`WNOHANG`：非阻塞，立即返回 |

**返回值：**

- `> 0`：返回结束的子进程PID
- `0`：使用`WNOHANG`时，子进程还未结束
- `-1`：出错

💻 **实践代码：** `src/chapter05/wait_demo.c`

```
/*
 * 文件: wait_demo.c
 * 功能: 演示wait和waitpid的使用
 *
 * 编译: gcc -o wait_demo wait_demo.c
 * 运行: ./wait_demo
 */
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid1, pid2;
    int status;

    printf("父进程PID: %d\n\n", getpid());

    // 创建第一个子进程
    pid1 = fork();
    if (pid1 == 0) {
        printf("【子进程1】PID: %d，睡眠2秒...\n", getpid());
        sleep(2);
        printf("【子进程1】准备退出，退出码: 10\n");
        exit(10);  // 退出码为10
    }

    // 创建第二个子进程
    pid2 = fork();
    if (pid2 == 0) {
        printf("【子进程2】PID: %d，睡眠1秒...\n", getpid());
        sleep(1);
        printf("【子进程2】准备退出，退出码: 20\n");
        exit(20);  // 退出码为20
    }

    // 父进程：等待子进程
    printf("【父进程】创建了两个子进程: %d 和 %d\n", pid1, pid2);
    printf("【父进程】开始等待子进程...\n\n");

    // 等待第一个结束的子进程
    pid_t ret = wait(&status);
    if (ret > 0) {
        printf("【父进程】子进程 %d 结束\n", ret);
        if (WIFEXITED(status)) {
            printf("【父进程】正常退出，退出码: %d\n\n", WEXITSTATUS(status));
        }
    }

    // 等待第二个子进程
    ret = wait(&status);
    if (ret > 0) {
        printf("【父进程】子进程 %d 结束\n", ret);
        if (WIFEXITED(status)) {
            printf("【父进程】正常退出，退出码: %d\n\n", WEXITSTATUS(status));
        }
    }
```

```
    printf("【父进程】所有子进程已结束，父进程退出\n");

    return 0;
}
```

**运行效果：**

```
$ ./wait_demo
父进程PID: 12345

【子进程1】PID: 12346，睡眠2秒...
【子进程2】PID: 12347，睡眠1秒...
【父进程】创建了两个子进程: 12346 和 12347
【父进程】开始等待子进程...

【子进程2】准备退出，退出码: 20
【父进程】子进程 12347 结束
【父进程】正常退出，退出码: 20

【子进程1】准备退出，退出码: 10
【父进程】子进程 12346 结束
【父进程】正常退出，退出码: 10

【父进程】所有子进程已结束，父进程退出
```

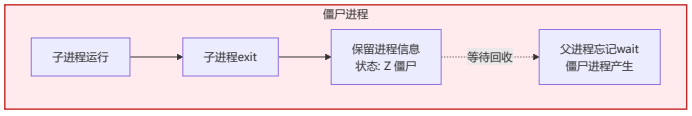


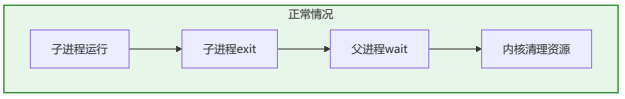
# 5.5 僵尸进程与孤儿进程

## 5.5.1 僵尸进程（Zombie Process）

### 什么是僵尸进程?

当子进程结束后，如果父进程没有调用 `wait/waitpid` 来回收子进程，子进程就会变成**僵尸进程。**




**僵尸进程的特点：**

- ✅ 子进程已经结束
- ❌ 但进程表项还在（保存退出状态）
- ❌ 不占用CPU和内存，但占用进程号
- ❌ 用 `ps` 命令看到状态为 `Z`

**查看僵尸进程：**

```
$ ps aux | grep Z
ubuntu   12346  0.0  0.0      0     0 ?        Z    10:30   0:00 [defunct]
```

**危害：**

- 占用进程号（系统进程数有限）
- 大量僵尸进程可能导致无法创建新进程

💻 **实践代码：** `src/chapter05/zombie_demo.c`

```c
/*
 * 文件: zombie_demo.c
 * 功能: 演示僵尸进程的产生
 *
 * 编译: gcc -o zombie_demo zombie_demo.c
 * 运行: ./zombie_demo
 *       在另一个终端执行: ps aux | grep zombie_demo
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main(void) {
    pid_t pid;

    printf("父进程PID: %d\n", getpid());

    pid = fork();

    if (pid < 0) {
        perror("fork失败");
        return 1;
    } else if (pid == 0) {
        // 子进程：运行5秒后退出
        printf("【子进程】PID: %d，5秒后退出\n", getpid());
        sleep(5);
        printf("【子进程】退出，将变成僵尸进程\n");
        exit(0);
    } else {
        // 父进程：不调用wait，继续运行
        printf("【父进程】子进程PID: %d\n", pid);
        printf("【父进程】我不调用wait，继续睡眠30秒\n");
        printf("【父进程】请在另一个终端执行: ps aux | grep zombie_demo\n");
        printf("【父进程】你会看到子进程状态为 Z (zombie)\n\n");

        sleep(30);

        printf("\n【父进程】睡眠结束，现在回收子进程\n");
        wait(NULL);
        printf("【父进程】子进程已回收，僵尸进程消失\n");
    }

    return 0;
}
```

**如何避免僵尸进程?**

1. **及时调用wait/waitpid**

```
while (wait(NULL) > 0);  // 回收所有子进程
```

2. **使用信号处理**（第6章会详细讲）

```
signal(SIGCHLD, SIG_IGN);  // 忽略子进程结束信号
```

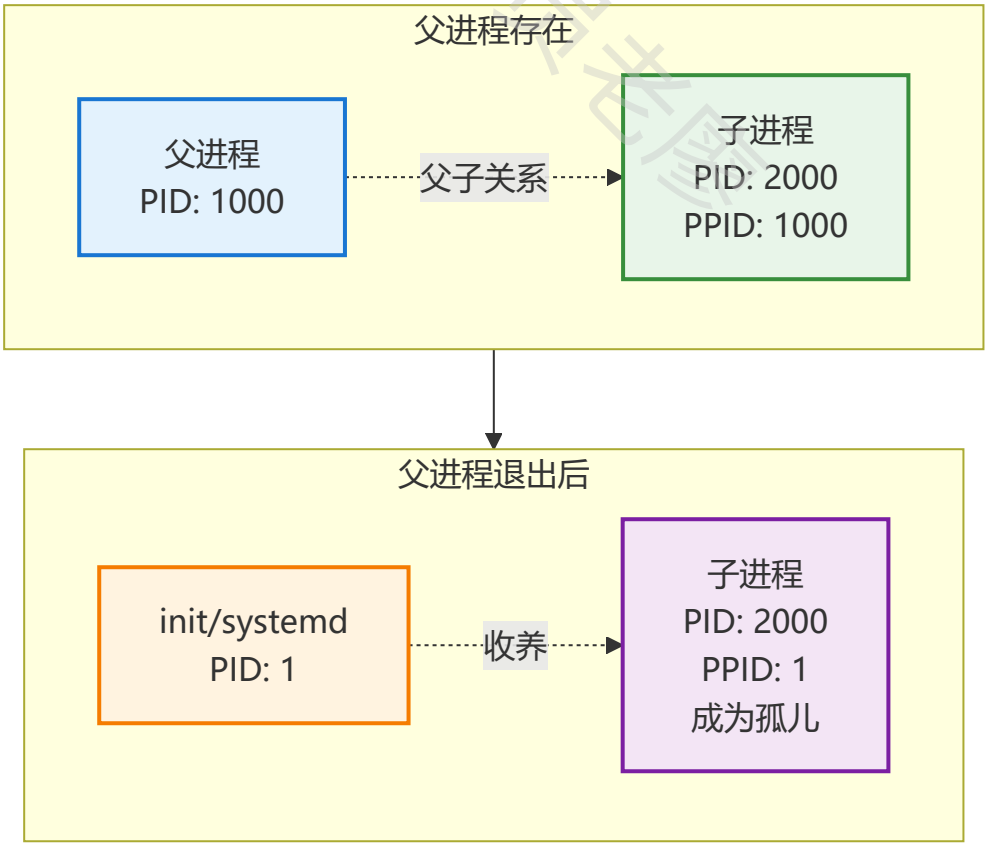
3. **两次fork**

```
if (fork() == 0) {
    if (fork() == 0) {
        // 孙子进程做实际工作
    }
    exit(0);  // 子进程立即退出
}
wait(NULL);  // 回收子进程
// 孙子进程成为孤儿，由init回收
```

## 5.5.2 孤儿进程（Orphan Process）

**什么是孤儿进程?**

如果父进程先于子进程结束，子进程就会变成**孤儿进程。**




**孤儿进程的特点:**

- ✅ 父进程提前退出
- ✅ 子进程被`init`进程（PID=1）收养
- ✅ 子进程正常运行，不会成为僵尸
- ✅ `init`会自动回收孤儿进程

💻 **实践代码：** `src/chapter05/orphan_demo.c`

```c
/*
 * 文件: orphan_demo.c
 * 功能: 演示孤儿进程
 *
 * 编译: gcc -o orphan_demo orphan_demo.c
 * 运行: ./orphan_demo
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main(void) {
    pid_t pid;

    printf("父进程PID: %d\n", getpid());

    pid = fork();

    if (pid < 0) {
        perror("fork失败");
        return 1;
    } else if (pid == 0) {
        // 子进程：先睡眠，让父进程先退出
        sleep(2);

        printf("\n【子进程】睡眠结束\n");
        printf("【子进程】我的PID: %d\n", getpid());
        printf("【子进程】我的PPID: %d\n", getppid());
        printf("【子进程】注意：PPID不再是原父进程，而是init(1)\n");
        printf("【子进程】我已经是孤儿进程了\n");

        sleep(3);
        printf("【子进程】退出\n");
    } else {
        // 父进程：创建子进程后立即退出
        printf("【父进程】子进程PID: %d\n", pid);
        printf("【父进程】我马上退出，子进程将成为孤儿\n");
        sleep(1);
        printf("【父进程】退出\n");
    }

    return 0;
```

```
}
```

**运行效果：**

```
$ ./orphan_demo
父进程PID: 12345
【父进程】子进程PID: 12346
【父进程】我马上退出，子进程将成为孤儿
【父进程】退出

$ # 父进程已退出，回到Shell提示符

【子进程】睡眠结束
【子进程】我的PID: 12346
【子进程】我的PPID: 1
【子进程】注意：PPID不再是原父进程，而是init(1)
【子进程】我已经是孤儿进程了
【子进程】退出
```

**孤儿进程 vs 僵尸进程：**

| 特性 | 孤儿进程 | 僵尸进程 |
| --- | --- | --- |
| 定义 | 父进程先退出，子进程还在运行 | 子进程退出，父进程未回收 |
| 父进程 | 变为 `init` (PID=1) | 原父进程 |
| 状态 | 正常运行 | 已结束，状态为 `Z` |
| 资源占用 | 正常占用CPU和内存 | 不占CPU/内存，但占进程号 |
| 危害 | ✅ 无危害，`init`会自动回收 | ❌ 大量僵尸会耗尽进程号 |



# 5.6 实战案例：多进程并发处理

## 5.6.1 需求分析

模拟一个简单的任务调度场景：父进程创建多个子进程，每个子进程处理一个任务，父进程等待所有子进程完成。

💻 **实践代码：** `src/chapter05/multi_process.c`

```
/*
 * 文件: multi_process.c
 * 功能: 多进程并发处理任务
 *
 * 编译: gcc -o multi_process multi_process.c
 * 运行: ./multi_process
 */

#include <stdio.h>
#include <stdlib.h>
```

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>

#define NUM_TASKS 5

// 模拟任务处理
void process_task(int task_id) {
    printf("【子进程 %d】开始处理任务 %d\n", getpid(), task_id);

    // 随机处理时间 1-3秒
    int time_needed = 1 + (rand() % 3);
    sleep(time_needed);

    printf("【子进程 %d】任务 %d 完成（耗时 %d秒）\n",
           getpid(), task_id, time_needed);
}

int main(void) {
    pid_t pids[NUM_TASKS];
    int i;

    srand(time(NULL));

    printf("————————————————————————————————\n");
    printf("【主进程】PID: %d\n", getpid());
    printf("【主进程】启动 %d 个子进程处理任务\n", NUM_TASKS);
    printf("————————————————————————————————\n\n");

    // 创建多个子进程
    for (i = 0; i < NUM_TASKS; i++) {
        pids[i] = fork();

        if (pids[i] < 0) {
            perror("fork失败");
            return 1;
        } else if (pids[i] == 0) {
            // 子进程
            process_task(i + 1);
            exit(i + 1);  // 以任务ID作为退出码
        }
        // 父进程继续循环创建下一个子进程
    }

    // 父进程：等待所有子进程结束
    printf("【主进程】等待所有子进程完成...\n\n");

    int status;
    pid_t finished_pid;
    int finished_count = 0;

    while ((finished_pid = wait(&status)) > 0) {
```

```
        finished_count++;

        if (WIFEXITED(status)) {
            int task_id = WEXITSTATUS(status);
            printf("【主进程】子进程 %d 完成（任务 %d）[%d/%d]\n",
                   finished_pid, task_id, finished_count, NUM_TASKS);
        }
    }

    printf("\n──────────────────────────────────\n");
    printf("【主进程】所有任务完成！\n");
    printf("──────────────────────────────────\n");

    return 0;
}
```

**运行效果：**

```
$ ./multi_process
──────────────────────────────────
【主进程】PID: 12345
【主进程】启动 5 个子进程处理任务
──────────────────────────────────

【子进程 12346】开始处理任务 1
【子进程 12347】开始处理任务 2
【子进程 12348】开始处理任务 3
【子进程 12349】开始处理任务 4
【子进程 12350】开始处理任务 5
【主进程】等待所有子进程完成...

【子进程 12347】任务 2 完成（耗时 1秒）
【主进程】子进程 12347 完成（任务 2）[1/5]
【子进程 12348】任务 3 完成（耗时 2秒）
【主进程】子进程 12348 完成（任务 3）[2/5]
【子进程 12346】任务 1 完成（耗时 3秒）
【主进程】子进程 12346 完成（任务 1）[3/5]
【子进程 12349】任务 4 完成（耗时 3秒）
【主进程】子进程 12349 完成（任务 4）[4/5]
【子进程 12350】任务 5 完成（耗时 3秒）
【主进程】子进程 12350 完成（任务 5）[5/5]

──────────────────────────────────
【主进程】所有任务完成！
──────────────────────────────────
```

# 5.7 本章总结

## 5.7.1 知识点回顾

**1. 进程基本概念**

- 进程是程序的运行实例
- 每个进程有唯一的PID
- 进程内存布局：代码段、数据段、堆、栈

**2. fork() - 创建进程**

- 一次调用，两次返回
- 父进程返回子进程PID，子进程返回0
- 写时复制（COW）技术提高效率

**3. exec族 - 执行新程序**

- 替换当前进程的代码段
- PID不变，文件描述符保留
- 常用：`execl()`、`execvp()`

**4. wait/waitpid - 等待子进程**

- `wait()`：等待任意子进程
- `waitpid()`：等待指定子进程，支持非阻塞
- 必须回收子进程，避免僵尸进程

**5. 僵尸进程与孤儿进程**

- **僵尸进程**：子进程结束，父进程未回收
- **孤儿进程**：父进程先退出，子进程被init收养



## 5.7.2 API快速参考

**进程信息：**

| 函数 | 功能 | 头文件 |
| --- | --- | --- |
| getpid | 获取当前进程PID | unistd.h |
| getppid | 获取父进程PID | unistd.h |
| getuid | 获取用户ID | unistd.h |
| getgid | 获取组ID | unistd.h |

**进程控制：**

| 函数 | 功能 | 头文件 |
| --- | --- | --- |
| fork | 创建子进程 | unistd.h |
| execl | 执行新程序（列表） | unistd.h |
| execvp | 执行新程序（数组） | unistd.h |
| exit | 终止进程 | stdlib.h |
| wait | 等待子进程 | sys/wait.h |
| waitpid | 等待指定子进程 | sys/wait.h |

**退出状态宏：**

| 宏 | 功能 |
| --- | --- |
| WIFEXITED | 是否正常退出 |
| WEXITSTATUS | 获取退出码 |
| WIFSIGNALED | 是否被信号终止 |
| WTERMSIG | 获取终止信号编号 |



## 5.7.3 常见错误与注意事项

**1. 忘记检查fork返回值**

```
// ❌ 错误
fork();
// 无法区分父子进程

// ✅ 正确
pid_t pid = fork();
if (pid < 0) {
    perror("fork");
} else if (pid == 0) {
    // 子进程
} else {
    // 父进程
}
```

**2. exec后的代码永远不会执行**

```
// ❌ 错误理解
execl("/bin/ls", "ls", NULL);
printf("这行代码不会执行\n");  // exec成功后不会返回

// ✅ 正确理解
execl("/bin/ls", "ls", NULL);
perror("exec");  // 只有exec失败才会执行到这里
return 1;
```

**3. 忘记回收子进程**

```
// ❌ 错误：产生僵尸进程
fork();

// ✅ 正确：及时回收
pid_t pid = fork();
if (pid > 0) {
    wait(NULL);  // 父进程等待子进程
}
```

**4. 错误使用exit vs _exit**

```
// fork后在子进程中：
exit(0);   // ✅ 正确：会刷新缓冲区
_exit(0);  // ⚠️ 特殊场景：不刷新缓冲区，直接退出
```



## 5.7.4 进阶学习建议

1. **实践项目**
   - 实现一个简单的Shell（支持管道）
   - 开发进程池管理器
   - 编写守护进程
2. **深入理解**
   - 进程调度算法
   - 进程上下文切换
   - 进程地址空间
3. **性能优化**
   - fork的开销分析
   - vfork vs fork
   - 进程 vs 线程的选择

---

# 配套代码清单

本章提供了以下示例代码（位于`src/chapter05/`）：

| 文件 | 功能说明 |
| --- | --- |
| fork_basic.c | fork基本使用 |
| fork_exec.c | fork + exec组合 |
| wait_demo.c | wait/waitpid使用 |
| zombie_demo.c | 僵尸进程演示 |
| orphan_demo.c | 孤儿进程演示 |
| multi_process.c | 多进程并发任务 |

**编译运行说明请参考：** `src/chapter05/README.md`

# 第06章 信号机制

## 引言：为什么需要信号?

在Linux系统编程中，进程之间需要一种异步通知机制来处理各种事件。

**场景1：优雅地关闭服务器**

当管理员执行 `kill -TERM <pid>` 时，服务器需要：

- 停止接受新请求
- 完成当前请求
- 关闭数据库连接
- 保存日志后退出

```
$ kill -TERM 1234  # 发送SIGTERM信号给进程1234
# 进程1234接收到信号后执行清理逻辑，然后退出
```

**场景2：处理子进程退出**

父进程需要知道子进程何时结束，避免产生僵尸进程：

```
// 父进程注册SIGCHLD信号处理函数
// 子进程退出时自动回收，无需阻塞等待
```

**场景3：处理用户中断**

用户按下 `Ctrl+C` 时，程序需要保存当前状态后退出：

```
$ ./long_running_task
正在处理数据... [按 Ctrl+C]
^C收到中断信号，正在保存进度...
进度已保存到 progress.dat
程序退出
```

**本章解决的问题：**

- ✅ 什么是信号？如何发送和接收信号？
- ✅ 如何自定义信号处理函数？
- ✅ 信号的可靠性和不可靠性？
- ✅ 如何正确处理信号中断的系统调用？
- ✅ 信号的阻塞和解除阻塞？

**学完本章你能做什么：**

- 实现优雅关闭的服务器程序
- 处理用户中断（Ctrl+C）
- 实现定时任务（SIGALRM）
- 正确处理子进程退出（SIGCHLD）

---

# 6.1 信号的基本概念

## 6.1.1 什么是信号？

**信号（Signal）** 是Linux进程间通信的一种机制，用于通知进程发生了某个事件。

**信号的特点：**



**关键概念：**

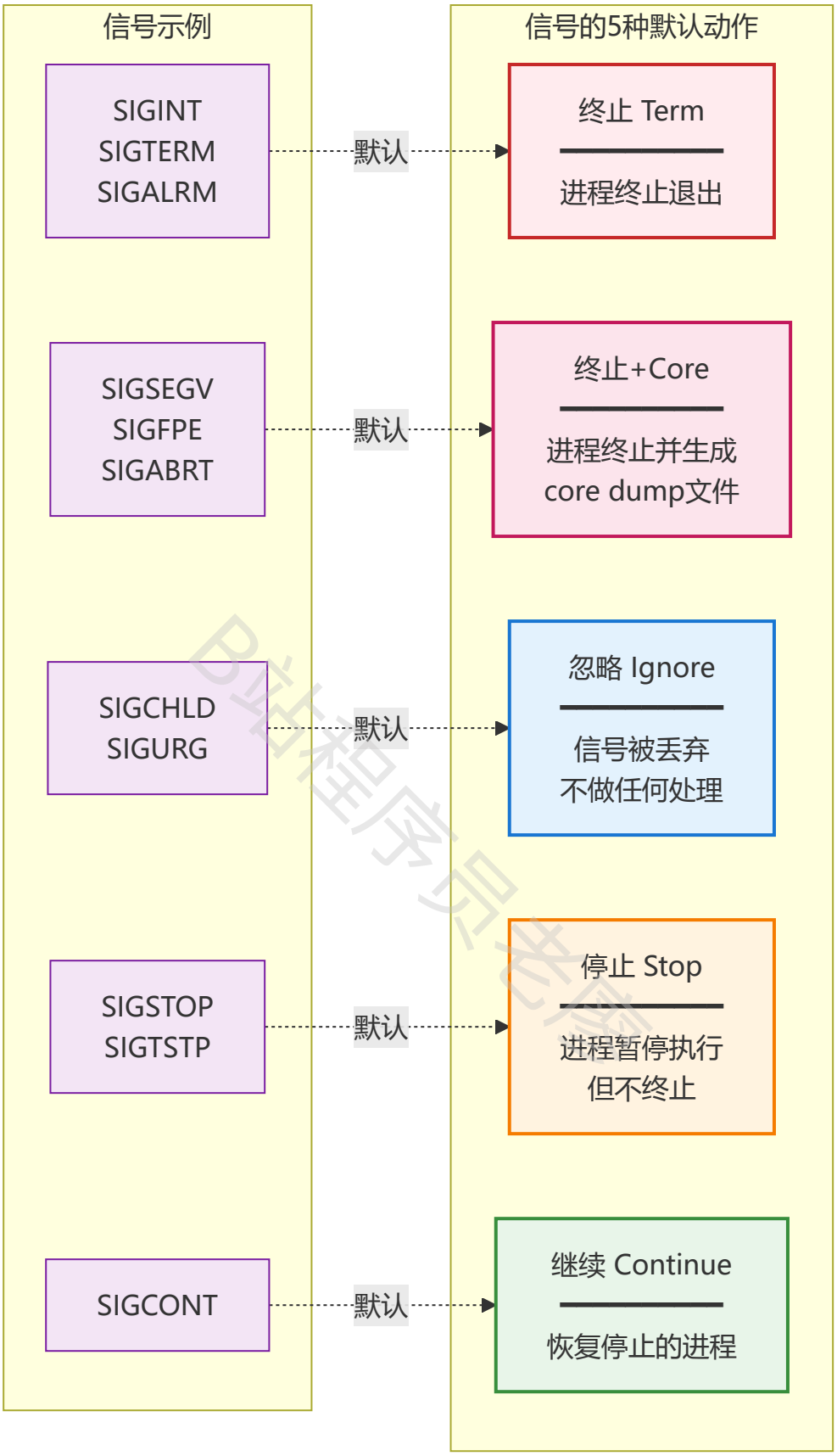
- **异步性**：信号可以在任何时刻到达，打断进程的正常执行流程
- **简单性**：信号只携带一个整数编号，不能传递复杂数据
- **不可靠性**：传统信号可能丢失（多个相同信号只保留一个）

## 6.1.2 常用信号列表

Linux定义了多种标准信号，每个信号有唯一的编号和名称：

| 信号名 | 编号 | 默认动作 | 说明 | 典型场景 |
| --- | --- | --- | --- | --- |
| SIGHUP | 1 | 终止 | 挂断信号（终端断开） | 重新加载配置文件 |
| SIGINT | 2 | 终止 | 中断信号（Ctrl+C） | 用户请求终止 |
| SIGQUIT | 3 | 终止+core | 退出信号（Ctrl+\） | 调试时生成core文件 |
| SIGILL | 4 | 终止+core | 非法指令 | CPU执行非法指令 |
| SIGTRAP | 5 | 终止+core | 断点陷阱 | 调试器断点 |
| SIGABRT | 6 | 终止+core | 异常终止（abort()） | 程序主动终止 |
| SIGFPE | 8 | 终止+core | 浮点异常（除零） | 数学运算错误 |
| SIGKILL | 9 | 终止 | 强制杀死（不可捕获） | 强制结束进程 |
| SIGSEGV | 11 | 终止+core | 段错误（非法内存访问） | 访问无效指针 |
| SIGPIPE | 13 | 终止 | 管道破裂（写入无读端的管道） | 网络连接断开 |
| SIGALRM | 14 | 终止 | 定时器到期 | 实现超时机制 |
| SIGTERM | 15 | 终止 | 终止信号（可捕获） | 优雅关闭程序 |
| SIGCHLD | 17 | 忽略 | 子进程状态改变 | 回收子进程 |
| SIGCONT | 18 | 继续 | 继续执行（如果停止） | 恢复暂停进程 |
| SIGSTOP | 19 | 停止 | 停止执行（不可捕获） | 暂停进程 |
| SIGTSTP | 20 | 停止 | 停止信号（Ctrl+Z） | 用户暂停程序 |
| SIGUSR1 | 10 | 终止 | 用户自定义信号1 | 自定义进程通信 |
| SIGUSR2 | 12 | 终止 | 用户自定义信号2 | 自定义进程通信 |

**信号的默认动作：**




**不可捕获的信号：**

- `SIGKILL (9)`：强制杀死进程，无法被捕获、阻塞或忽略
- `SIGSTOP (19)`：强制停止进程，无法被捕获、阻塞或忽略

```
# 这两个信号总是会生效，无法被程序拦截
$ kill -9 1234   # 强制杀死进程（SIGKILL）
$ kill -19 1234  # 强制停止进程（SIGSTOP）
```

### 6.1.3 查看系统支持的信号

可以使用 `kill -l` 命令查看系统支持的所有信号：

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
...
```

也可以在C程序中使用 `strsignal()` 函数获取信号的描述：

```
#include <string.h>
printf("SIGINT: %s\n", strsignal(SIGINT));  // 输出: Interrupt
```

## 6.2 发送信号

### 6.2.1 使用 kill 命令发送信号

```
# 发送SIGTERM信号（默认，优雅终止）
$ kill 1234

# 发送SIGKILL信号（强制杀死）
$ kill -9 1234
$ kill -KILL 1234

# 发送SIGINT信号（中断）
$ kill -2 1234
$ kill -INT 1234

# 发送SIGUSR1信号（用户自定义）
$ kill -USR1 1234

# 向进程组发送信号（负数PID）
$ kill -TERM -5678  # 向进程组5678的所有进程发送SIGTERM
```

## 6.2.2 使用 kill() 系统调用发送信号

### API 说明

```
#include <signal.h>
#include <sys/types.h>

int kill(pid_t pid, int sig);
```

**功能**：向指定进程或进程组发送信号

**参数**：

- `pid`：目标进程ID
  - `> 0`：发送给指定PID的进程
  - `= 0`：发送给当前进程组的所有进程
  - `= -1`：发送给所有有权限发送的进程（除init）
  - `< -1`：发送给进程组ID为 `|pid|` 的所有进程
- `sig`：信号编号
  - 可以使用信号名宏（如 `SIGTERM`）
  - 传入 `0` 可以检测进程是否存在（不发送实际信号）

**返回值**：

- 成功返回 `0`
- 失败返回 `-1`，设置 `errno`

**常见错误码**：

- `EINVAL`：无效的信号编号
- `EPERM`：没有权限发送信号
- `ESRCH`：目标进程不存在

**示例**：

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>

int main() {
    pid_t target_pid = 1234;

    // 检测进程是否存在
    if (kill(target_pid, 0) == 0) {
        printf("进程 %d 存在\n", target_pid);

        // 发送SIGUSR1信号
        if (kill(target_pid, SIGUSR1) == 0) {
```

```
            printf("成功发送SIGUSR1给进程 %d\n", target_pid);
        } else {
            perror("发送信号失败");
        }
    } else {
        if (errno == ESRCH) {
            printf("进程 %d 不存在\n", target_pid);
        } else if (errno == EPERM) {
            printf("没有权限发送信号给进程 %d\n", target_pid);
        }
    }

    return 0;
}
```

💻 **实践代码**：`src/chapter06/send_signal.c`

## 6.2.3 使用 raise() 向自己发送信号

### API 说明

```
#include <signal.h>

int raise(int sig);
```

**功能**：向调用进程自己发送信号（等价于 `kill(getpid(), sig)`）

**参数**：

- `sig`：信号编号

**返回值**：

- 成功返回 `0`
- 失败返回非零值

**示例**：

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void signal_handler(int sig) {
    printf("收到信号 %d\n", sig);
}

int main() {
    signal(SIGUSR1, signal_handler);

    printf("进程 %d 即将向自己发送SIGUSR1\n", getpid());
    raise(SIGUSR1);  // 向自己发送信号

    printf("信号处理完成\n");
```

```
    return 0;
}
```

## 6.2.4 使用 alarm() 设置定时器信号

### API 说明

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

**功能**：设置一个定时器，`seconds` 秒后向进程发送 `SIGALRM` 信号

**参数**：

- `seconds`：秒数
  - `> 0`：设置定时器
  - `= 0`：取消之前设置的定时器

**返回值**：

- 返回之前设置的定时器剩余秒数
- 如果没有设置过，返回 `0`

**特点**：

- 每个进程只能有一个 `alarm` 定时器
- 调用 `alarm` 会取消之前的定时器
- `SIGALRM` 的默认动作是终止进程

**示例**：

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void alarm_handler(int sig) {
    printf("定时器到期! \n");
}

int main() {
    signal(SIGALRM, alarm_handler);

    printf("设置3秒定时器\n");
    alarm(3);

    // 模拟工作
    for (int i = 1; i <= 5; i++) {
        printf("工作中... %d秒\n", i);
        sleep(1);
    }
```

```
    return 0;
}
```

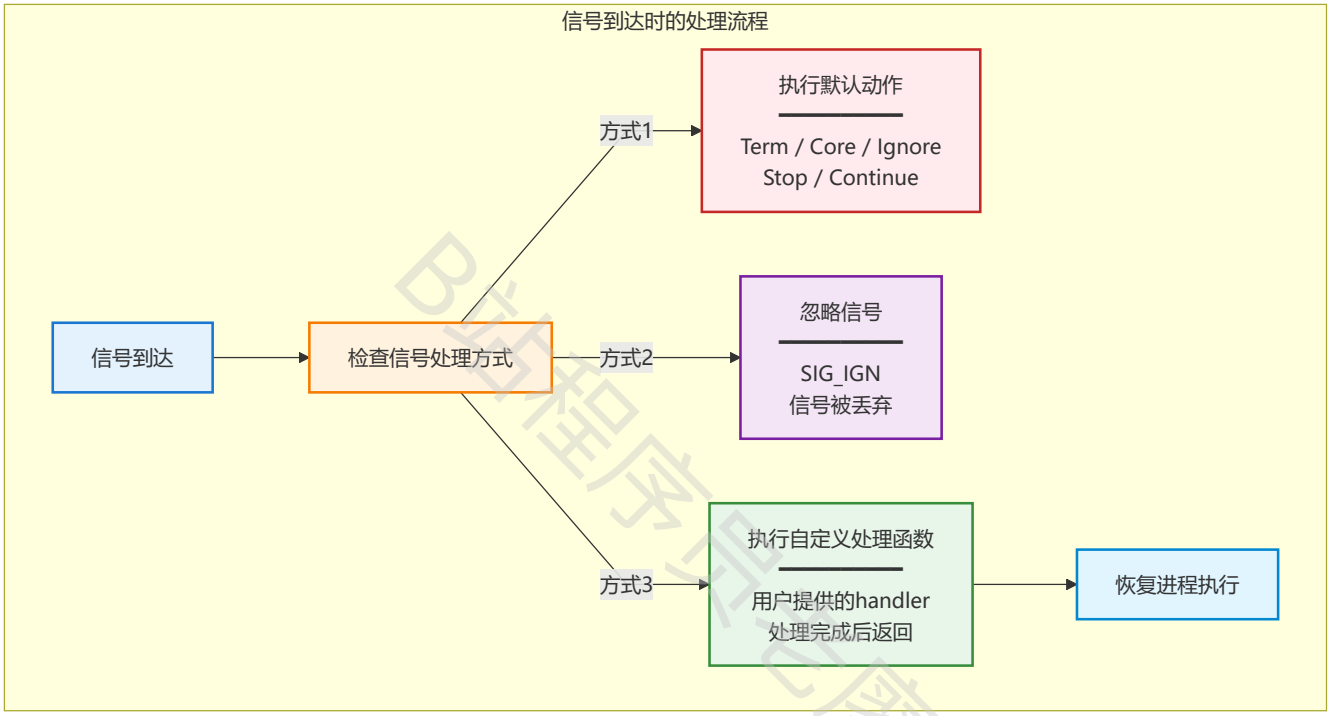
💻 **实践代码**：`src/chapter06/alarm_demo.c`

---

# 6.3 接收和处理信号

## 6.3.1 信号处理的三种方式

当进程收到信号时，可以采取三种处理方式：




**处理方式总结：**

| 方式 | 设置方法 | 说明 |
|---|---|---|
| 默认动作 | 不设置 | 使用系统定义的默认行为 |
| 忽略信号 | `SIG_IGN` | 信号被丢弃，进程不做任何处理 |
| 自定义处理 | 自定义函数指针 | 执行用户定义的信号处理函数 |

## 6.3.2 使用 signal() 注册信号处理函数（不推荐）

### API 说明

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

**功能**：注册信号处理函数（旧式API，不推荐使用）

**参数**：

- `signum`：信号编号
- `handler`：处理函数指针，可以是：
  - 自定义函数地址：`void handler(int sig)`
  - `SIG_DFL`：恢复默认动作
  - `SIG_IGN`：忽略信号

**返回值**：

- 成功返回之前的处理函数指针
- 失败返回 `SIG_ERR`

**不推荐使用的原因**：

- 行为在不同Unix系统上不一致
- 信号处理函数执行一次后可能被自动重置为默认
- 不能可靠地阻塞其他信号
- 无法获取信号的详细信息

**示例**：

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sigint_handler(int sig) {
    printf("\n收到中断信号 (Ctrl+C)，但程序不退出\n");
}

int main() {
    // 注册SIGINT处理函数
    signal(SIGINT, sigint_handler);

    printf("按 Ctrl+C 测试...\n");

    while (1) {
        printf("运行中...\n");
        sleep(1);
    }

    return 0;
}
```

## 6.3.3 使用 sigaction() 注册信号处理函数（推荐）

### API 说明

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

struct sigaction {
    void (*sa_handler)(int);                    // 简单处理函数
    void (*sa_sigaction)(int, siginfo_t *, void *);  // 详细处理函数
    sigset_t sa_mask;                           // 处理期间需要阻塞的信号集
    int sa_flags;                               // 标志位
    void (*sa_restorer)(void);                  // 已废弃，不使用
};
```

**功能**：注册信号处理函数（推荐使用，可靠且功能完整）

**参数**：

- `signum`：信号编号
- `act`：新的信号处理设置
- `oldact`：保存旧的信号处理设置（可为 `NULL`）

**struct sigaction 字段说明**：

| 字段 | 说明 |
| --- | --- |
| `sa_handler` | 简单处理函数，接收一个int参数（信号编号） |
| `sa_sigaction` | 详细处理函数，接收三个参数（需配合 `SA_SIGINFO`） |
| `sa_mask` | 信号掩码：处理当前信号期间需要阻塞的其他信号 |
| `sa_flags` | 标志位，控制信号处理行为 |

**常用 sa_flags 标志位**：

| 标志 | 说明 |
| --- | --- |
| `SA_RESTART` | 被信号中断的系统调用自动重启 |
| `SA_SIGINFO` | 使用 `sa_sigaction` 而不是 `sa_handler` |
| `SA_NODEFER` | 处理期间不自动阻塞当前信号 |
| `SA_RESETHAND` | 处理一次后自动恢复为默认动作 |
| `SA_NOCLDSTOP` | 子进程停止时不发送 `SIGCHLD` |
| `SA_NOCLDWAIT` | 子进程退出时自动回收，不产生僵尸进程 |

**返回值**：

- 成功返回 `0`
- 失败返回 `-1`，设置 `errno`

**示例1：基本用法**

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>

void sigint_handler(int sig) {
    // 注意：printf不是异步信号安全函数，这里仅作演示
    printf("\n收到信号 %d (Ctrl+C)\n", sig);
}

int main() {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));

    sa.sa_handler = sigint_handler;
    sigemptyset(&sa.sa_mask);  // 清空信号掩码
    sa.sa_flags = SA_RESTART;  // 自动重启被中断的系统调用

    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction");
        return 1;
    }

    printf("按 Ctrl+C 测试...\n");

    while (1) {
        sleep(1);
    }

    return 0;
}
```



💻 **实践代码**：`src/chapter06/sigaction_basic.c`

**示例2：使用 SA_SIGINFO 获取详细信息**

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>

void siginfo_handler(int sig, siginfo_t *info, void *context) {
    printf("\n收到信号 %d\n", sig);
    printf("  发送者PID: %d\n", info->si_pid);
    printf("  发送者UID: %d\n", info->si_uid);
    printf("  信号代码: %d\n", info->si_code);
```

```
}

int main() {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));

    sa.sa_sigaction = siginfo_handler;  // 使用详细处理函数
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;  // 启用详细信息

    sigaction(SIGUSR1, &sa, NULL);

    printf("进程PID: %d\n", getpid());
    printf("等待SIGUSR1信号（使用 kill -USR1 %d）\n", getpid());

    while (1) {
        sleep(1);
    }

    return 0;
}
```

💻 **实践代码**：`src/chapter06/sigaction_info.c`

## 6.3.4 信号处理函数的编写规则

信号处理函数运行在异步上下文中，必须遵守以下规则：

**规则1：只调用异步信号安全（async-signal-safe）的函数**

不安全的函数包括：

- `printf`、`fprintf` 等标准I/O函数（可能死锁）
- `malloc`、`free`（可能破坏堆结构）
- 大多数库函数

安全的函数包括：

- `write()`、`read()`、`_exit()`
- `signal()`、`sigaction()`
- `getpid()`、`kill()`

完整列表见 `man 7 signal-safety`。

**规则2：访问全局变量时使用 volatile sig_atomic_t**

```
#include <signal.h>

volatile sig_atomic_t got_signal = 0;  // 安全的信号标志

void signal_handler(int sig) {
    got_signal = 1;  // 只设置标志，不做复杂操作
}
```

```c
int main() {
    signal(SIGINT, signal_handler);

    while (!got_signal) {
        // 主循环
    }

    printf("收到信号，程序退出\n");
    return 0;
}
```

**规则3：保持处理函数简短**

推荐模式：

```c
volatile sig_atomic_t need_reload = 0;

void sighup_handler(int sig) {
    need_reload = 1;  // 只设置标志
}

int main() {
    signal(SIGHUP, sighup_handler);

    while (1) {
        if (need_reload) {
            need_reload = 0;
            reload_config();  // 在主循环中处理
        }

        // 正常工作
        do_work();
    }

    return 0;
}
```



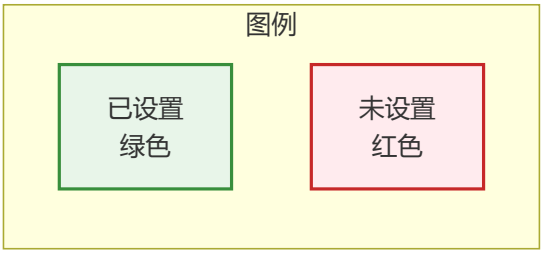
💻 **实践代码**：`src/chapter06/signal_safe.c`

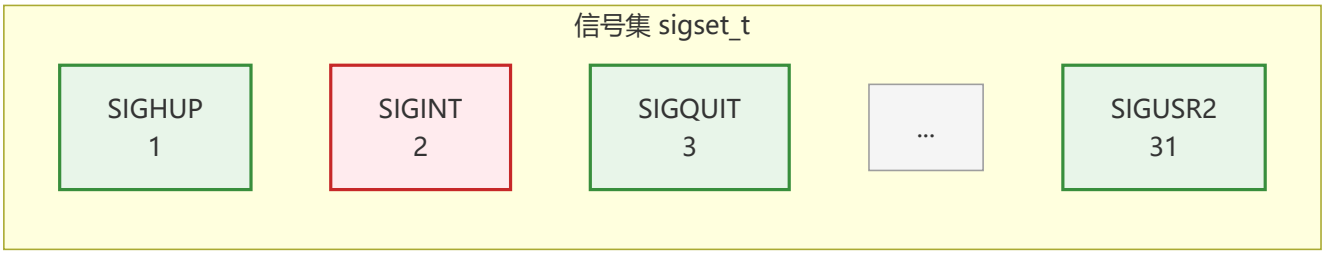
---

# 6.4 信号集和信号掩码

## 6.4.1 什么是信号集？

**信号集（Signal Set）** 是一个数据结构，用于表示一组信号。




## 6.4.2 信号集操作函数

### API 说明

```
#include <signal.h>

int sigemptyset(sigset_t *set);          // 清空信号集
int sigfillset(sigset_t *set);           // 填充所有信号
int sigaddset(sigset_t *set, int signum);   // 添加信号
int sigdelset(sigset_t *set, int signum);   // 删除信号
int sigismember(const sigset_t *set, int signum);  // 检测信号是否在集合中
```

**功能**：操作信号集

**返回值**：

- `sigemptyset`、`sigfillset`、`sigaddset`、`sigdelset`：成功返回0，失败返回-1
- `sigismember`：在集合中返回1，不在返回0，出错返回-1

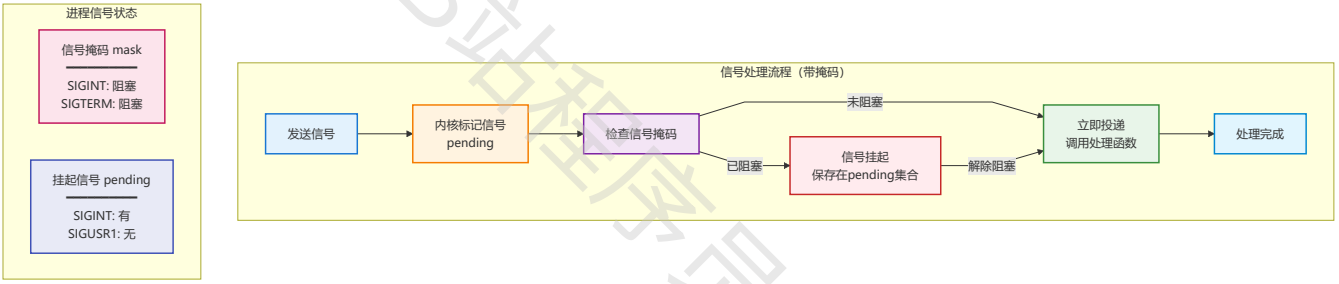
**示例**：

```
#include <signal.h>
#include <stdio.h>

int main() {
    sigset_t set;

    // 清空信号集
    sigemptyset(&set);

    // 添加信号
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGTERM);

    // 检测信号
```

```
    if (sigismember(&set, SIGINT)) {
        printf("SIGINT 在信号集中\n");
    }

    if (!sigismember(&set, SIGUSR1)) {
        printf("SIGUSR1 不在信号集中\n");
    }

    // 删除信号
    sigdelset(&set, SIGINT);

    // 填充所有信号
    sigfillset(&set);

    return 0;
}
```

## 6.4.3 信号掩码（Signal Mask）

每个进程都有一个**信号掩码**，用于指定哪些信号被阻塞（暂时不处理）。




**关键概念：**

- **信号掩码（mask）**：指定哪些信号被阻塞
- **挂起信号（pending）**：已发送但因被阻塞而尚未处理的信号
- **阻塞（block）≠ 忽略（ignore）**：
    - 阻塞的信号会被保存，解除阻塞后会处理
    - 忽略的信号会被丢弃，永远不会处理

## 6.4.4 使用 sigprocmask() 修改信号掩码

### API 说明

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

**功能**：修改进程的信号掩码

**参数**：

- `how`：操作方式

    - `SIG_BLOCK`：将 `set` 中的信号添加到掩码中（mask = mask | set）
    - `SIG_UNBLOCK`：从掩码中移除 `set` 中的信号（mask = mask & ~set）
    - `SIG_SETMASK`：直接设置掩码为 `set`（mask = set）
- `set`：要操作的信号集（如果为 `NULL`，不修改掩码）
- `oldset`：保存旧的信号掩码（如果为 `NULL`，不保存）

**返回值**：

- 成功返回 `0`
- 失败返回 `-1`，设置 `errno`

**示例**：

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sigint_handler(int sig) {
    printf("\n收到SIGINT信号\n");
}

int main() {
    sigset_t newmask, oldmask;

    signal(SIGINT, sigint_handler);

    // 阻塞SIGINT
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);

    printf("SIGINT已阻塞，按Ctrl+C不会立即响应\n");
    sleep(5);

    // 解除阻塞
    printf("解除阻塞，如果之前按过Ctrl+C，现在会处理\n");
    sigprocmask(SIG_SETMASK, &oldmask, NULL);

    sleep(3);
    printf("程序结束\n");

    return 0;
}
```



💻 **实践代码**：`src/chapter06/sigprocmask_demo.c`

## 6.4.5 使用 sigpending() 查询挂起的信号

### API 说明

```
#include <signal.h>

int sigpending(sigset_t *set);
```

**功能**：获取当前进程中挂起（pending）的信号集

**参数**：

- `set`：用于保存挂起的信号集

**返回值**：

- 成功返回 `0`
- 失败返回 `-1`

**示例**：



```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void print_pending() {
    sigset_t pending;
    sigpending(&pending);

    printf("当前挂起的信号: ");
    if (sigismember(&pending, SIGINT)) printf("SIGINT ");
    if (sigismember(&pending, SIGTERM)) printf("SIGTERM ");
    if (sigismember(&pending, SIGUSR1)) printf("SIGUSR1 ");
    printf("\n");
}

int main() {
    sigset_t mask;

    // 阻塞SIGINT
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigprocmask(SIG_BLOCK, &mask, NULL);

    printf("SIGINT已阻塞，按Ctrl+C测试...\n");

    for (int i = 0; i < 5; i++) {
        sleep(1);
        print_pending();
    }

    printf("解除阻塞\n");
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
```

```
    return 0;
}
```

💻 **实践代码**：`src/chapter06/sigpending_demo.c`

---

# 6.5 信号与系统调用

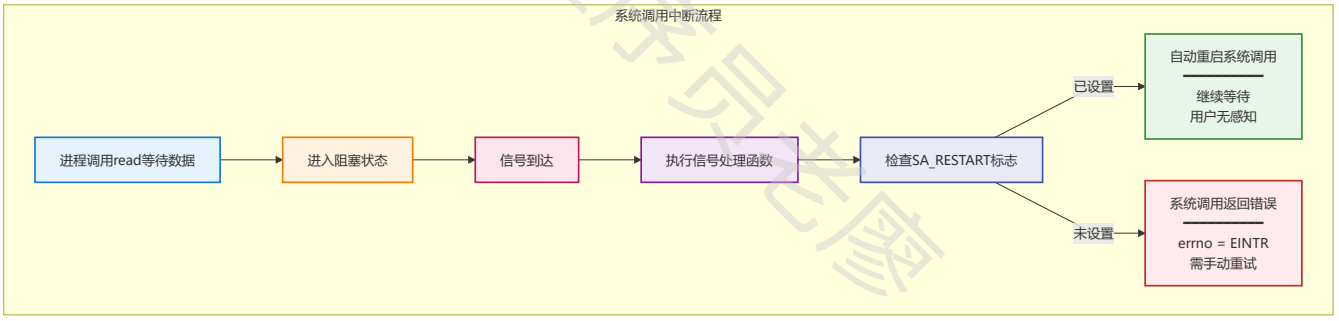
## 6.5.1 慢速系统调用被信号中断

某些系统调用（称为**慢速系统调用**）可能会被信号中断：

**慢速系统调用包括：**

- `read()`：从管道、终端、网络socket读取
- `write()`：向管道、终端、网络socket写入
- `open()`：打开特殊文件（如FIFO）
- `wait()`、`waitpid()`：等待子进程
- `sleep()`、`nanosleep()`：休眠
- `pause()`、`sigsuspend()`：等待信号

**被中断后的行为：**




**示例：读取被中断**

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>

void sigint_handler(int sig) {
    write(STDOUT_FILENO, "收到信号\n", 13);
}

int main() {
    struct sigaction sa;
    sa.sa_handler = sigint_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;  // 不设置SA_RESTART，read会被中断
    sigaction(SIGINT, &sa, NULL);
```

```
    char buf[100];
    printf("输入文本（按Ctrl+C测试中断）：\n");

    ssize_t n = read(STDIN_FILENO, buf, sizeof(buf));
    if (n == -1) {
        if (errno == EINTR) {
            printf("read被信号中断\n");
        } else {
            perror("read");
        }
    } else {
        printf("读取了 %zd 字节\n", n);
    }

    return 0;
}
```

## 6.5.2 使用 SA_RESTART 自动重启系统调用

将 `sa_flags` 设置为 `SA_RESTART` 可以让被中断的系统调用自动重启：

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sigint_handler(int sig) {
    write(STDOUT_FILENO, "收到信号但read会继续\n", 27);
}

int main() {
    struct sigaction sa;
    sa.sa_handler = sigint_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;  // 设置自动重启
    sigaction(SIGINT, &sa, NULL);

    char buf[100];
    printf("输入文本（按Ctrl+C测试，read不会中断）：\n");

    ssize_t n = read(STDIN_FILENO, buf, sizeof(buf));
    printf("读取了 %zd 字节\n", n);

    return 0;
}
```



💻 **实践代码**：`src/chapter06/signal_restart.c`

### 6.5.3 手动处理 EINTR

对于不支持 `SA_RESTART` 或需要更精细控制的情况，可以手动重试：

```
#include <errno.h>
#include <unistd.h>

ssize_t safe_read(int fd, void *buf, size_t count) {
    ssize_t n;

    while (1) {
        n = read(fd, buf, count);
        if (n == -1 && errno == EINTR) {
            continue;  // 被信号中断，重试
        }
        break;
    }

    return n;
}

// 使用
char buf[100];
ssize_t n = safe_read(STDIN_FILENO, buf, sizeof(buf));
```

## 6.6 实战案例

### 6.6.1 案例1：优雅关闭服务器

💻 **实践代码**：`src/chapter06/graceful_shutdown.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>

volatile sig_atomic_t shutdown_requested = 0;

void signal_handler(int sig) {
    if (sig == SIGTERM || sig == SIGINT) {
        shutdown_requested = 1;
    }
}

void cleanup() {
    printf("清理资源...\n");
    printf("  关闭数据库连接\n");
    printf("  保存日志\n");
    printf("  释放内存\n");
```

```
}

int main() {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = signal_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    sigaction(SIGTERM, &sa, NULL);
    sigaction(SIGINT, &sa, NULL);

    printf("服务器启动，PID: %d\n", getpid());
    printf("使用 kill -TERM %d 或 Ctrl+C 优雅关闭\n", getpid());

    int request_count = 0;
    while (!shutdown_requested) {
        printf("处理请求 #%d\n", ++request_count);
        sleep(2);
    }

    printf("\n收到关闭信号，开始优雅关闭...\n");
    cleanup();
    printf("服务器已关闭\n");

    return 0;
}
```



## 6.6.2 案例2：使用SIGCHLD回收子进程

💻 **实践代码**：`src/chapter06/sigchld_handler.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>

void sigchld_handler(int sig) {
    int status;
    pid_t pid;

    // 循环回收所有已退出的子进程
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        if (WIFEXITED(status)) {
            printf("子进程 %d 退出，状态码: %d\n",
                   pid, WIFEXITED(status));
        }
    }
}
```

```c
int main() {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = sigchld_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;

    sigaction(SIGCHLD, &sa, NULL);

    printf("父进程 PID: %d\n", getpid());

    // 创建3个子进程
    for (int i = 0; i < 3; i++) {
        pid_t pid = fork();

        if (pid == 0) {
            // 子进程
            printf("子进程 %d 启动\n", getpid());
            sleep(i + 1);  // 不同的睡眠时间
            printf("子进程 %d 退出\n", getpid());
            exit(i);
        }
    }

    // 父进程继续工作
    printf("父进程继续工作，子进程会自动回收\n");
    for (int i = 0; i < 5; i++) {
        printf("父进程工作中... %d\n", i);
        sleep(1);
    }

    printf("父进程退出\n");
    return 0;
}
```



### 6.6.3 案例3：实现超时机制

💻 **实践代码**：`src/chapter06/timeout_read.c`

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <setjmp.h>
#include <string.h>

static sigjmp_buf jmpbuf;
static volatile sig_atomic_t timeout_occurred = 0;

void alarm_handler(int sig) {
    timeout_occurred = 1;
    siglongjmp(jmpbuf, 1);  // 跳转回sigsetjmp
}
```

```c
int main() {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = alarm_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGALRM, &sa, NULL);

    char buf[100];
    printf("请在5秒内输入文本：\n");

    if (sigsetjmp(jmpbuf, 1) == 0) {
        alarm(5);  // 设置5秒超时

        ssize_t n = read(STDIN_FILENO, buf, sizeof(buf) - 1);
        alarm(0);  // 取消定时器

        if (n > 0) {
            buf[n] = '\0';
            printf("读取成功: %s\n", buf);
        }
    } else {
        printf("\n超时！未读取到输入\n");
    }

    return 0;
}
```



# 6.7 信号的可靠性问题

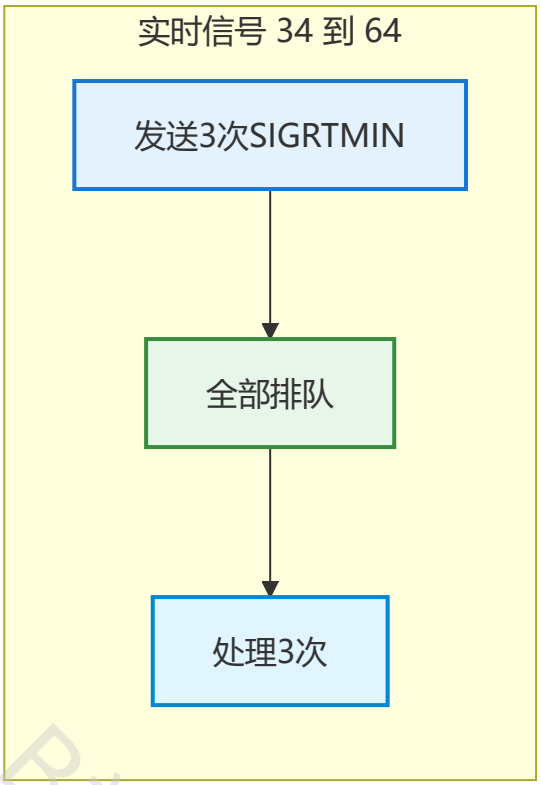
## 6.7.1 不可靠信号 vs 可靠信号

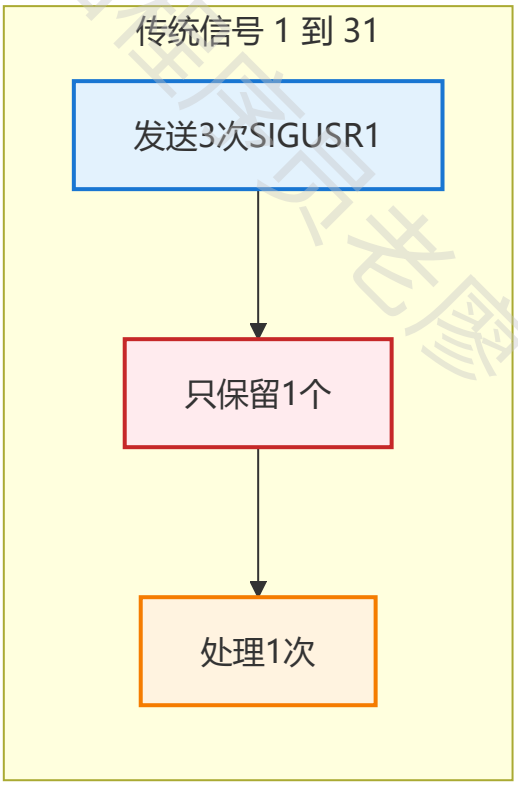
**不可靠信号（1-31）**：

- 传统Unix信号
- 可能丢失（多个相同信号只保留一个）
- 不支持排队

**可靠信号（34-64，实时信号）**：

- POSIX实时信号（`SIGRTMIN`到`SIGRTMAX`）
- 支持排队（不会丢失）
- 可以携带额外数据






### 6.7.2 使用实时信号

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
```

```c
void rt_handler(int sig, siginfo_t *info, void *context) {
    printf("收到实时信号 %d，携带数据: %d\n", sig, info->si_value.sival_int);
}

int main() {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_sigaction = rt_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;

    sigaction(SIGRTMIN, &sa, NULL);

    printf("进程PID: %d\n", getpid());

    // 发送3次实时信号，携带不同数据
    union sigval value;
    for (int i = 1; i <= 3; i++) {
        value.sival_int = i * 100;
        sigqueue(getpid(), SIGRTMIN, value);
    }

    sleep(1);  // 等待信号处理
    return 0;
}
```

💻 **实践代码**：`src/chapter06/realtime_signal.c`



# 6.8 API快速参考

## 6.8.1 发送信号

| 函数 | 功能 | 说明 |
| --- | --- | --- |
| `kill(pid, sig)` | 向进程发送信号 | 最常用 |
| `raise(sig)` | 向自己发送信号 | 等价于`kill(getpid(), sig)` |
| `alarm(seconds)` | 定时发送SIGALRM | 只能有一个定时器 |
| `sigqueue(pid, sig, value)` | 发送实时信号并携带数据 | 可靠信号 |

## 6.8.2 注册信号处理函数

| 函数 | 功能 | 推荐度 |
| --- | --- | --- |
| `signal(sig, handler)` | 注册处理函数（旧式） | ❌ 不推荐 |
| `sigaction(sig, act, oldact)` | 注册处理函数（新式） | ✅ 推荐使用 |

### 6.8.3 信号集操作

| 函数 | 功能 |
| --- | --- |
| `sigemptyset(set)` | 清空信号集 |
| `sigfillset(set)` | 填充所有信号 |
| `sigaddset(set, sig)` | 添加信号 |
| `sigdelset(set, sig)` | 删除信号 |
| `sigismember(set, sig)` | 检测信号是否在集合中 |

### 6.8.4 信号掩码操作

| 函数 | 功能 |
| --- | --- |
| `sigprocmask(how, set, oldset)` | 修改信号掩码 |
| `sigpending(set)` | 查询挂起的信号 |
| `sigsuspend(mask)` | 原子地修改掩码并等待信号 |

### 6.8.5 等待信号

| 函数 | 功能 |
| --- | --- |
| `pause()` | 暂停进程，直到收到信号 |
| `sigsuspend(mask)` | 原子地设置掩码并等待信号 |



## 6.9 小结

本章介绍了Linux信号机制的核心概念和实践：

**核心知识点：**

1. **信号基础**：
   - 信号是异步通知机制
   - 每个信号有编号、名称和默认动作
   - `SIGKILL` 和 `SIGSTOP` 不可捕获
2. **信号处理**：
   - 使用 `sigaction()`（推荐）而不是 `signal()`
   - 信号处理函数要简短、使用异步信号安全函数
   - 使用 `volatile sig_atomic_t` 访问全局变量
3. **信号掩码**：

- 使用信号集管理多个信号
- `sigprocmask()` 阻塞/解除阻塞信号
- 阻塞的信号会挂起，解除后处理

4. **系统调用中断**：
   - 慢速系统调用可能被信号中断
   - 使用 `SA_RESTART` 自动重启
   - 或手动检测 `EINTR` 并重试
5. **实战应用**：
   - 优雅关闭服务器（`SIGTERM`、`SIGINT`）
   - 回收子进程（`SIGCHLD`）
   - 实现超时机制（`SIGALRM`）

**学习建议：**

1. 理解信号的异步性和限制
2. 掌握 `sigaction()` 的完整用法
3. 实践各种信号处理场景
4. 注意信号安全性问题

**下一章预告：**

第07章将介绍进程间通信（IPC）机制，包括管道、消息队列、共享内存等，让进程之间能够交换更复杂的数据。

---

## 6.10 练习题

1. **基础练习**：
   - 编写程序捕获 `SIGINT`，按3次 `Ctrl+C` 才退出
   - 实现一个简单的定时提醒工具（使用 `SIGALRM`）
2. **进阶练习**：
   - 实现一个进程管理工具，可以向指定进程发送不同信号
   - 编写一个多进程程序，父进程通过 `SIGUSR1`/`SIGUSR2` 控制子进程
3. **实战练习**：
   - 实现一个支持优雅关闭的简单HTTP服务器
   - 编写一个支持超时的文件下载工具

---

💡 **提示**：信号机制是系统编程的基础，理解信号的异步性和正确处理信号中断是编写健壮程序的关键。

# 第07章 进程间通信（IPC）

## 引言：为什么需要进程间通信?

在实际应用中，多个进程经常需要协同工作、共享数据或相互通知事件。

**场景1：父子进程协作**

父进程启动子进程处理任务，需要获取处理结果：

```
父进程 → 创建管道 → fork子进程 → 子进程通过管道发送结果 → 父进程接收
```

**场景2：多进程服务器**

Web服务器使用多个worker进程处理请求，需要共享配置数据：

```
主进程（管理配置）
├── worker1（共享内存读取配置）
├── worker2（共享内存读取配置）
└── worker3（共享内存读取配置）
```

**场景3：进程间消息传递**

日志进程接收其他进程发送的日志消息：

```
进程A → 消息队列 → 日志进程
进程B → 消息队列 → 日志进程
进程C → 消息队列 → 日志进程
```



**本章解决的问题：**

- ✅ 进程间如何传递数据？（管道、消息队列）
- ✅ 进程间如何共享内存？（共享内存）
- ✅ 进程间如何同步？（信号量）
- ✅ 不同IPC机制的适用场景？

**学完本章你能做什么：**

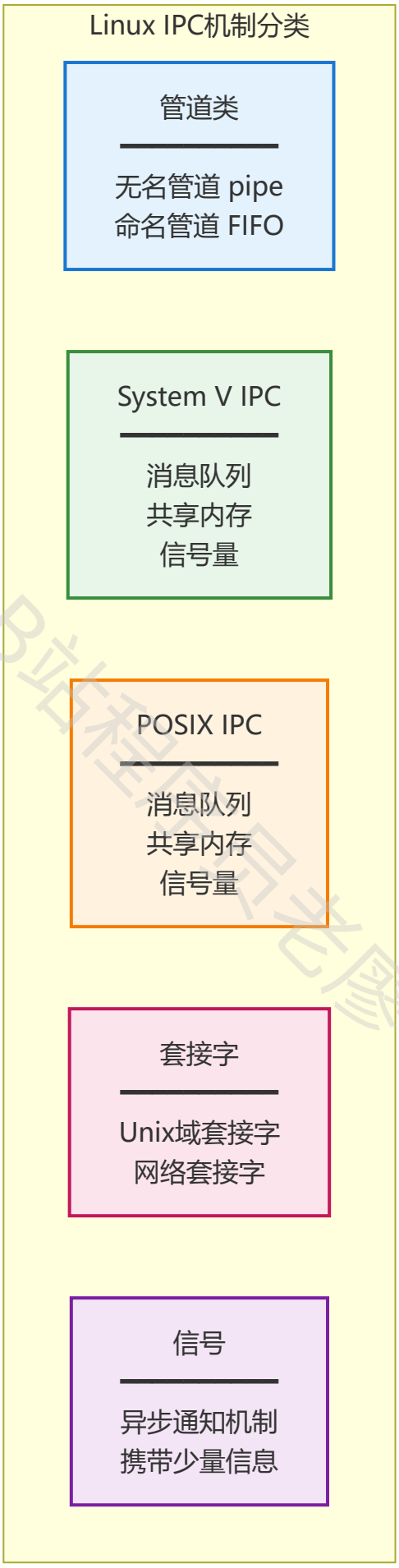
- 实现父子进程通信（管道）
- 构建多进程应用（消息队列、共享内存）
- 理解IPC性能特点
- 选择合适的IPC机制

---

# 7.1 IPC机制概览

## 7.1.1 Linux支持的IPC机制

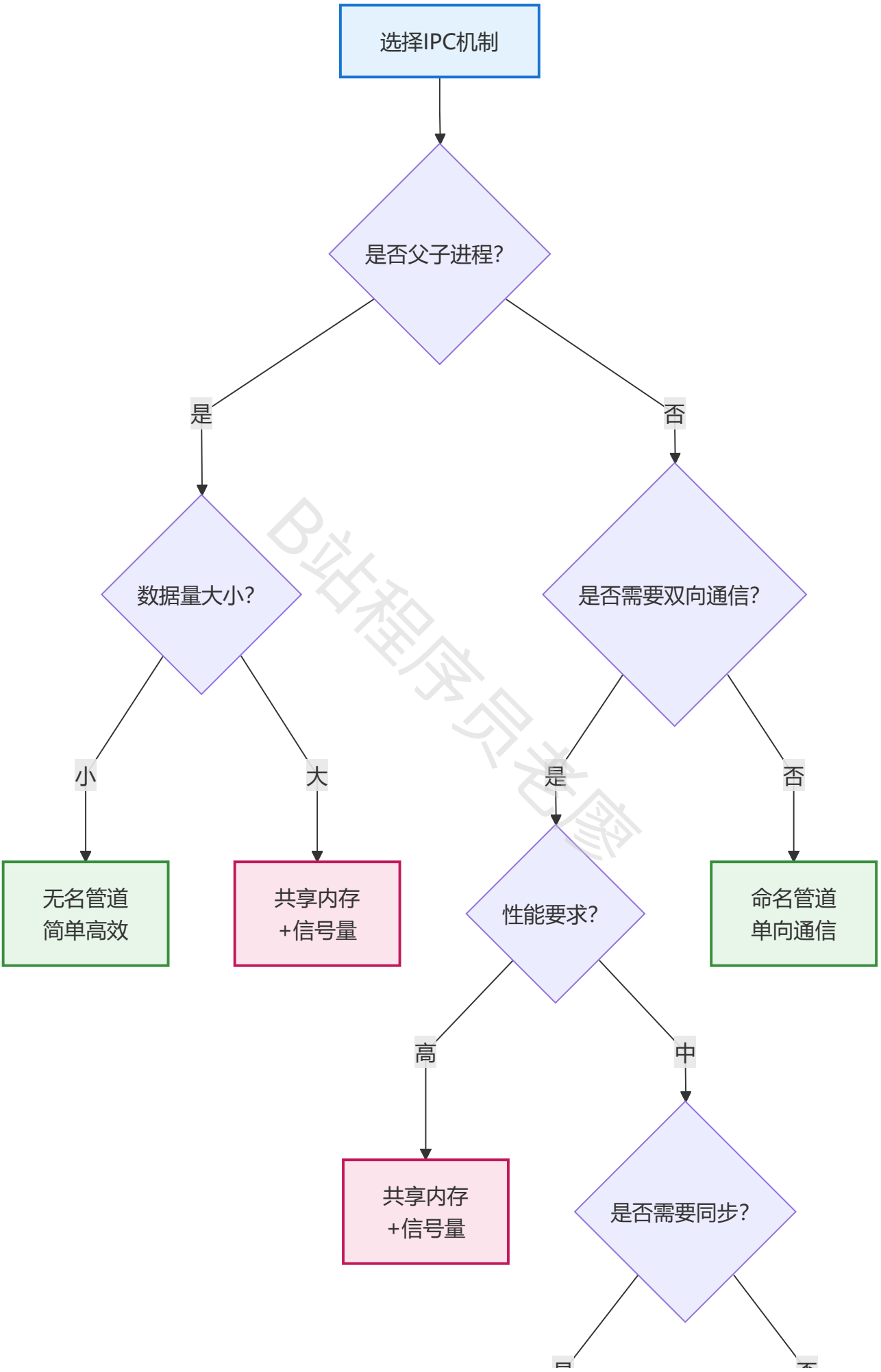
Linux提供了多种进程间通信方式：

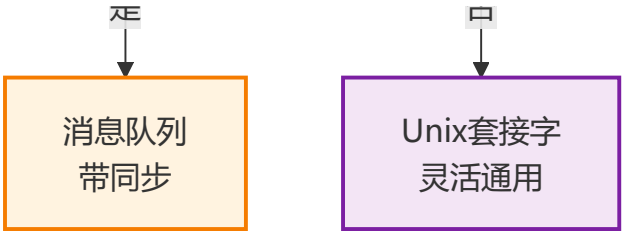

Linux IPC机制分类
管道类
无名管道 pipe
命名管道 FIFO
System V IPC
消息队列
共享内存
信号量
POSIX IPC
消息队列
共享内存
信号量
套接字
Unix域套接字
网络套接字
信号
异步通知机制
携带少量信息

### 7.1.2 IPC机制对比

| IPC机制 | 通信方向 | 数据量 | 速度 | 使用场景 | 难度 |
| --- | --- | --- | --- | --- | --- |
| **无名管道** | 单向 | 中等 | 快 | 父子进程通信 | ⭐ |
| **命名管道FIFO** | 单向 | 中等 | 快 | 无亲缘关系进程 | ⭐⭐ |
| **消息队列** | 双向 | 中等 | 中 | 多进程消息传递 | ⭐⭐⭐ |
| **共享内存** | 双向 | 大 | 最快 | 大数据共享 | ⭐⭐⭐⭐ |
| **信号量** | - | 无数据 | - | 进程同步 | ⭐⭐⭐ |
| **Unix套接字** | 双向 | 大 | 快 | 本地进程通信 | ⭐⭐⭐ |

### 7.1.3 IPC选择决策


选择IPC机制
是否父子进程?
是
否
数据量大小?
是否需要双向通信?
小
大
是
否
无名管道
简单高效
共享内存
+信号量
性能要求?
命名管道
单向通信
高
中
共享内存
+信号量
是否需要同步?

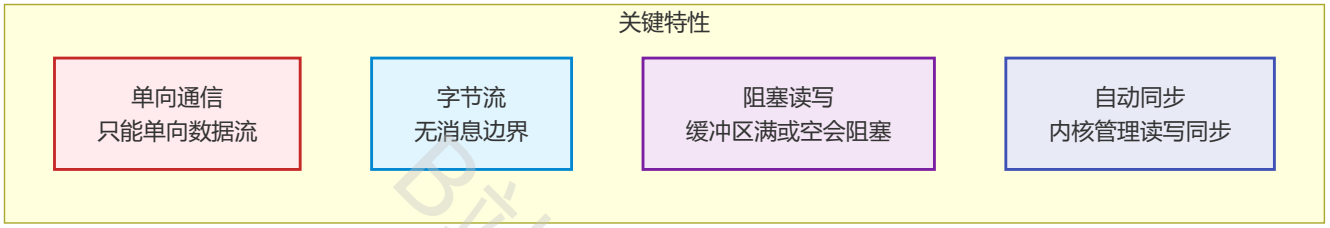


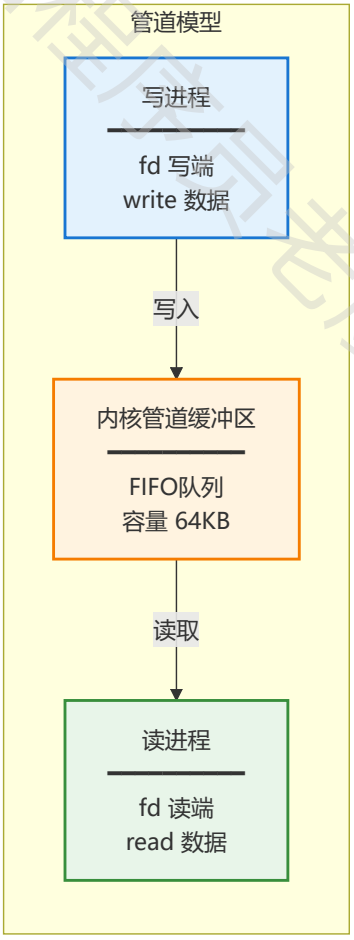

## 7.2 无名管道（Pipe）

### 7.2.1 什么是管道?

**管道（Pipe）** 是一个单向的数据通道，数据从写端流向读端。

**管道的特点:**




**管道的限制:**

- 只能用于有亲缘关系的进程（父子、兄弟）

- 单向通信（**需要双向通信则创建两个管道**）
- 数据一旦读出，缓冲区中就不再保留

## 7.2.2 创建管道 pipe()

### API 说明

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

**功能**：创建一个管道，返回两个文件描述符

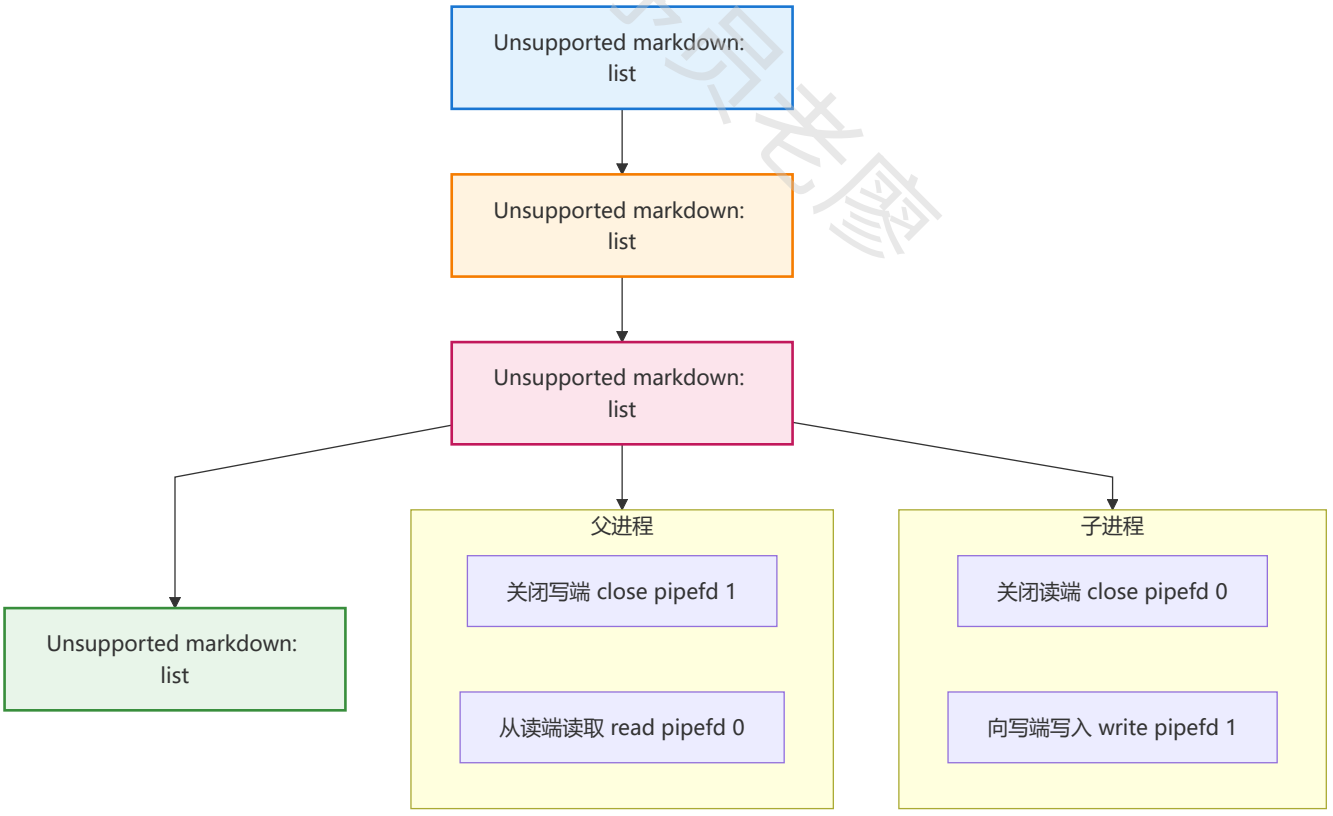
**参数**：

- `pipefd[2]`：用于返回两个文件描述符
  - `pipefd[0]`：读端（read end）
  - `pipefd[1]`：写端（write end）

**返回值**：

- 成功返回 `0`，`pipefd[0]` 和 `pipefd[1]` 被设置
- 失败返回 `-1`，设置 `errno`

**使用流程**：




**示例：基本用法**

```
#include <stdio.h>
```

```c
#include <unistd.h>
#include <string.h>

int main() {
    int pipefd[2];
    pid_t pid;
    char buf[100];

    // 1. 创建管道
    if (pipe(pipefd) == -1) {
        perror("pipe");
        return 1;
    }

    printf("管道创建成功\n");
    printf("  读端: fd=%d\n", pipefd[0]);
    printf("  写端: fd=%d\n", pipefd[1]);

    // 2. fork子进程
    pid = fork();

    if (pid == -1) {
        perror("fork");
        return 1;
    }

    if (pid == 0) {
        // 子进程：写入数据
        close(pipefd[0]);  // 关闭读端

        const char *msg = "Hello from child!";
        write(pipefd[1], msg, strlen(msg) + 1);
        printf("子进程：已发送消息\n");

        close(pipefd[1]);
        return 0;
    } else {
        // 父进程：读取数据
        close(pipefd[1]);  // 关闭写端

        ssize_t n = read(pipefd[0], buf, sizeof(buf));
        printf("父进程：收到消息(%zd字节): %s\n", n, buf);

        close(pipefd[0]);
    }

    return 0;
}
```



💻 **实践代码**：`src/chapter07/pipe_basic.c`

## 7.2.3 管道的读写特性

### 1. 阻塞行为

| 场景 | 读操作 | 写操作 |
| --- | --- | --- |
| 管道有数据，有写端 | 返回数据 | 正常写入 |
| 管道无数据，有写端 | **阻塞等待** | 正常写入 |
| 管道有数据，无写端 | 返回数据 | - |
| 管道无数据，无写端 | **返回0（EOF）** | - |
| 管道已满，有读端 | 正常读取 | **阻塞等待** |
| 管道已满，无读端 | - | **收到SIGPIPE** |

### 2. 管道容量

```
# 查看管道容量（通常是64KB）
$ ulimit -p    # pipe size
$ cat /proc/sys/fs/pipe-max-size
```



**示例：演示管道阻塞**

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {
    int pipefd[2];
    pipe(pipefd);

    if (fork() == 0) {
        // 子进程：延迟后写入
        close(pipefd[0]);

        sleep(3);
        printf("子进程：3秒后写入数据\n");
        write(pipefd[1], "Data", 5);

        close(pipefd[1]);
        return 0;
    }

    // 父进程：立即读取（会阻塞）
    close(pipefd[1]);

    char buf[100];
    printf("父进程：等待数据...（会阻塞）\n");
    ssize_t n = read(pipefd[0], buf, sizeof(buf));
    printf("父进程：收到数据(%zd字节): %s\n", n, buf);
```

```
    close(pipefd[0]);
    return 0;
}
```

💻 **实践代码**：`src/chapter07/pipe_block.c`

## 7.2.4 实战案例：父子进程执行命令

实现类似 `ls | wc -l` 的功能：

💻 **实践代码**：`src/chapter07/pipe_exec.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int pipefd[2];
    pid_t pid1, pid2;

    // 创建管道
    if (pipe(pipefd) == -1) {
        perror("pipe");
        return 1;
    }

    // 第一个子进程：执行 ls
    pid1 = fork();
    if (pid1 == 0) {
        close(pipefd[0]);  // 关闭读端

        // 重定向标准输出到管道写端
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[1]);

        // 执行 ls
        execlp("ls", "ls", "-l", NULL);
        perror("execlp ls");
        exit(1);
    }

    // 第二个子进程：执行 wc -l
    pid2 = fork();
    if (pid2 == 0) {
        close(pipefd[1]);  // 关闭写端

        // 重定向标准输入到管道读端
        dup2(pipefd[0], STDIN_FILENO);
        close(pipefd[0]);

        // 执行 wc -l
```

```
        execlp("wc", "wc", "-l", NULL);
        perror("execlp wc");
        exit(1);
    }

    // 父进程：关闭管道两端，等待子进程
    close(pipefd[0]);
    close(pipefd[1]);

    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);

    return 0;
}
```

# 7.3 命名管道（FIFO）

## 7.3.1 什么是FIFO?

**FIFO（First In First Out）** 也叫命名管道，是一个特殊的文件，可以让无亲缘关系的进程通信。

**无名管道 vs 命名管道**

## 7.3.2 创建FIFO mkfifo()

### API 说明

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

**功能**：创建一个命名管道文件

**参数**：

- `pathname`：FIFO文件路径
- `mode`：权限（如 `0666`）

**返回值**：

- 成功返回 `0`
- 失败返回 `-1`，设置 `errno`

**命令行创建**：

```
# 使用mkfifo命令创建
$ mkfifo /tmp/myfifo

# 查看FIFO文件（注意类型为p）
$ ls -l /tmp/myfifo
prw-rw-r-- 1 user user 0 Oct 11 10:00 /tmp/myfifo

# 删除FIFO
$ rm /tmp/myfifo
```



**示例：创建和使用FIFO**

写进程（`fifo_writer.c`）：

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>

int main() {
    const char *fifo_path = "/tmp/myfifo";

    // 创建FIFO
    if (mkfifo(fifo_path, 0666) == -1) {
        perror("mkfifo");
    }

    printf("打开FIFO写端...\n");
```

```
    int fd = open(fifo_path, O_WRONLY);  // 会阻塞直到有读端打开
    if (fd == -1) {
        perror("open");
        return 1;
    }

    printf("开始写入数据\n");
    for (int i = 1; i <= 5; i++) {
        char buf[100];
        snprintf(buf, sizeof(buf), "Message %d", i);
        write(fd, buf, strlen(buf) + 1);
        printf("已发送: %s\n", buf);
        sleep(1);
    }

    close(fd);
    unlink(fifo_path);  // 删除FIFO文件
    return 0;
}
```

读进程（`fifo_reader.c`）：

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    const char *fifo_path = "/tmp/myfifo";

    printf("打开FIFO读端...\n");
    int fd = open(fifo_path, O_RDONLY);  // 会阻塞直到有写端打开
    if (fd == -1) {
        perror("open");
        return 1;
    }

    printf("开始读取数据\n");
    char buf[100];
    ssize_t n;
    while ((n = read(fd, buf, sizeof(buf))) > 0) {
        printf("收到消息: %s\n", buf);
    }

    printf("写端关闭，读取完毕\n");
    close(fd);
    return 0;
}
```



💻 **实践代码**：`src/chapter07/fifo_writer.c`，`src/chapter07/fifo_reader.c`

**运行方式**：

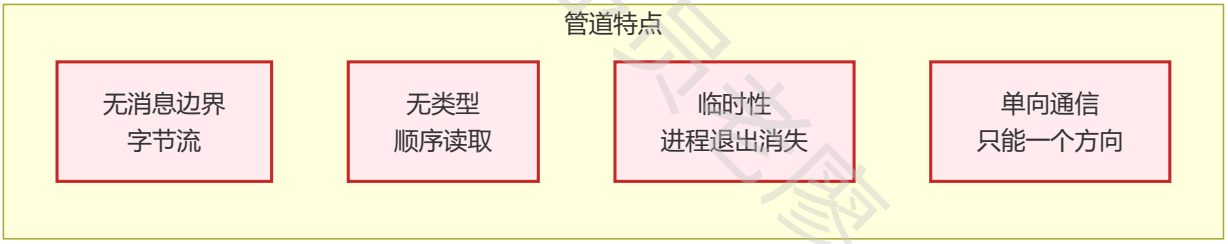
```
# 终端1
$ ./fifo_reader
打开FIFO读端...

# 终端2
$ ./fifo_writer
打开FIFO写端...
开始写入数据
已发送: Message 1
已发送: Message 2
...

# 终端1输出
开始读取数据
收到消息: Message 1
收到消息: Message 2
...
```

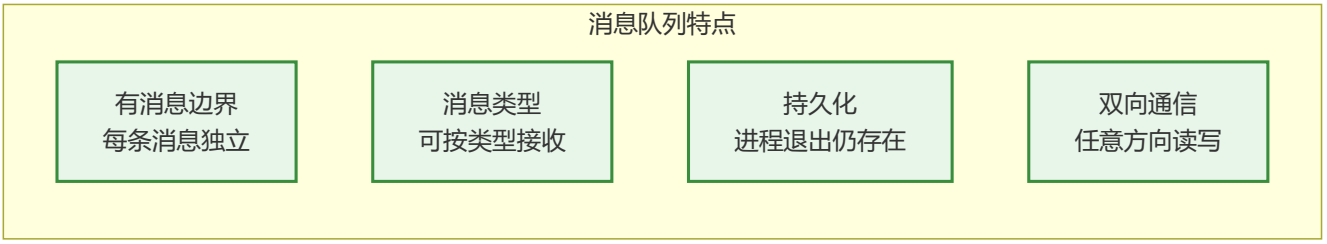
# 7.4 System V 消息队列

## 7.4.1 什么是消息队列?

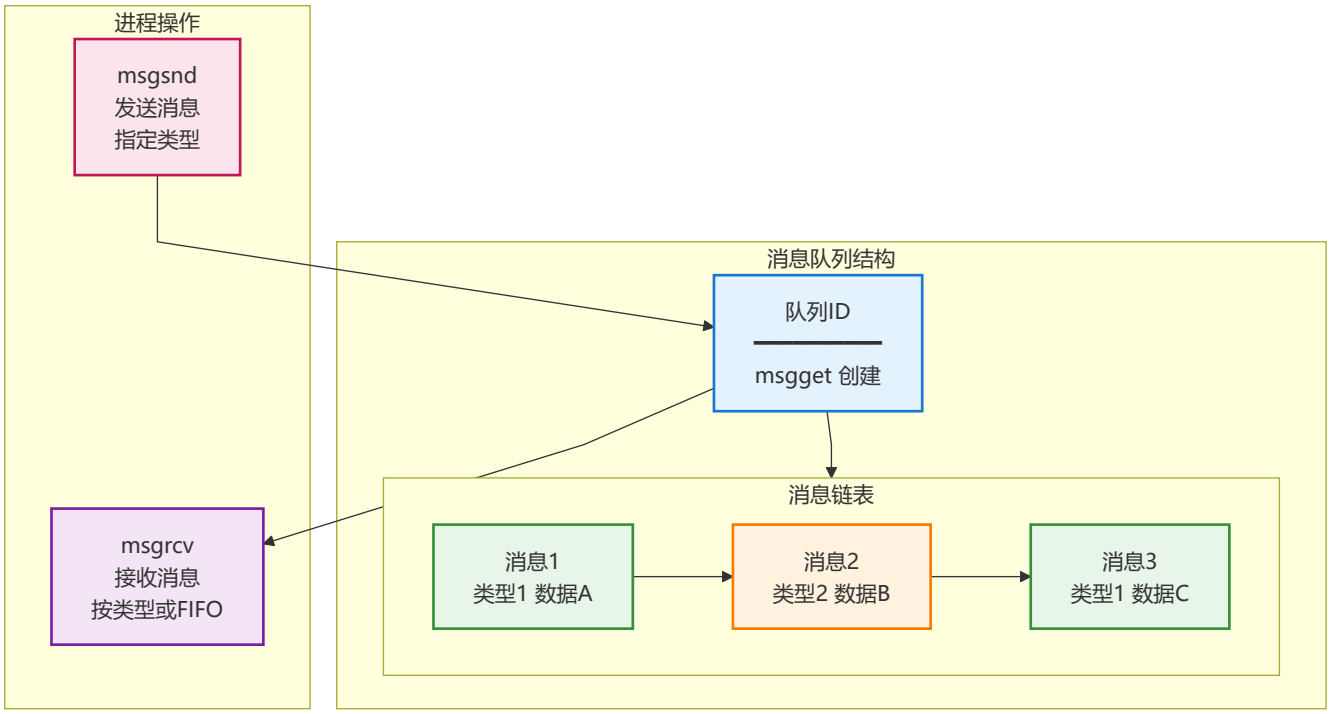
**消息队列（Message Queue）** 是内核中的消息链表，进程可以向队列发送消息，也可以从队列接收消息。

**消息队列 vs 管道**




**消息队列的结构**




## 7.4.2 创建/获取消息队列 msgget()

### API 说明

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

**功能**：创建或获取一个消息队列

**参数**：

- `key`：消息队列的键值
  - 可以是 `IPC_PRIVATE`（私有队列）
  - 或使用 `ftok()` 生成的键值
- `msgflg`：标志位
  - `IPC_CREAT`：如果不存在则创建
  - `IPC_EXCL`：如果已存在则失败（配合 `IPC_CREAT` 使用）
  - 权限位：如 `0666`

**返回值**：

- 成功返回消息队列ID（非负整数）
- 失败返回 `-1`，设置 `errno`

**生成键值 ftok()**

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(const char *pathname, int proj_id);
```

**功能**：根据文件路径和项目ID生成唯一的键值

**参数**：

- `pathname`：已存在的文件路径
- `proj_id`：项目ID（1-255）

**示例**：

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main() {
    // 方法1：使用ftok生成key
    key_t key = ftok("/tmp", 'A');
    if (key == -1) {
        perror("ftok");
        return 1;
    }
    printf("生成的key: 0x%x\n", key);

    // 方法2：创建消息队列
    int msqid = msgget(key, IPC_CREAT | 0666);
    if (msqid == -1) {
        perror("msgget");
        return 1;
    }
    printf("消息队列ID: %d\n", msqid);

    // 查看消息队列信息
    system("ipcs -q");

    return 0;
}
```



## 7.4.3 发送消息 msgsnd()

### API 说明

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

// 消息结构体
```

```
struct msgbuf {
    long mtype;       // 消息类型（必须 > 0）
    char mtext[1];    // 消息数据（可变长度）
};
```

**功能**：向消息队列发送一条消息

**参数**：

- `msqid`：消息队列ID
- `msgp`：指向消息结构体的指针
- `msgsz`：消息数据的大小（不包括`mtype`）
- `msgflg`：标志位
    - `0`：阻塞模式（队列满时阻塞）
    - `IPC_NOWAIT`：非阻塞模式（队列满时立即返回错误）

**返回值**：

- 成功返回`0`
- 失败返回`-1`，设置`errno`

**示例**：

```
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct message {
    long mtype;
    char mtext[100];
};

int main() {
    key_t key = ftok("/tmp", 'A');
    int msqid = msgget(key, IPC_CREAT | 0666);

    struct message msg;
    msg.mtype = 1;  // 消息类型1
    strcpy(msg.mtext, "Hello, Message Queue!");

    // 发送消息
    if (msgsnd(msqid, &msg, strlen(msg.mtext) + 1, 0) == -1) {
        perror("msgsnd");
        return 1;
    }

    printf("消息已发送: %s\n", msg.mtext);
    return 0;
}
```

💻 **实践代码**：`src/chapter07/msgqueue_send.c`

## 7.4.4 接收消息 msgrcv()

### API 说明

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

**功能**：从消息队列接收一条消息

**参数**：

- `msqid`：消息队列ID
- `msgp`：用于存放接收消息的缓冲区
- `msgsz`：消息数据的最大大小
- `msgtyp`：指定接收哪种类型的消息
  - `= 0`：接收队列中第一条消息（FIFO）
  - `> 0`：接收第一条类型为 `msgtyp` 的消息
  - `< 0`：接收类型 ≤ `|msgtyp|` 中最小类型的消息
- `msgflg`：标志位
  - `0`：阻塞模式（队列空时阻塞）
  - `IPC_NOWAIT`：非阻塞模式
  - `MSG_NOERROR`：消息过大时截断

**返回值**：

- 成功返回接收的消息数据字节数
- 失败返回 `-1`，设置 `errno`

**示例**：

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct message {
    long mtype;
    char mtext[100];
};

int main() {
    key_t key = ftok("/tmp", 'A');
    int msqid = msgget(key, 0666);
```

```
    struct message msg;

    // 接收类型为1的消息
    ssize_t n = msgrcv(msqid, &msg, sizeof(msg.mtext), 1, 0);
    if (n == -1) {
        perror("msgrcv");
        return 1;
    }

    printf("收到消息(类型%ld): %s\n", msg.mtype, msg.mtext);
    return 0;
}
```

💻 **实践代码**：`src/chapter07/msgqueue_recv.c`

## 7.4.5 删除消息队列 msgctl()

### API 说明

```
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

**功能**：控制消息队列

**常用命令**：

- `IPC_STAT`：获取消息队列信息
- `IPC_SET`：设置消息队列属性
- `IPC_RMID`：删除消息队列

**示例：删除消息队列**

```
#include <sys/ipc.h>
#include <sys/msg.h>

int main() {
    key_t key = ftok("/tmp", 'A');
    int msqid = msgget(key, 0666);

    // 删除消息队列
    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl");
        return 1;
    }

    printf("消息队列已删除\n");
    return 0;
}
```

**命令行管理**：

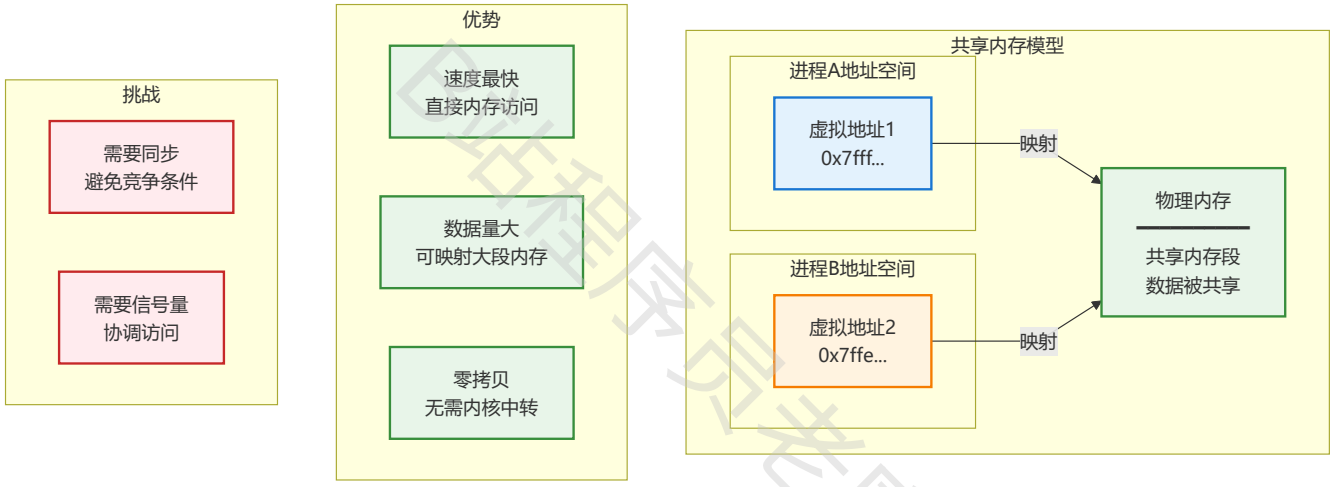
```
# 查看所有消息队列
$ ipcs -q

# 删除消息队列（msqid为队列ID）
$ ipcrm -q <msqid>

# 删除所有消息队列
$ ipcrm -a msg
```

# 7.5 System V 共享内存

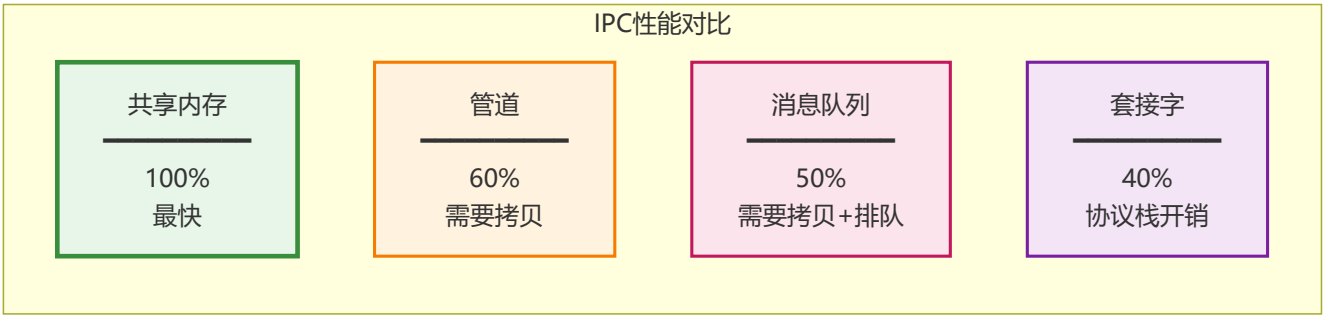
## 7.5.1 什么是共享内存?

**共享内存（Shared Memory）** 是最快的IPC方式，多个进程可以直接访问同一块物理内存。

**共享内存的特点：**




**性能对比（相对速度）**



## 7.5.2 创建/获取共享内存 shmget()

### API 说明

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

**功能**：创建或获取一个共享内存段

**参数**：

- `key`：共享内存的键值（可用`ftok`生成）
- `size`：共享内存大小（字节）
- `shmflg`：标志位
  - `IPC_CREAT`：不存在则创建
  - `IPC_EXCL`：已存在则失败
  - 权限位：如 `0666`

**返回值**：

- 成功返回共享内存ID
- 失败返回 `-1`，设置 `errno`

**示例**：

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024

int main() {
    key_t key = ftok("/tmp", 'S');

    // 创建共享内存
    int shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        return 1;
    }

    printf("共享内存ID: %d\n", shmid);
    printf("大小: %d 字节\n", SHM_SIZE);

    // 查看共享内存
    system("ipcs -m");

    return 0;
```

```
}
```

## 7.5.3 映射共享内存 shmat()

### API 说明

```
#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

**功能**：将共享内存映射到进程的地址空间

**参数**：

- `shmid`：共享内存ID
- `shmaddr`：映射地址（通常设为 `NULL`，让系统选择）
- `shmflg`：标志位
  - `0`：可读可写
  - `SHM_RDONLY`：只读

**返回值**：

- 成功返回映射地址指针
- 失败返回 `(void *) -1`，设置 `errno`

## 7.5.4 解除映射 shmdt()

### API 说明

```
#include <sys/types.h>
#include <sys/shm.h>

int shmdt(const void *shmaddr);
```

**功能**：解除共享内存的映射

**参数**：

- `shmaddr`：之前 `shmat` 返回的地址

**返回值**：

- 成功返回 `0`
- 失败返回 `-1`，设置 `errno`

## 7.5.5 删除共享内存 shmctl()

### API 说明

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

**功能**：控制共享内存

**常用命令**：

- `IPC_STAT`：获取共享内存信息
- `IPC_SET`：设置共享内存属性
- `IPC_RMID`：删除共享内存

**完整示例：共享内存通信**

写进程（`shm_writer.c`）：



```
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>

#define SHM_SIZE 1024

int main() {
    key_t key = ftok("/tmp", 'S');
    int shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666);

    // 映射共享内存
    char *shmaddr = (char *)shmat(shmid, NULL, 0);
    if (shmaddr == (char *) -1) {
        perror("shmat");
        return 1;
    }

    printf("共享内存已映射到地址: %p\n", shmaddr);

    // 写入数据
    for (int i = 1; i <= 5; i++) {
        snprintf(shmaddr, SHM_SIZE, "Message %d from writer", i);
        printf("已写入: %s\n", shmaddr);
        sleep(2);
    }

    // 解除映射
    shmdt(shmaddr);
```

```
    printf("写进程结束\n");
    return 0;
}
```

读进程（`shm_reader.c`）：

```c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>

#define SHM_SIZE 1024

int main() {
    key_t key = ftok("/tmp", 'S');
    int shmid = shmget(key, SHM_SIZE, 0666);

    if (shmid == -1) {
        perror("shmget");
        return 1;
    }

    // 映射共享内存
    char *shmaddr = (char *)shmat(shmid, NULL, 0);
    if (shmaddr == (char *) -1) {
        perror("shmat");
        return 1;
    }

    printf("共享内存已映射到地址: %p\n", shmaddr);

    // 读取数据
    for (int i = 0; i < 5; i++) {
        printf("当前内容: %s\n", shmaddr);
        sleep(2);
    }

    // 解除映射
    shmdt(shmaddr);

    // 删除共享内存
    shmctl(shmid, IPC_RMID, NULL);
    printf("共享内存已删除\n");

    return 0;
}
```



💻 **实践代码**：`src/chapter07/shm_writer.c`，`src/chapter07/shm_reader.c`

**运行方式**：

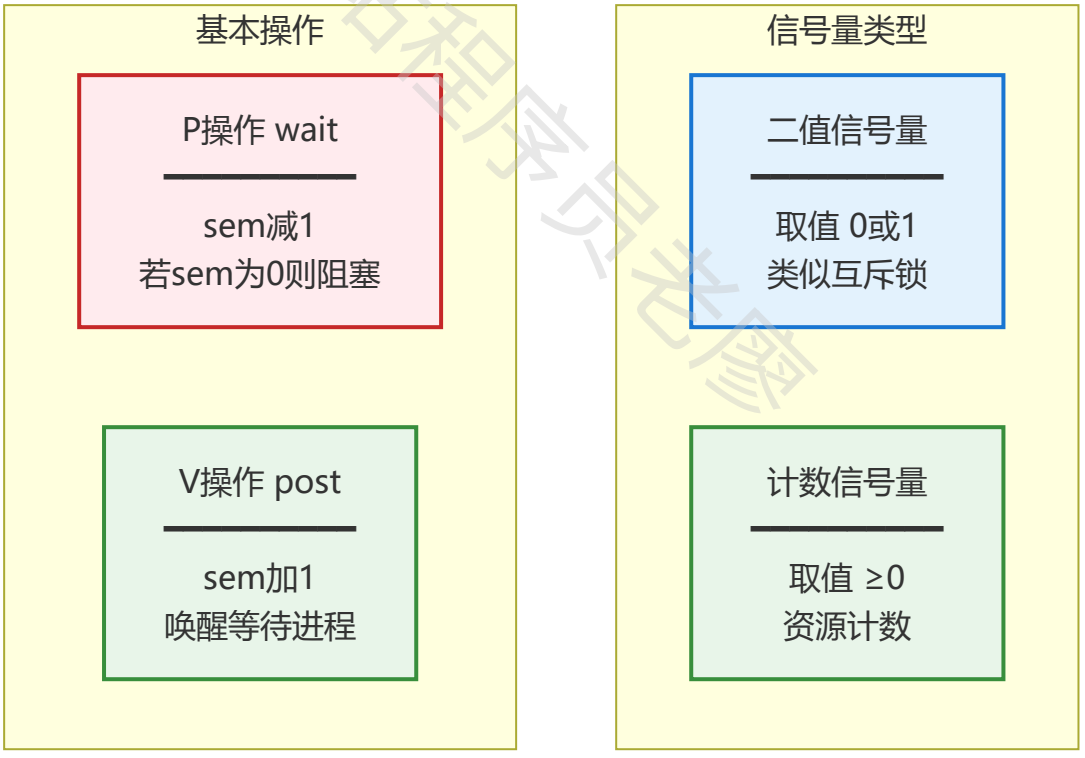
```
# 终端1
$ ./shm_reader
共享内存已映射到地址: 0x7f...
当前内容:
当前内容: Message 1 from writer
...

# 终端2
$ ./shm_writer
共享内存已映射到地址: 0x7f...
已写入: Message 1 from writer
...
```

# 7.6 System V 信号量

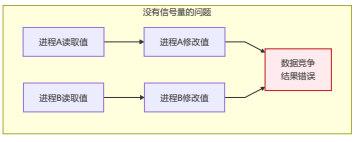
## 7.6.1 什么是信号量?

**信号量（Semaphore）** 是用于进程间同步的计数器，主要用于保护共享资源。

**信号量的类型:**




**信号量解决的问题:**

## 7.6.2 创建信号量集 semget()

### API 说明

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

**功能**：创建或获取一个信号量集

**参数**：

- `key`：信号量的键值
- `nsems`：信号量集中信号量的个数
- `semflg`：标志位（`IPC_CREAT | 0666`）

**返回值**：

- 成功返回信号量集ID
- 失败返回 `-1`

## 7.6.3 操作信号量 semop()

### API 说明

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);

struct sembuf {
    unsigned short sem_num;  // 信号量编号
    short sem_op;            // 操作：-1(P), +1(V), 0(等待为0)
    short sem_flg;           // 标志：SEM_UNDO, IPC_NOWAIT
};
```

**功能**：对信号量进行操作

**参数**：

- `semid`：信号量集ID
- `sops`：操作数组
- `nsops`：操作数量

**sem_op 的含义**：

- `> 0`：V操作，增加信号量值
- `< 0`：P操作，减少信号量值（可能阻塞）

- `= 0`：等待信号量值变为0

## 7.6.4 控制信号量 semctl()

### API 说明

```
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

**常用命令**：

- `SETVAL`：设置信号量的值
- `GETVAL`：获取信号量的值
- `IPC_RMID`：删除信号量集

**完整示例：使用信号量保护共享内存**

💻 **实践代码**：`src/chapter07/sem_shm_demo.c`



```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <unistd.h>
#include <string.h>

#define SHM_SIZE 100

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};

// P操作（等待）
void sem_wait(int semid) {
    struct sembuf sb = {0, -1, SEM_UNDO};
    semop(semid, &sb, 1);
}

// V操作（信号）
void sem_signal(int semid) {
    struct sembuf sb = {0, 1, SEM_UNDO};
    semop(semid, &sb, 1);
}

int main() {
    key_t key = ftok("/tmp", 'X');
```

```
    // 创建共享内存
    int shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666);
    int *shmaddr = (int *)shmat(shmid, NULL, 0);

    // 创建信号量
    int semid = semget(key, 1, IPC_CREAT | 0666);
    union semun arg;
    arg.val = 1;  // 初始化为1（二值信号量）
    semctl(semid, 0, SETVAL, arg);

    pid_t pid = fork();

    if (pid == 0) {
        // 子进程
        for (int i = 0; i < 5; i++) {
            sem_wait(semid);  // 获取锁

            (*shmaddr)++;
            printf("子进程: counter = %d\n", *shmaddr);

            sem_signal(semid);  // 释放锁
            sleep(1);
        }
    } else {
        // 父进程
        for (int i = 0; i < 5; i++) {
            sem_wait(semid);  // 获取锁

            (*shmaddr)++;
            printf("父进程: counter = %d\n", *shmaddr);

            sem_signal(semid);  // 释放锁
            sleep(1);
        }

        wait(NULL);

        printf("最终值: %d（预期10）\n", *shmaddr);

        // 清理
        shmdt(shmaddr);
        shmctl(shmid, IPC_RMID, NULL);
        semctl(semid, 0, IPC_RMID);
    }

    return 0;
}
```

# 7.7 API快速参考

## 7.7.1 管道相关

| 函数 | 功能 | 返回值 |
| --- | --- | --- |
| `pipe(pipefd)` | 创建无名管道 | 0成功，-1失败 |
| `mkfifo(path, mode)` | 创建命名管道 | 0成功，-1失败 |
| `read/write` | 读写管道 | 字节数或-1 |

## 7.7.2 消息队列相关

| 函数 | 功能 | 返回值 |
| --- | --- | --- |
| `msgget(key, flag)` | 创建/获取消息队列 | 队列ID或-1 |
| `msgsnd(id, msg, size, flag)` | 发送消息 | 0成功，-1失败 |
| `msgrcv(id, msg, size, type, flag)` | 接收消息 | 字节数或-1 |
| `msgctl(id, cmd, buf)` | 控制/删除消息队列 | 0成功，-1失败 |

## 7.7.3 共享内存相关

| 函数 | 功能 | 返回值 |
| --- | --- | --- |
| `shmget(key, size, flag)` | 创建/获取共享内存 | 共享内存ID或-1 |
| `shmat(id, addr, flag)` | 映射共享内存 | 地址或(void*)-1 |
| `shmdt(addr)` | 解除映射 | 0成功，-1失败 |
| `shmctl(id, cmd, buf)` | 控制/删除共享内存 | 0成功，-1失败 |

## 7.7.4 信号量相关

| 函数 | 功能 | 返回值 |
| --- | --- | --- |
| `semget(key, num, flag)` | 创建/获取信号量集 | 信号量ID或-1 |
| `semop(id, ops, num)` | 操作信号量（P/V） | 0成功，-1失败 |
| `semctl(id, num, cmd, arg)` | 控制/删除信号量 | 取决于cmd |

# 7.8 小结

本章介绍了Linux进程间通信（IPC）的主要机制：

**核心知识点：**

1. **管道（Pipe）**：
    - 无名管道：父子进程通信，简单高效
    - 命名管道：无亲缘关系进程通信
    - 单向、字节流、阻塞式
2. **消息队列（Message Queue）**：
    - 有消息边界，支持类型
    - 双向通信，持久化
    - 适合多进程消息传递
3. **共享内存（Shared Memory）**：
    - 最快的IPC方式，零拷贝
    - 需要配合信号量同步
    - 适合大数据共享
4. **信号量（Semaphore）**：
    - 用于进程间同步
    - P/V操作保护临界区
    - 常与共享内存配合使用

**IPC选择指南：**

- 父子进程，数据量小 → **无名管道**
- 无亲缘关系，单向通信 → **命名管道**
- 多进程消息传递 → **消息队列**
- 大数据共享，性能要求高 → **共享内存 + 信号量**

**学习建议：**

1. 先掌握管道（最简单）
2. 理解共享内存的同步问题
3. 实践各种IPC的适用场景
4. 对比不同IPC的性能特点



---

# 7.9 练习题

1. **基础练习**：
    - 编写程序，父进程通过管道向子进程发送数据，子进程反转字符串后返回
    - 使用FIFO实现一个简单的聊天程序
2. **进阶练习**：
    - 使用消息队列实现生产者-消费者模型
    - 使用共享内存和信号量实现多进程计数器

3. **实战练习**：
    - 实现一个多进程任务调度系统（消息队列）
    - 实现一个多进程共享缓存（共享内存+信号量）

---

💡 **提示**：IPC是多进程编程的基础，理解各种机制的优缺点和适用场景非常重要。实际项目中，往往需要组合使用多种IPC机制。

# 第08章 线程编程

## 引言：为什么需要线程?

在现代应用中，线程是实现并发的重要手段，比进程更轻量、更高效。

**场景1：Web服务器处理并发请求**

每个请求创建一个线程，共享服务器配置和缓存：

```
主线程（监听端口）
├── 工作线程1（处理请求1）
├── 工作线程2（处理请求2）
└── 工作线程3（处理请求3）
    └ 共享：配置数据、数据库连接池、缓存
```

**场景2：GUI应用的响应性**

主线程处理界面，后台线程执行耗时任务：

```
UI线程（保持界面响应）
后台线程1（下载文件）→ 通知UI更新进度
后台线程2（数据处理）→ 通知UI显示结果
```

**场景3：数据处理的并行计算**

多个线程并行处理数据，充分利用多核CPU：

```
数据集（100万条记录）
├── 线程1 处理 0-25万
├── 线程2 处理 25-50万
├── 线程3 处理 50-75万
└── 线程4 处理 75-100万
```

**本章解决的问题：**

- ✅ 什么是线程？线程 vs 进程?
- ✅ 如何创建和管理线程?
- ✅ 线程间如何同步和通信?
- ✅ 如何避免数据竞争和死锁?
- ✅ 线程的最佳实践?

**学完本章你能做什么：**

- 实现多线程服务器
- 编写并行计算程序
- 正确使用互斥锁和条件变量
- 理解并发编程的常见陷阱

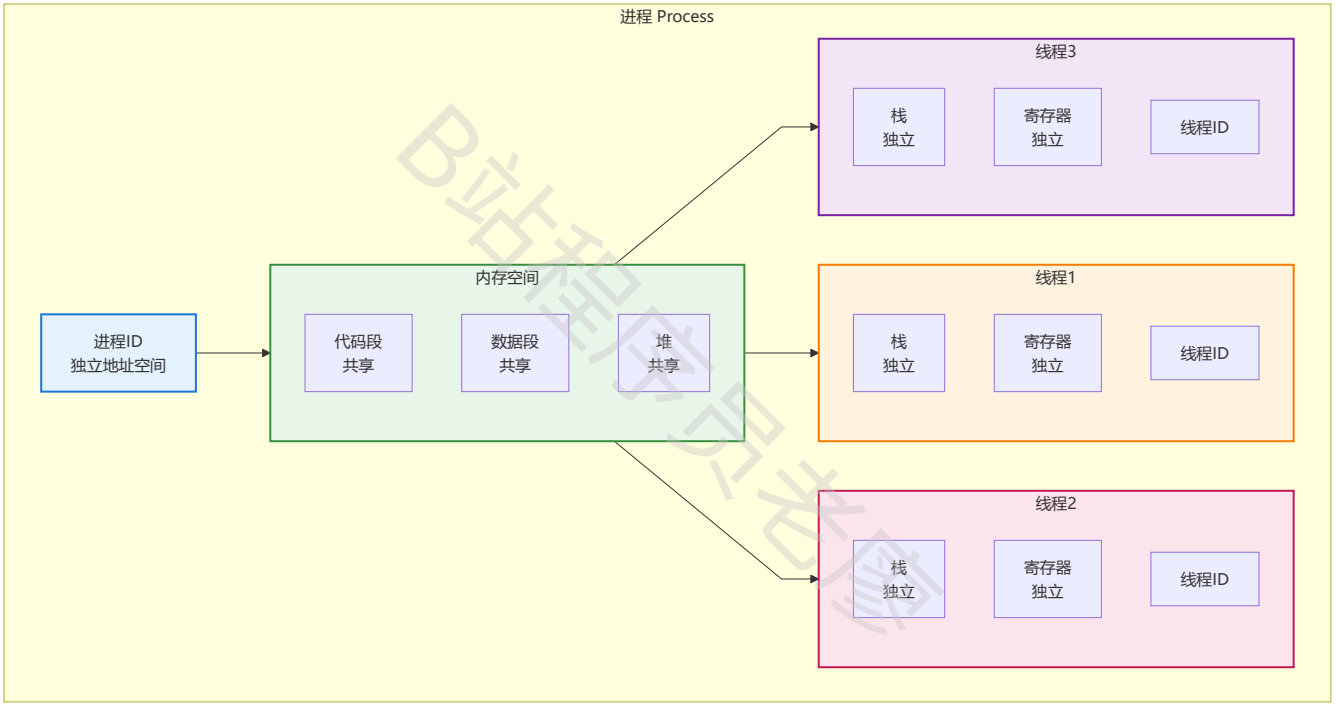
---

# 8.1 线程的基本概念

## 8.1.1 什么是线程?

**线程（Thread）** 是进程内的执行单元，一个进程可以包含多个线程。

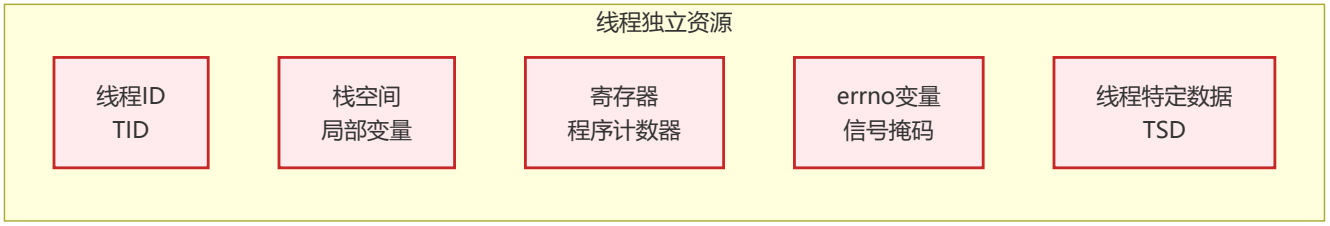
**进程 vs 线程：**

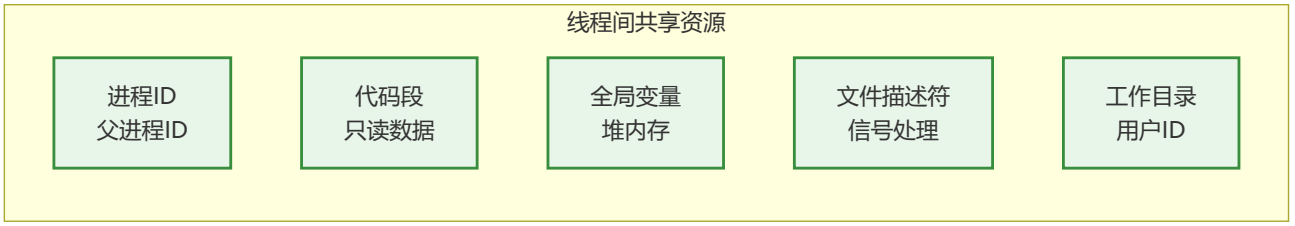



**关键区别：**

| 特性 | 进程 | 线程 |
| --- | --- | --- |
| **地址空间** | 独立 | 共享进程地址空间 |
| **数据共享** | 困难（需要IPC） | 容易（共享全局变量） |
| **创建开销** | 大（复制地址空间） | 小（只创建栈和寄存器） |
| **切换开销** | 大（切换地址空间） | 小（只切换栈和寄存器） |
| **通信方式** | IPC（管道、共享内存等） | 直接读写内存 |
| **健壮性** | 高（一个进程崩溃不影响其他） | 低（一个线程崩溃导致进程退出） |

### 8.1.2 线程的共享与独立资源




### 8.1.3 POSIX线程（pthread）

Linux使用POSIX线程标准（pthread），所有线程API都以 `pthread_` 开头。

**编译时需要链接pthread库：**

```
gcc program.c -o program -lpthread
```

## 8.2 创建和终止线程

### 8.2.1 创建线程 pthread_create()

#### API 说明

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

**功能**：创建一个新线程

**参数**：

- `thread`：用于存储新线程的ID
- `attr`：线程属性（通常为 `NULL`，使用默认属性）
- `start_routine`：线程执行的函数（函数指针）
    - 函数签名：`void *func(void *arg)`
- `arg`：传递给线程函数的参数

**返回值**：

- 成功返回 `0`

- 失败返回错误码（注意：不设置 `errno`）

**线程函数的要求：**

```
// 线程函数必须是这个签名
void *thread_function(void *arg) {
    // 线程执行的代码

    // 返回值可以被 pthread_join 获取
    return NULL;  // 或返回其他指针
}
```

**示例：基本用法**

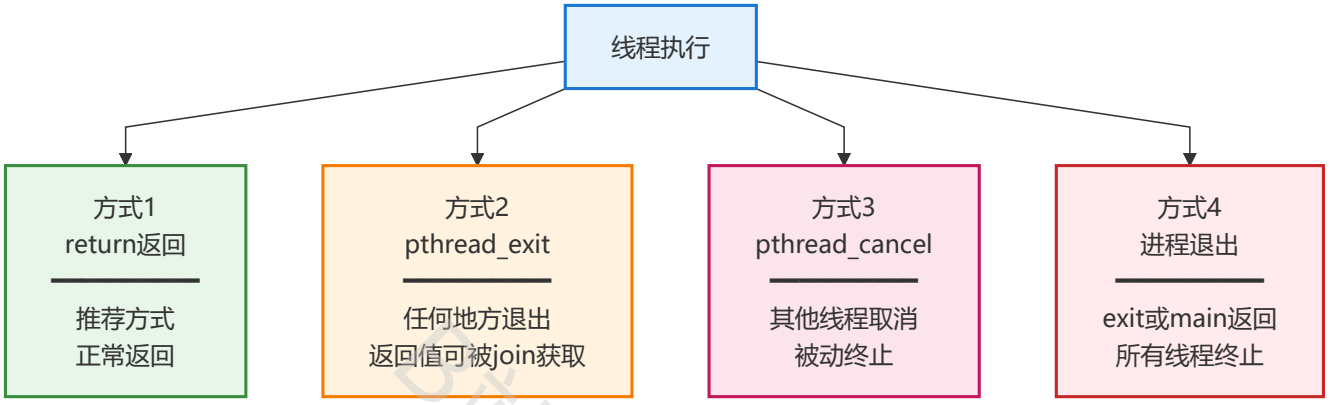
```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

void *thread_func(void *arg) {
    int id = *(int *)arg;
    printf("线程 %d 启动\n", id);

    for (int i = 0; i < 5; i++) {
        printf("线程 %d: 计数 %d\n", id, i);
        sleep(1);
    }

    printf("线程 %d 结束\n", id);
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    int id1 = 1, id2 = 2;

    // 创建线程1
    if (pthread_create(&thread1, NULL, thread_func, &id1) != 0) {
        perror("pthread_create");
        return 1;
    }

    // 创建线程2
    if (pthread_create(&thread2, NULL, thread_func, &id2) != 0) {
        perror("pthread_create");
        return 1;
    }

    printf("主线程：已创建两个线程\n");

    // 等待线程结束
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
```

```
    printf("主线程：所有线程已结束\n");
    return 0;
}
```

💻 **实践代码**：`src/chapter08/thread_basic.c`

## 8.2.2 终止线程

线程可以通过以下方式终止：




### pthread_exit()

```
#include <pthread.h>

void pthread_exit(void *retval);
```

**功能**：终止调用线程

**参数**：

- `retval`：返回值（可被 `pthread_join` 获取）

**特点**：

- 可以在线程函数的任何地方调用
- 等价于在线程函数中 `return retval`
- 不影响其他线程

**示例**：

```c
void *thread_func(void *arg) {
    int *result = malloc(sizeof(int));
    *result = 42;

    // 提前退出线程
    pthread_exit(result);

    // 这里的代码不会执行
    printf("不会打印\n");
    return NULL;
}
```

## 8.2.3 等待线程 pthread_join()

### API 说明

```c
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

**功能**：等待指定线程结束，并获取其返回值

**参数**：

- `thread`：要等待的线程ID
- `retval`：用于存储线程返回值的指针（可为 `NULL`）

**返回值**：

- 成功返回 `0`
- 失败返回错误码

**特点**：

- 阻塞调用线程，直到目标线程结束
- 类似于进程的 `wait()`
- 只能被调用一次（线程资源会被回收）

**示例：获取返回值**

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *compute(void *arg) {
    int n = *(int *)arg;
    int *result = malloc(sizeof(int));

    // 计算平方
    *result = n * n;

    return result;
```

```c
}

int main() {
    pthread_t thread;
    int input = 5;
    int *output;

    pthread_create(&thread, NULL, compute, &input);

    // 等待并获取返回值
    pthread_join(thread, (void **)&output);

    printf("输入: %d, 输出: %d\n", input, *output);

    free(output);
    return 0;
}
```

💻 **实践代码**：`src/chapter08/thread_join.c`

## 8.2.4 分离线程 pthread_detach()

### API 说明

```c
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

**功能**：将线程设置为分离状态（detached）

**特点**：

- 分离线程结束后自动释放资源
- 不能被 `pthread_join` 等待
- 适合"发射后不管"的线程

**joinable vs detached：**

| 特性 | Joinable（默认） | Detached（分离） |
| --- | --- | --- |
| 资源回收 | 需要pthread_join回收 | 自动回收 |
| 可被join | 是 | 否 |
| 返回值 | 可获取 | 无法获取 |
| 用途 | 需要获取结果的线程 | 独立运行的后台线程 |

**示例**：

```c
#include <stdio.h>
#include <pthread.h>
```

```c
#include <unistd.h>

void *background_task(void *arg) {
    printf("后台任务启动\n");
    sleep(2);
    printf("后台任务完成\n");
    return NULL;
}

int main() {
    pthread_t thread;

    pthread_create(&thread, NULL, background_task, NULL);

    // 设置为分离状态
    pthread_detach(thread);

    printf("主线程继续执行，不需要等待后台任务\n");

    // 给后台线程一些时间完成
    sleep(3);

    return 0;
}
```

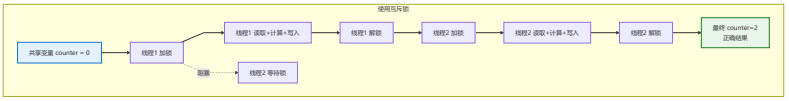
💻 **实践代码**：`src/chapter08/thread_detach.c`



# 8.3 线程同步：互斥锁

## 8.3.1 为什么需要同步?

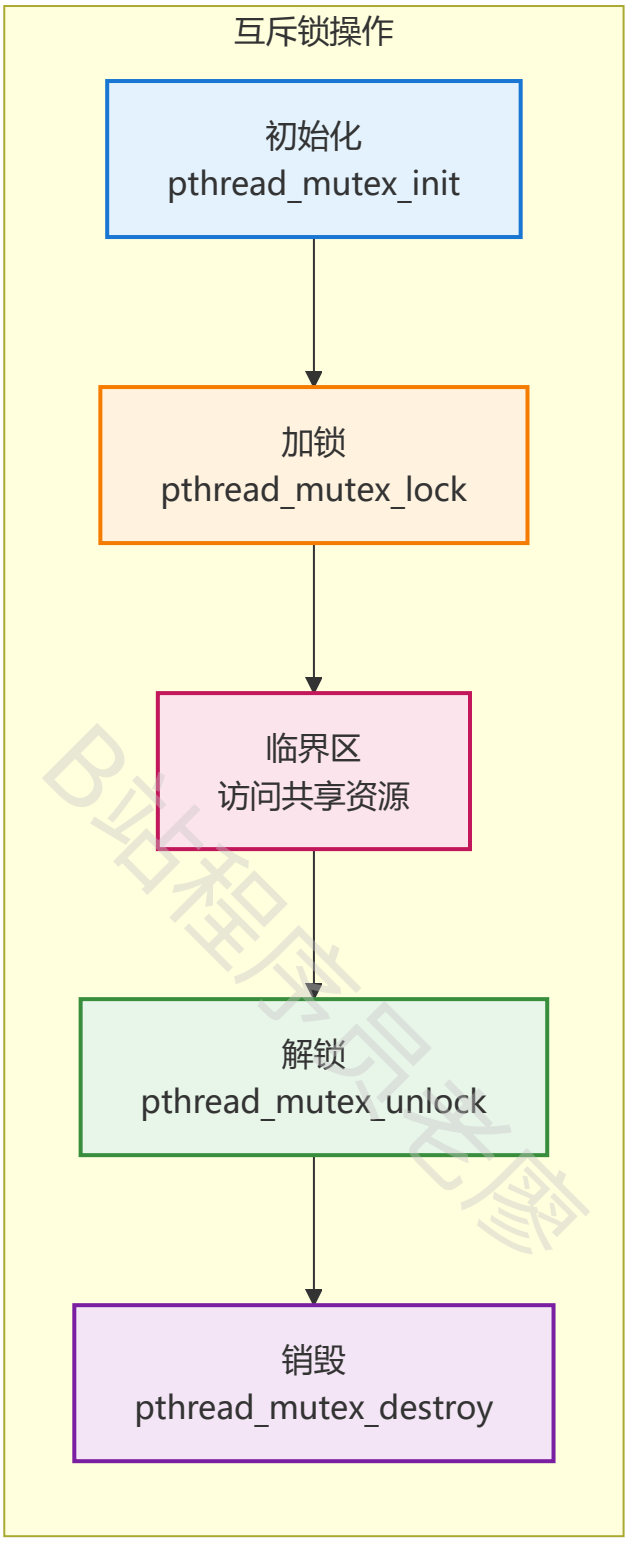
多个线程同时访问共享资源会导致**数据竞争（Race Condition）**：





## 8.3.2 互斥锁（Mutex）

**互斥锁（Mutual Exclusion Lock）** 保证同一时刻只有一个线程访问共享资源。


互斥锁操作
初始化
pthread_mutex_init
加锁
pthread_mutex_lock
临界区
访问共享资源
解锁
pthread_mutex_unlock
销毁
pthread_mutex_destroy

### 初始化和销毁

```c
#include <pthread.h>

// 静态初始化（推荐用于全局变量）
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// 动态初始化
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

// 销毁
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

### 加锁和解锁

```c
// 加锁（阻塞）
int pthread_mutex_lock(pthread_mutex_t *mutex);

// 尝试加锁（非阻塞）
int pthread_mutex_trylock(pthread_mutex_t *mutex);

// 解锁
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

#### 完整示例：计数器保护

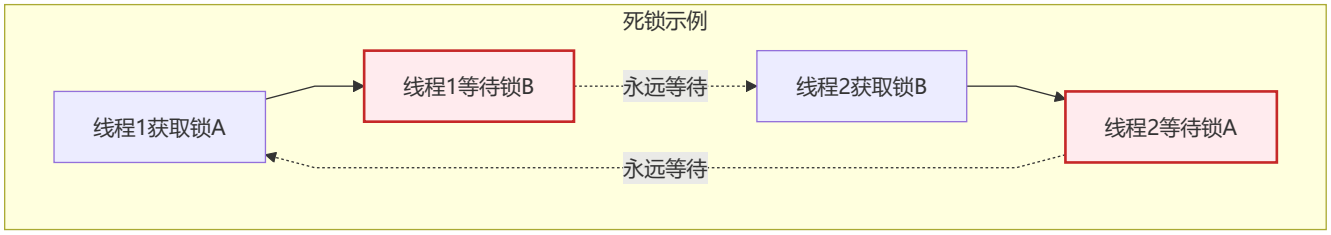
```c
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

void *increment(void *arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&mutex);    // 加锁
        counter++;                     // 临界区
        pthread_mutex_unlock(&mutex);  // 解锁
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("计数器最终值: %d\n", counter);
```

```
    printf("预期值: 200000\n");

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

💻 **实践代码**：`src/chapter08/mutex_basic.c`

## 8.3.3 死锁及其避免

**死锁（Deadlock）** 是指两个或多个线程相互等待对方释放锁，导致永久阻塞。




**死锁的四个必要条件：**

1. **互斥**：资源同时只能被一个线程使用
2. **持有并等待**：持有锁的同时等待其他锁
3. **不可剥夺**：锁不能被强制释放
4. **循环等待**：形成锁的循环依赖

**避免死锁的方法：**

```
// ❌ 错误：可能死锁
void *thread1(void *arg) {
    pthread_mutex_lock(&mutex_a);
    pthread_mutex_lock(&mutex_b);  // 等待mutex_b
    // ...
    pthread_mutex_unlock(&mutex_b);
    pthread_mutex_unlock(&mutex_a);
}

void *thread2(void *arg) {
    pthread_mutex_lock(&mutex_b);
    pthread_mutex_lock(&mutex_a);  // 等待mutex_a，死锁！
    // ...
    pthread_mutex_unlock(&mutex_a);
    pthread_mutex_unlock(&mutex_b);
}

// ✅ 正确：统一加锁顺序
void *thread1(void *arg) {
    pthread_mutex_lock(&mutex_a);  // 总是先A后B
    pthread_mutex_lock(&mutex_b);
    // ...
    pthread_mutex_unlock(&mutex_b);
```

```
    pthread_mutex_unlock(&mutex_a);
}

void *thread2(void *arg) {
    pthread_mutex_lock(&mutex_a);  // 总是先A后B
    pthread_mutex_lock(&mutex_b);
    // ...
    pthread_mutex_unlock(&mutex_b);
    pthread_mutex_unlock(&mutex_a);
}
```

💻 **实践代码**：`src/chapter08/deadlock_demo.c`

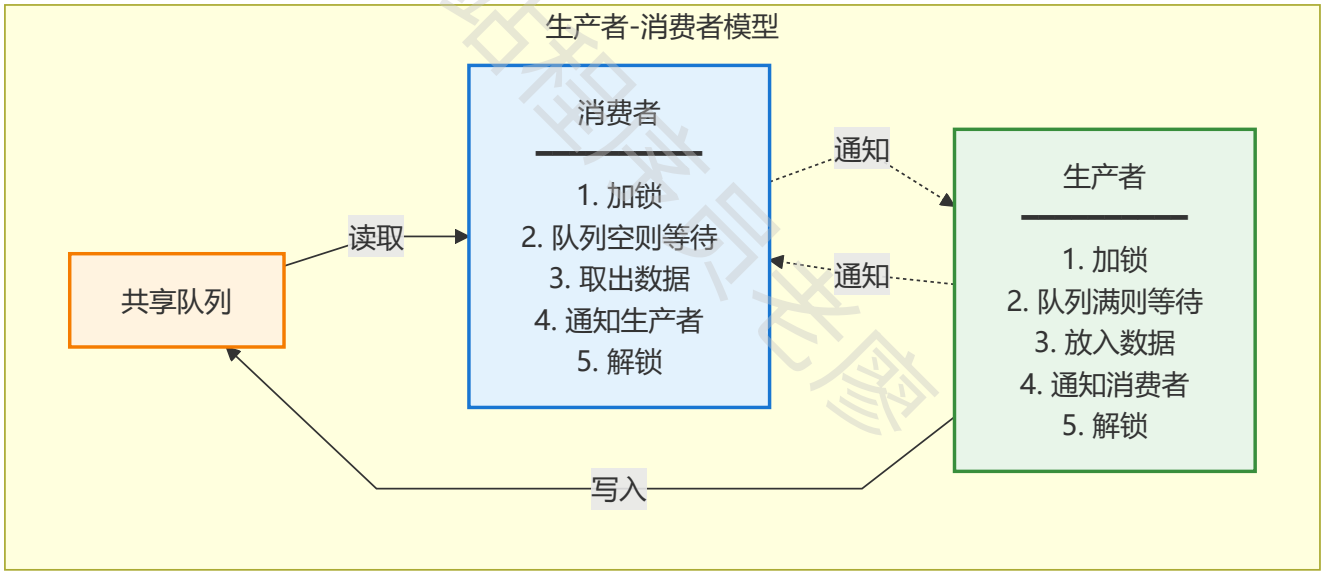
---

# 8.4 线程同步：条件变量

## 8.4.1 什么是条件变量?

**条件变量（Condition Variable）** 用于线程间等待某个条件成立，避免轮询浪费CPU。

**场景：生产者-消费者模型**




**条件变量 vs 轮询：**

```
// ❌ 轮询方式（浪费CPU）
while (queue_is_empty()) {
    pthread_mutex_unlock(&mutex);
    usleep(1000);  // 休眠一会
    pthread_mutex_lock(&mutex);
}

// ✅ 条件变量方式（高效）
while (queue_is_empty()) {
    pthread_cond_wait(&cond, &mutex);  // 等待信号，自动释放和重新获取锁
}
```

## 8.4.2 条件变量API

### 初始化和销毁

```
#include <pthread.h>

// 静态初始化
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// 动态初始化
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

// 销毁
int pthread_cond_destroy(pthread_cond_t *cond);
```

### 等待条件

```
// 等待条件成立（自动释放mutex，被唤醒后重新获取）
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

// 带超时的等待
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

**关键点**:

- 调用 `pthread_cond_wait` 前必须持有锁
- `pthread_cond_wait` 会原子地释放锁并进入等待
- 被唤醒后会重新获取锁

### 通知/唤醒

```
// 唤醒一个等待的线程
int pthread_cond_signal(pthread_cond_t *cond);

// 唤醒所有等待的线程
int pthread_cond_broadcast(pthread_cond_t *cond);
```

## 8.4.3 完整示例：生产者-消费者

💻 **实践代码**: `src/chapter08/producer_consumer.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
```

```c
int count = 0;  // 缓冲区中的数据个数

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;    // 缓冲区不满
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;  // 缓冲区不空

void *producer(void *arg) {
    int id = *(int *)arg;

    for (int i = 0; i < 10; i++) {
        pthread_mutex_lock(&mutex);

        // 等待缓冲区不满
        while (count == BUFFER_SIZE) {
            printf("生产者%d: 缓冲区满，等待...\n", id);
            pthread_cond_wait(&not_full, &mutex);
        }

        // 生产数据
        buffer[count] = i;
        count++;
        printf("生产者%d: 生产数据 %d，缓冲区数量=%d\n", id, i, count);

        // 通知消费者
        pthread_cond_signal(&not_empty);

        pthread_mutex_unlock(&mutex);
        usleep(100000);
    }

    return NULL;
}

void *consumer(void *arg) {
    int id = *(int *)arg;

    for (int i = 0; i < 10; i++) {
        pthread_mutex_lock(&mutex);

        // 等待缓冲区不空
        while (count == 0) {
            printf("消费者%d: 缓冲区空，等待...\n", id);
            pthread_cond_wait(&not_empty, &mutex);
        }

        // 消费数据
        count--;
        int data = buffer[count];
        printf("消费者%d: 消费数据 %d，缓冲区数量=%d\n", id, data, count);

        // 通知生产者
        pthread_cond_signal(&not_full);
```

```
        pthread_mutex_unlock(&mutex);
        usleep(150000);
    }

    return NULL;
}

int main() {
    pthread_t prod1, prod2, cons1, cons2;
    int id1 = 1, id2 = 2;

    pthread_create(&prod1, NULL, producer, &id1);
    pthread_create(&prod2, NULL, producer, &id2);
    pthread_create(&cons1, NULL, consumer, &id1);
    pthread_create(&cons2, NULL, consumer, &id2);

    pthread_join(prod1, NULL);
    pthread_join(prod2, NULL);
    pthread_join(cons1, NULL);
    pthread_join(cons2, NULL);

    printf("所有线程结束\n");

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&not_full);
    pthread_cond_destroy(&not_empty);

    return 0;
}
```



**关键点**：

1. **使用while而不是if判断条件**：

   ```
   // ❌ 错误
   if (count == 0) {
       pthread_cond_wait(&not_empty, &mutex);
   }

   // ✅ 正确
   while (count == 0) {
       pthread_cond_wait(&not_empty, &mutex);
   }
   ```

   原因：被唤醒后条件可能已被其他线程改变（虚假唤醒）

2. **持有锁时调用wait和signal**

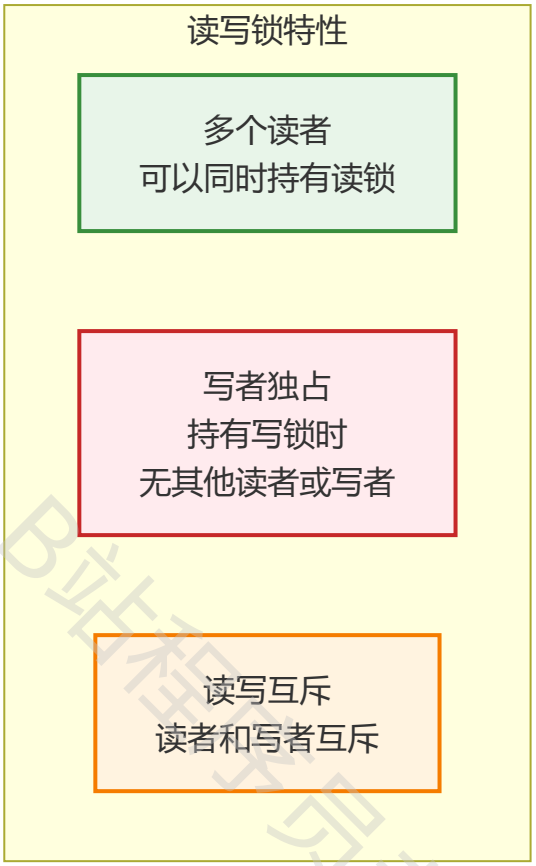
3. **signal在unlock之前或之后都可以**

---

# 8.5 其他同步机制

## 8.5.1 读写锁（Read-Write Lock）

**读写锁** 允许多个读者同时访问，但写者独占访问。






**API：**

```c
#include <pthread.h>

pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;

// 初始化和销毁
int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

// 读锁
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);    // 获取读锁
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock); // 尝试获取读锁

// 写锁
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);    // 获取写锁
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock); // 尝试获取写锁

// 解锁
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

**示例**：

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
int shared_data = 0;

void *reader(void *arg) {
    pthread_rwlock_rdlock(&rwlock);  // 获取读锁
    printf("读取数据: %d\n", shared_data);
    pthread_rwlock_unlock(&rwlock);
    return NULL;
}

void *writer(void *arg) {
    pthread_rwlock_wrlock(&rwlock);  // 获取写锁
    shared_data++;
    printf("写入数据: %d\n", shared_data);
    pthread_rwlock_unlock(&rwlock);
    return NULL;
}
```

💻 **实践代码**：`src/chapter08/rwlock_demo.c`



## 8.5.2 自旋锁（Spin Lock）

**自旋锁** 在获取锁失败时忙等待（spin），而不是休眠。

**特点**：

- 适合持锁时间极短的场景
- 避免线程切换开销
- 但会浪费CPU（忙等待）

**API**：

```
#include <pthread.h>

pthread_spinlock_t spinlock;

int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy(pthread_spinlock_t *lock);

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

**互斥锁 vs 自旋锁**：

| 特性 | 互斥锁 | 自旋锁 |
| --- | --- | --- |
| 获取失败时 | 休眠等待 | 忙等待（循环检测） |
| CPU使用 | 不占用 | 持续占用 |

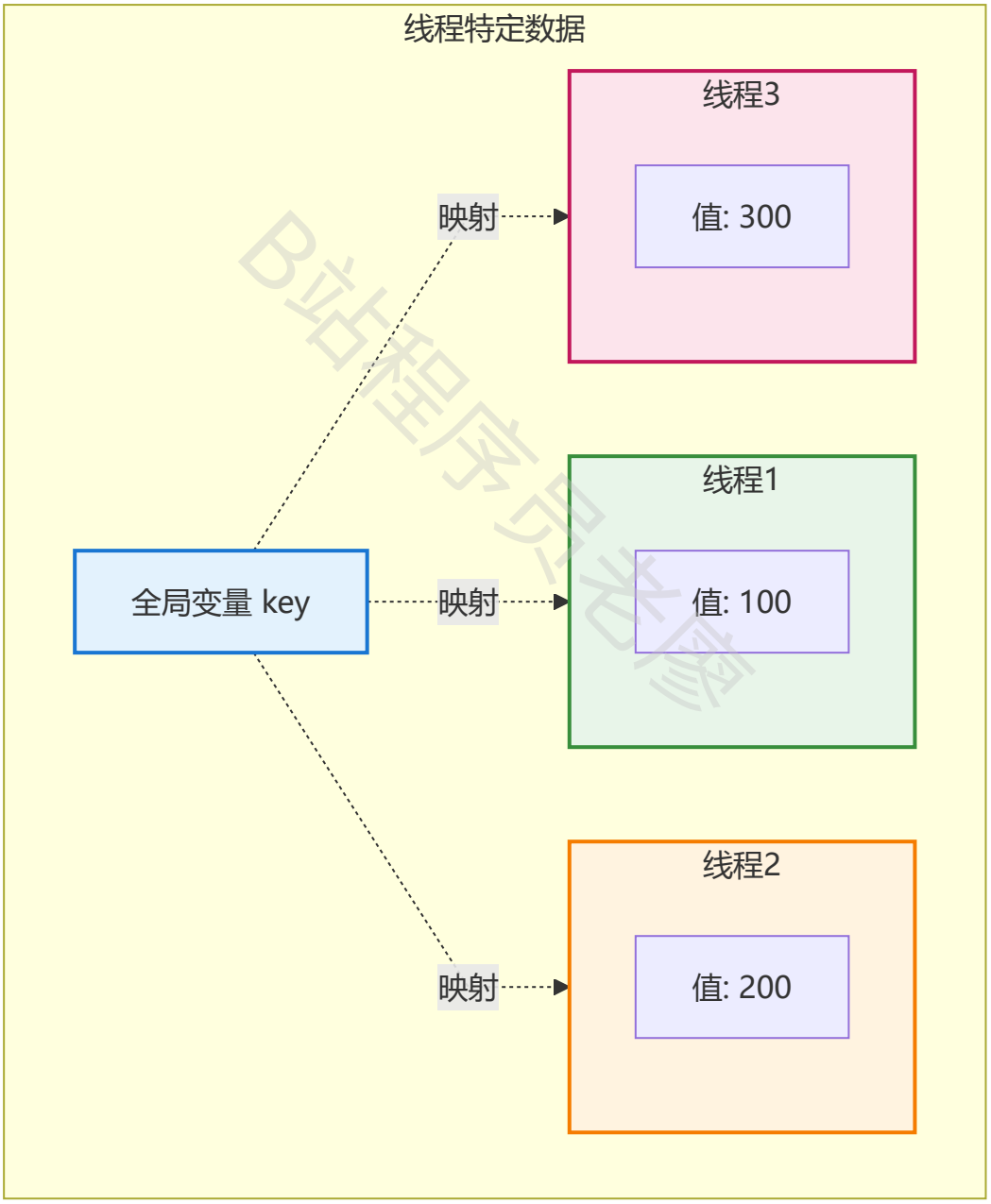
| 特性 | 互斥锁 | 自旋锁 |
| --- | --- | --- |
| 适用场景 | 持锁时间长 | 持锁时间极短 |
| 线程切换 | 有 | 无 |

---

# 8.6 线程特定数据（TSD）

## 8.6.1 什么是线程特定数据?

**线程特定数据（Thread-Specific Data, TSD）** 也叫线程局部存储（TLS），让每个线程拥有变量的独立副本。




**API：**

```
#include <pthread.h>
```

```
pthread_key_t key;

// 创建key（所有线程共享同一个key）
int pthread_key_create(pthread_key_t *key, void (*destructor)(void *));

// 删除key
int pthread_key_delete(pthread_key_t key);

// 设置当前线程的值
int pthread_setspecific(pthread_key_t key, const void *value);

// 获取当前线程的值
void *pthread_getspecific(pthread_key_t key);
```

**示例**：

```
pthread_key_t key;

void *thread_func(void *arg) {
    int id = *(int *)arg;

    // 为当前线程设置值
    int *value = malloc(sizeof(int));
    *value = id * 100;
    pthread_setspecific(key, value);

    // 获取当前线程的值
    int *retrieved = pthread_getspecific(key);
    printf("线程%d: 值=%d\n", id, *retrieved);

    return NULL;
}

int main() {
    pthread_key_create(&key, free);  // free作为析构函数

    // 创建线程...

    pthread_key_delete(&key);
    return 0;
}
```



💻 **实践代码**：`src/chapter08/tsd_demo.c`

---

# 8.7 线程取消

## 8.7.1 取消线程

```
#include <pthread.h>

// 请求取消线程
int pthread_cancel(pthread_t thread);

// 设置取消状态
int pthread_setcancelstate(int state, int *oldstate);
// state: PTHREAD_CANCEL_ENABLE 或 PTHREAD_CANCEL_DISABLE

// 设置取消类型
int pthread_setcanceltype(int type, int *oldtype);
// type: PTHREAD_CANCEL_DEFERRED（延迟）或 PTHREAD_CANCEL_ASYNCHRONOUS（异步）
```

**取消点**：线程只能在取消点被取消，常见取消点：

- `pthread_join`
- `pthread_cond_wait`
- `sleep`，`usleep`
- `read`，`write`
- ...

**示例**：

```
void *thread_func(void *arg) {
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

    while (1) {
        printf("线程运行中...\n");
        sleep(1);  // 取消点
    }

    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_func, NULL);

    sleep(3);
    pthread_cancel(thread);  // 请求取消

    pthread_join(thread, NULL);
    printf("线程已取消\n");

    return 0;
}
```

💻 **实践代码**：`src/chapter08/thread_cancel.c`

---

# 8.8 实战案例

## 8.8.1 案例1：多线程Web服务器

💻 **实践代码**：`src/chapter08/thread_server.c`

简化的多线程服务器架构：

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_THREADS 10

typedef struct {
    int client_id;
    // 其他客户端信息
} client_info_t;

void *handle_client(void *arg) {
    client_info_t *client = (client_info_t *)arg;

    printf("线程处理客户端 %d\n", client->client_id);

    // 处理客户端请求...

    free(client);
    return NULL;
}

int main() {
    pthread_t threads[MAX_THREADS];

    for (int i = 0; i < MAX_THREADS; i++) {
        client_info_t *client = malloc(sizeof(client_info_t));
        client->client_id = i;

        pthread_create(&threads[i], NULL, handle_client, client);
        pthread_detach(threads[i]);  // 分离线程
    }

    // 主线程继续接受新连接...

    return 0;
}
```

## 8.8.2 案例2：线程池

💻 **实践代码**：`src/chapter08/thread_pool.c`

**线程池优势**：

- 避免频繁创建/销毁线程的开销
- 控制并发数量
- 任务队列缓冲请求

基本结构：

```
typedef struct {
    pthread_t *threads;       // 线程数组
    task_queue_t *queue;      // 任务队列
    int thread_count;         // 线程数量
    int shutdown;             // 关闭标志
    pthread_mutex_t lock;
    pthread_cond_t notify;
} thread_pool_t;

// 工作线程函数
void *worker_thread(void *arg) {
    thread_pool_t *pool = (thread_pool_t *)arg;

    while (1) {
        pthread_mutex_lock(&pool->lock);

        // 等待任务
        while (queue_is_empty(pool->queue) && !pool->shutdown) {
            pthread_cond_wait(&pool->notify, &pool->lock);
        }

        if (pool->shutdown) {
            pthread_mutex_unlock(&pool->lock);
            break;
        }

        // 取出任务
        task_t task = queue_pop(pool->queue);

        pthread_mutex_unlock(&pool->lock);

        // 执行任务
        task.function(task.arg);
    }

    return NULL;
}
```

# 8.9 API快速参考

## 8.9.1 线程管理

| 函数 | 功能 | 返回值 |
| --- | --- | --- |
| `pthread_create` | 创建线程 | 0成功，错误码 |
| `pthread_exit` | 退出线程 | 无 |
| `pthread_join` | 等待线程 | 0成功，错误码 |
| `pthread_detach` | 分离线程 | 0成功，错误码 |
| `pthread_self` | 获取当前线程ID | 线程ID |

## 8.9.2 互斥锁

| 函数 | 功能 |
| --- | --- |
| `pthread_mutex_init` | 初始化互斥锁 |
| `pthread_mutex_destroy` | 销毁互斥锁 |
| `pthread_mutex_lock` | 加锁（阻塞） |
| `pthread_mutex_trylock` | 尝试加锁（非阻塞） |
| `pthread_mutex_unlock` | 解锁 |



## 8.9.3 条件变量

| 函数 | 功能 |
| --- | --- |
| `pthread_cond_init` | 初始化条件变量 |
| `pthread_cond_destroy` | 销毁条件变量 |
| `pthread_cond_wait` | 等待条件 |
| `pthread_cond_signal` | 唤醒一个线程 |
| `pthread_cond_broadcast` | 唤醒所有线程 |

## 8.9.4 读写锁

| 函数 | 功能 |
| --- | --- |
| `pthread_rwlock_init` | 初始化读写锁 |
| `pthread_rwlock_destroy` | 销毁读写锁 |

| 函数 | 功能 |
| --- | --- |
| `pthread_rwlock_rdlock` | 获取读锁 |
| `pthread_rwlock_wrlock` | 获取写锁 |
| `pthread_rwlock_unlock` | 解锁 |

---

# 8.10 小结

本章介绍了Linux多线程编程的核心内容：

**核心知识点：**

1. **线程基础**：
    - 线程 vs 进程：共享地址空间，轻量级
    - 创建、终止、等待、分离线程
    - joinable vs detached
2. **互斥锁（Mutex）**：
    - 保护共享资源，防止数据竞争
    - 死锁及避免方法（统一加锁顺序）
3. **条件变量**：
    - 高效的线程等待机制
    - 生产者-消费者模型
    - 必须与互斥锁配合使用
4. **其他同步机制**：
    - 读写锁：读多写少场景
    - 自旋锁：持锁时间极短场景
5. **线程特定数据（TSD）**：
    - 每个线程独立的变量副本

**最佳实践：**

1. **总是保护共享资源**：使用互斥锁保护所有共享变量
2. **避免死锁**：统一加锁顺序，避免嵌套锁
3. **正确使用条件变量**：用while不用if，持有锁时调用
4. **及时释放锁**：临界区尽量小
5. **分离不需要join的线程**：避免资源泄漏

**常见错误：**

- ❌ 忘记初始化/销毁同步对象
- ❌ 访问共享资源时不加锁
- ❌ 条件变量用if而不是while

- ❌ 忘记unlock（考虑使用RAII）
- ❌ 创建线程后忘记join或detach

**学习建议：**

1. 理解线程和进程的本质区别
2. 掌握互斥锁的正确使用
3. 深入理解条件变量的工作原理
4. 实践生产者-消费者等经典模型
5. 学习使用线程调试工具（如Helgrind）

---

# 8.11 练习题

1. **基础练习**：
    - 编写多线程程序计算1到1亿的累加和
    - 实现一个线程安全的计数器
2. **进阶练习**：
    - 实现读者-写者问题（读写锁）
    - 实现哲学家就餐问题（避免死锁）
3. **实战练习**：
    - 实现一个简单的线程池
    - 编写多线程文件下载器

---

💡 **提示**：线程编程的关键是正确的同步，理解临界区、原子操作和同步机制的本质非常重要。