用户空间I/O

User Space I/O

本章内容

Topic

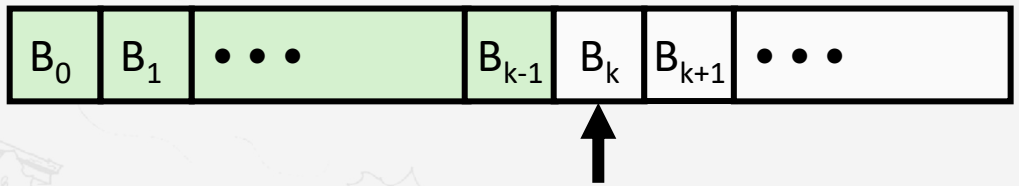
- Unix I/O
- RIO包
- 元数据、共享和重定向
- 标准I/O
- 如何选择I/O函数

概述

- Linux文件是包含了m个字节的序列：
  - B_0，B_1，…，B_k，…，B_{m-1}
- 一个有趣的事实：Linux系统中所有I/O设备都被表示为文件：
  - /dev/sda2 （磁盘分区）
  - /dev/tty2 （终端）
- 甚至内核也被表示为文件：
  - /boot/vmlinuz-3.13.0-55-generic （内核映像）
  - /proc （内核数据结构）

概述

- 优雅地将文件映射到设备，使得内核能够提供一个简单的接口称为Unix I/O：
  - 打开和关闭文件：open() 和 close()
  - 读取和写入文件：read()和write()
  - 改变当前文件游标： lseek()
    - 指示要读取或写入的文件的下一个偏移量



文件类型

- 每个文件都有一个类型，指示其在系统中的角色：
  - 普通文件：包含任意数据
  - 目录：相关文件组的索引
  - 套接字：用于与另一台机器上的进程通信
- 其他文件类型：
  - 命名管道（FIFOs）
  - 符号链接
  - 字符设备和块设备

普通文件

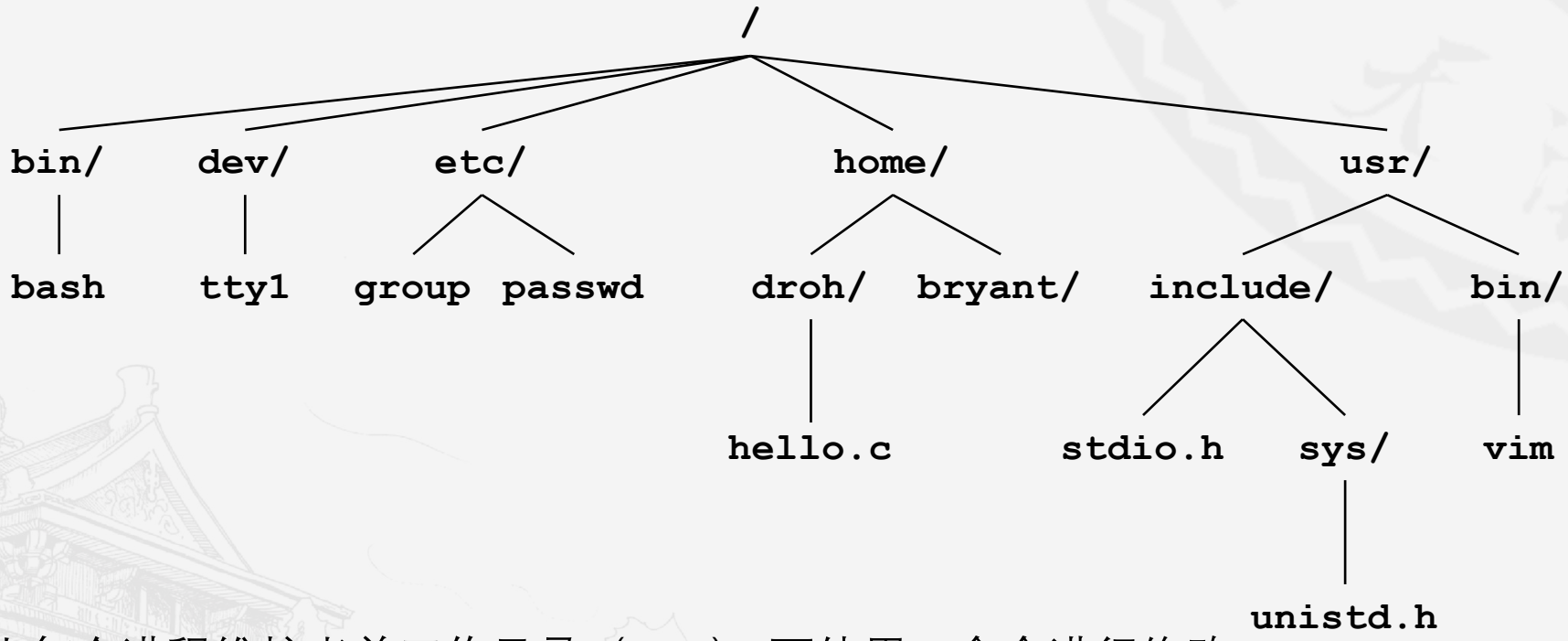
- 普通文件可以包含包含任意数据
- 应用程序通常将其分为两类：文本文件和二进制文件
  - 文本文件是只包含ASCII或Unicode字符的普通文件
  - 二进制文件是其他所有类型的文件
    - 例如，目标文件，JPEG图像
  - 内核并不知道以上两者的区别
- 文本文件是文本行的序列
  - 文本行是以换行符（‘\n‘）为终止标志的字符序列
    - 换行符是0xa，与ASCII换行字符（LF）相同
- 其他系统中的行结束（EOL）指示符
  - Linux和Mac OS：‘\n’（0xa）：换行（LF）
  - Windows和Internet协议：‘\r\n’（0xd 0xa）：回车（CR）后跟换行（LF）

目录

- 目录由一个链接数组组成
  - 每个链接将文件名映射到一个文件
- 每个目录至少包含两个基本项
  - . 是指向自身的链接
  - .. 是指向目录层次结构中父目录的链接
- 用于操作目录的命令
  - mkdir：创建空目录
  - ls：查看目录内容
  - rmdir：删除空目录

目录层次结构

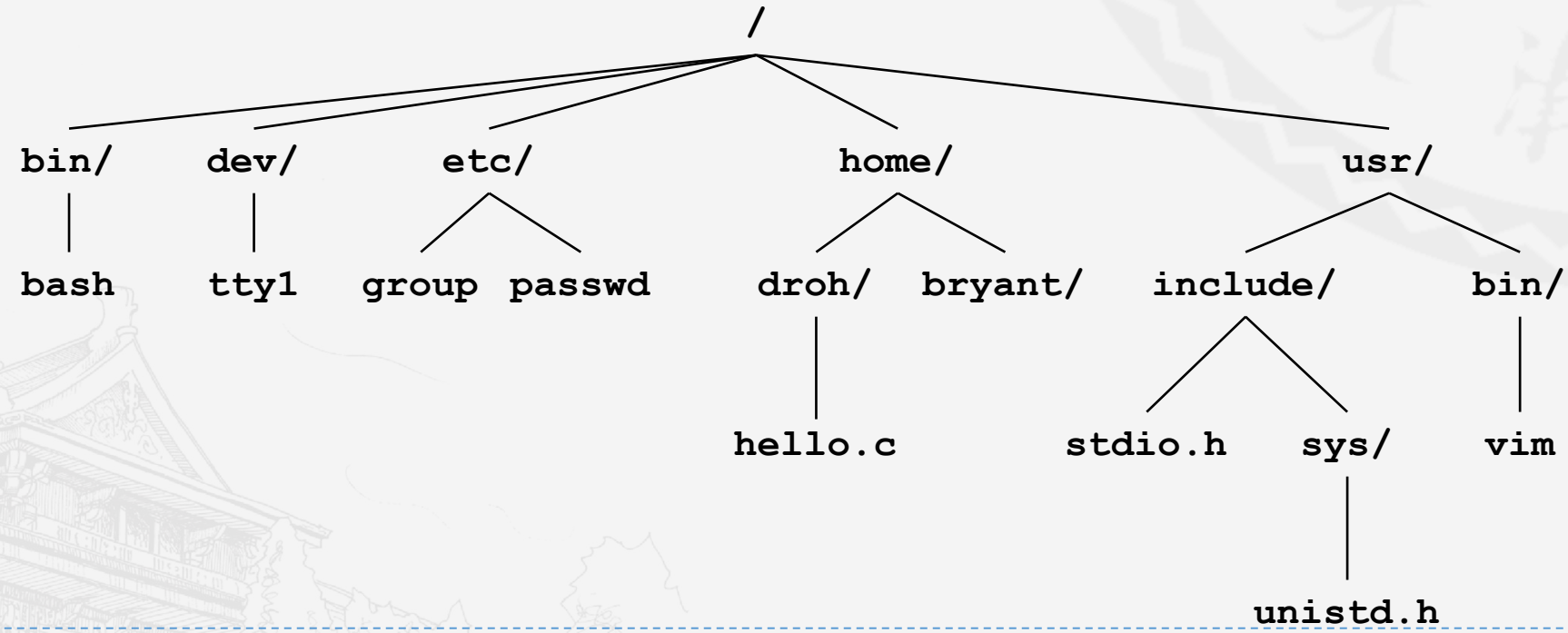
- 所有文件都组织成一个以根目录为锚点的层次结构，根目录名为/（斜杠）



- 内核为每个进程维护当前工作目录（cwd） 可使用cd命令进行修改

路径名

- 文件在层次结构中的位置由路径名表示
  - 绝对路径名以'/'开头，表示从根目录开始的路径 /home/droh/hello.c
  - 相对路径名表示从当前工作目录开始的路径 ../home/droh/hello.c



打开文件

- 打开文件将会通知内核你准备访问该文件

```
int fd;   /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

- 返回一个小的标识整数文件描述符
  - fd == -1 表示发生了错误
- 由Linux shell创建的每个进程在初始状态时与终端关联三个打开文件：
  - 0：标准输入（stdin）
  - 1：标准输出（stdout）
  - 2：标准错误（stderr）

关闭文件

- 关闭文件通知内核你已经完成对该文件的访问

```
int fd;     /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
   perror("close");
   exit(1);
}
```

- 在多线程程序中关闭已关闭的文件是灾难的开始
- 规范：即使是看似无害的函数，如close()，也要始终检查返回码

读文件

- 读取文件将字节从当前文件位置复制到内存，然后更新文件位置

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* Open file fd ...  */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

- 返回从文件fd读取到buf中的字节数
  - 返回类型ssize_t是有符号整数
  - nbytes < 0 表示发生了错误
  - 短计数（nbytes < sizeof(buf) ）是可能的，且不是错误!

写文件

- 写入文件将字节从内存复制到当前文件位置，然后更新当前文件位置

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

- 返回从buf写入到文件fd中的字节数
  - nbytes < 0 表示发生了错误
  - 与读取一样，短计数是可能的，且不是错误!

简单的Unix I/O示例

- 将stdin复制到stdout，每次复制一个字节

```
#include "csapp.h"

int main(void)
{
    char c;

    while(Read(STDIN_FILENO, &c, 1) != 0)
        Write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

关于短计数

- 在以下情况下可能出现短计数：
  - 在读取时遇到（文件结束符）EOF
  - 从终端读取文本行
  - 读取和写入网络套接字
- 在以下情况下永远不会出现短计数：
  - 从磁盘文件读取（除了EOF）
  - 向磁盘文件写入
- 在实践中最好的方式是，在程序中始终容忍短计数出现的可能

本章内容

Topic

- Unix I/O
- RIO包
- 元数据、共享和重定向
- 标准I/O
- 如何选择I/O函数

RIO包

Robust I/O Package

- RIO是一组包装器，提供了高效和健壮的I/O，适用于需要处理短计数的应用程序，比如网络程序

- RIO提供了两种不同类型的函数：
  - 无缓冲的二进制数据输入和输出：rio_readn 和 rio_writen
  - 有缓冲的文本行和二进制数据输入：rio_readlineb 和 rio_readnb

- 缓冲的RIO例程是线程安全的，并且可以任意交错地在相同的描述符上使用

RIO包

Robust I/O Package

无缓冲的输入和输出

- 与Unix的read和write具有相同的接口
- 特别适用于在网络套接字上传输数据

```
#include "csapp.h"

ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, void *usrbuf, size_t n);

     Return: num. bytes transferred if OK,  0 on EOF (rio_readn only), -1 on error
```

  - rio_readn仅在遇到EOF时返回短计数
  - 仅在知道要读取多少字节时才使用它
  - rio_writen永远不会返回短计数
  - 可以任意交错地在相同的描述符上调用rio_readn和rio_writen

rio_readn的实现

```
/* rio_readn - Robustly read n bytes (unbuffered) */
ssize_t rio_readn(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nread;
    char *bufp = usrbuf;

    while (nleft > 0) {
        if ((nread = read(fd, bufp, nleft)) < 0) {
            if (errno == EINTR) /* Interrupted by sig handler return */
                nread = 0;      /* and call read() again */
            else
                return -1;      /* errno set by read() */
        }
        else if (nread == 0)
            break;              /* EOF */
        nleft -= nread;
        bufp += nread;
    }
    return (n - nleft);         /* Return >= 0 */
}
```

RIO包

Robust I/O Package

有缓冲的输入

- 从部分缓存在内部内存缓冲区中的文件中高效读取文本行和二进制数据

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);

ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);

                          Return: num. bytes read if OK, 0 on EOF, -1 on error
```

  - rio_readlineb 从文件fd中读取最多maxlen字节的文本行，并将该行存储在usrbuf中
    - 特别适用于从网络套接字读取文本行
  - 停止条件：
    - 读取了maxlen字节
    - 遇到EOF
    - 遇到换行符（'\n'）

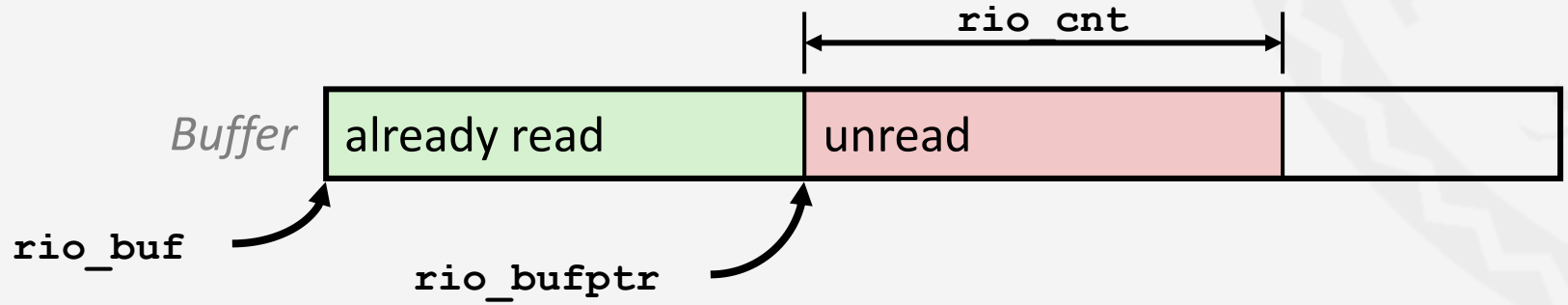
有缓冲的输入（2）

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);

ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);

                          Return: num. bytes read if OK, 0 on EOF, -1 on error
```

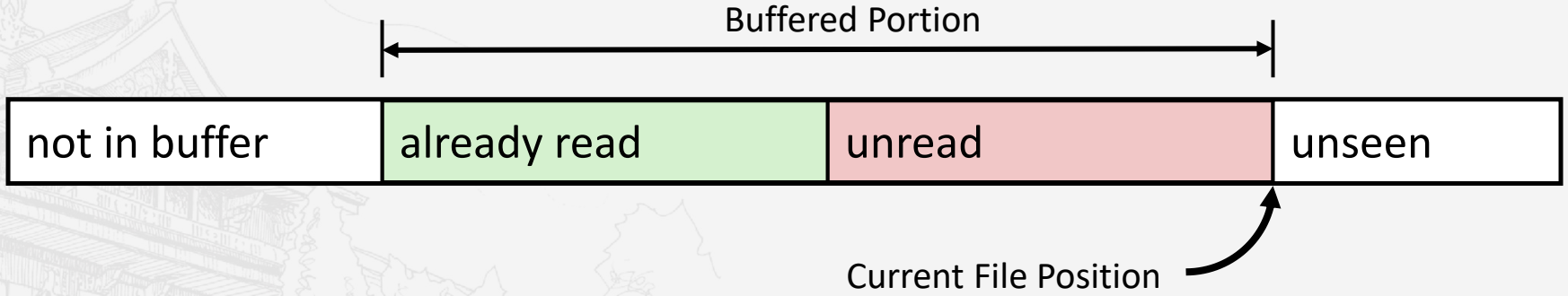
- rio_readnb 从文件fd中读取最多n字节的数据
- 停止条件：
  - 读取了maxlen字节
  - 遇到EOF
- 可以任意交错地在相同的描述符上调用rio_readlineb和rio_readnb
  - 注意：不要与对rio_readn的调用交错

有缓冲输入输出的实现

- 对于从文件读取的情况：
- 文件有关联的缓冲区，用于保存已从文件中读取但尚未被用户代码读取的字节



- 该缓冲区构建在Unix文件之上：

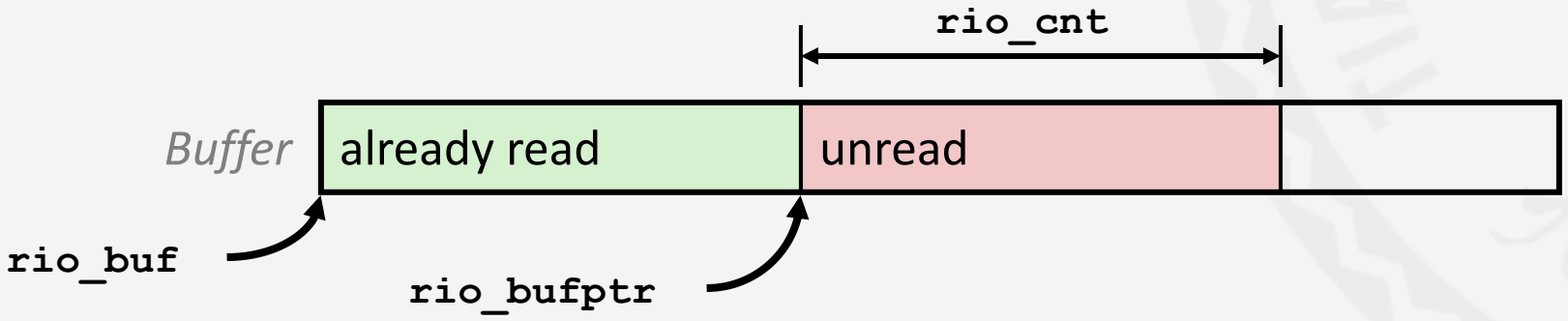


RIO包

Robust I/O Package

rio_t结构体

- 所有关于缓冲区的信息都包含在结构体rio_t中




```
typedef struct {
    int rio_fd;                /* 文件描述符 */
    int rio_cnt;               /* 缓冲区中未读数据的大小 */
    char *rio_bufptr;          /* 指向缓冲区未读数据的指针 */
    char rio_buf[RIO_BUFSIZE]; /* 缓冲区 */
} rio_t;
```

RIO的使用示例

- 将文本文件的行从标准输入复制到标准输出

```
#include "csapp.h"

int main(int argc, char **argv)
{
    int n;
    rio_t rio;
    char buf[MAXLINE];

    rio_readinitb(&rio, STDIN_FILENO);
    while((n = rio_readlineb(&rio, buf, MAXLINE)) != 0)
        rio_writen(STDOUT_FILENO, buf, n);
    exit(0);
}
```

cpfile.c

本章内容

Topic

- Unix I/O
- RIO包
- 元数据、共享和重定向
- 标准I/O
- 如何选择I/O函数

元数据、共享和重定向

Metadata, Sharing, and Redirection

文件元数据

- 元数据是关于数据的数据，在本节中特指文件的数据信息
- 每个文件的元数据都有内核维护
  - 用户可以使用stat和fstat函数访问

```
/* stat和fstat函数返回的元数据 */
struct stat {
    dev_t         st_dev;      /* 设备 */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* 保护模式和文件类型 */
    nlink_t       st_nlink;    /* 硬链接数 */
    uid_t         st_uid;      /* 拥有者ID */
    gid_t         st_gid;      /* 拥有者所在组ID */
    dev_t         st_rdev;     /* 设备类型（如果是inode设备）*/
    off_t         st_size;     /* 总大小（字节数） */
    unsigned long st_blksize;  /* 文件系统I/O块大小 */
    unsigned long st_blocks;   /* 已分配的块数 */
    time_t        st_atime;    /* 最后一次访问时间*/
    time_t        st_mtime;    /* 最后一次修改时间 */
    time_t        st_ctime;    /* inode的改变时间 */
};
```

元数据、共享和重定向

Metadata, Sharing, and Redirection

示例

```c
int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

    Stat(argv[1], &stat);
    if (S_ISREG(stat.st_mode))     /* Determine file type */
        type = "regular";
    else if (S_ISDIR(stat.st_mode))
        type = "directory";
    else
        type = "other";
    if ((stat.st_mode & S_IRUSR)) /* Check read access */
        readok = "yes";
    else
        readok = "no";

    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
```

statcheck.c

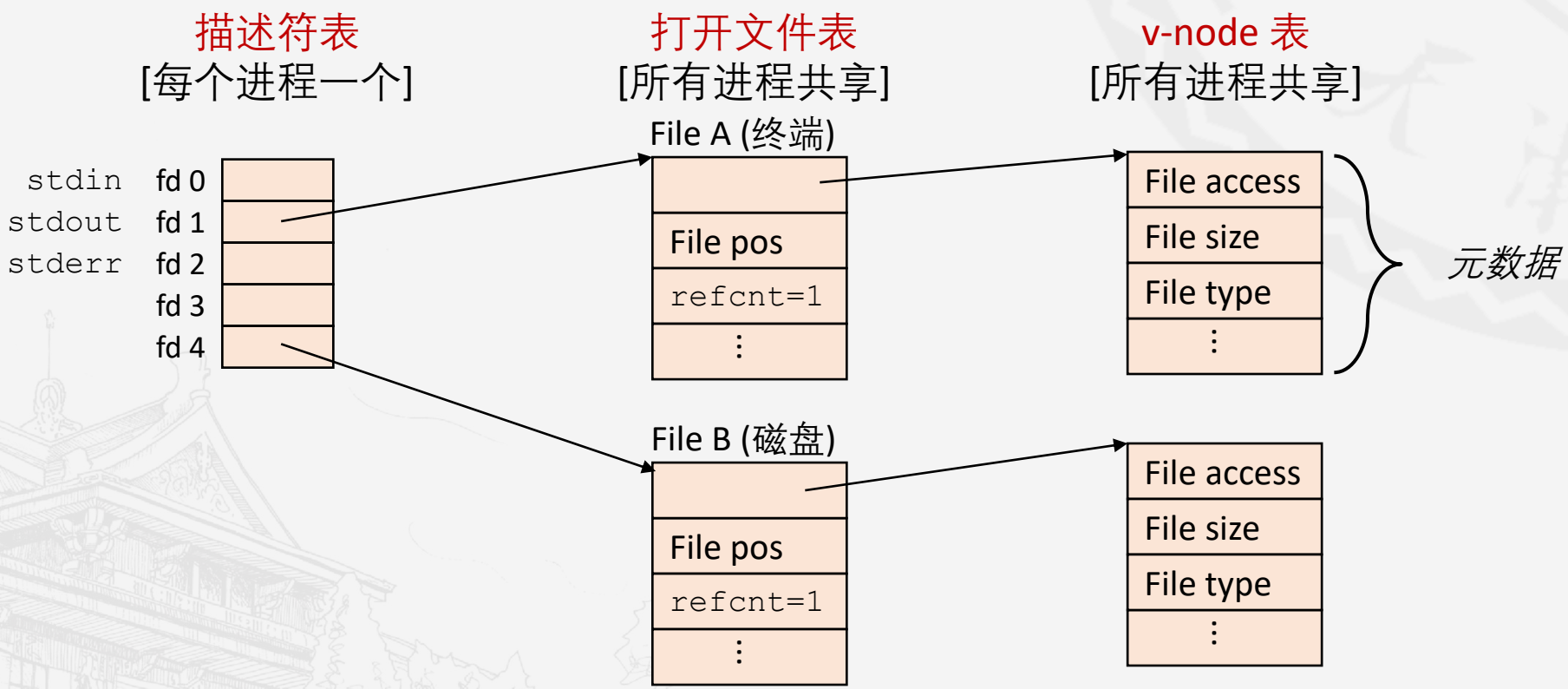
```
linux> ./statcheck statcheck.c
type: regular, read: yes
linux> chmod 000 statcheck.c
linux> ./statcheck statcheck.c
type: regular, read: no
linux> ./statcheck ..
type: directory, read: yes
```

元数据、共享和重定向

Metadata, Sharing, and Redirection

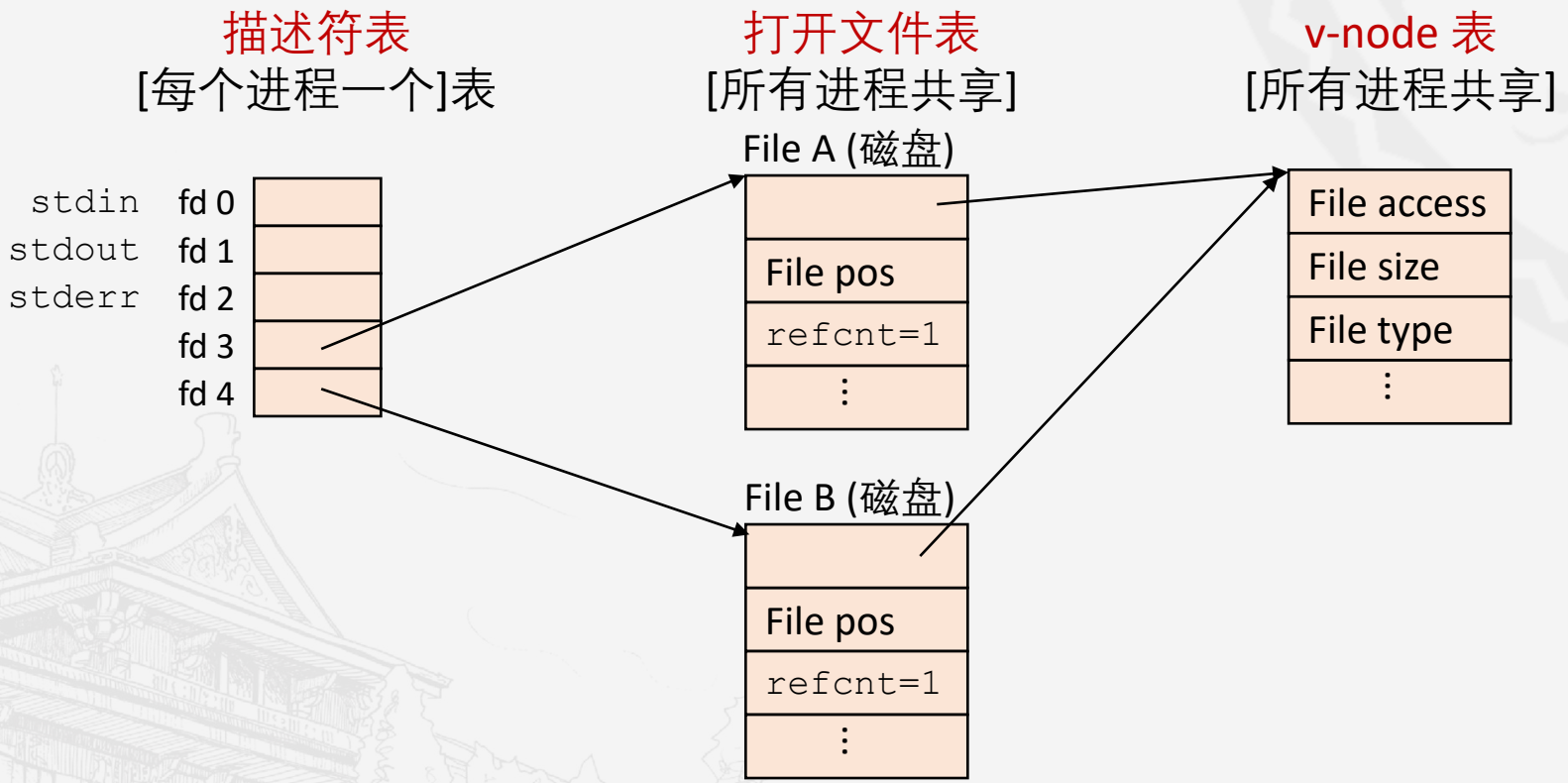
在内核中如何表示已打开的文件

- 两个描述符引用两个不同的打开文件
- 描述符1（stdout）指向终端，描述符4指向打开的磁盘文件



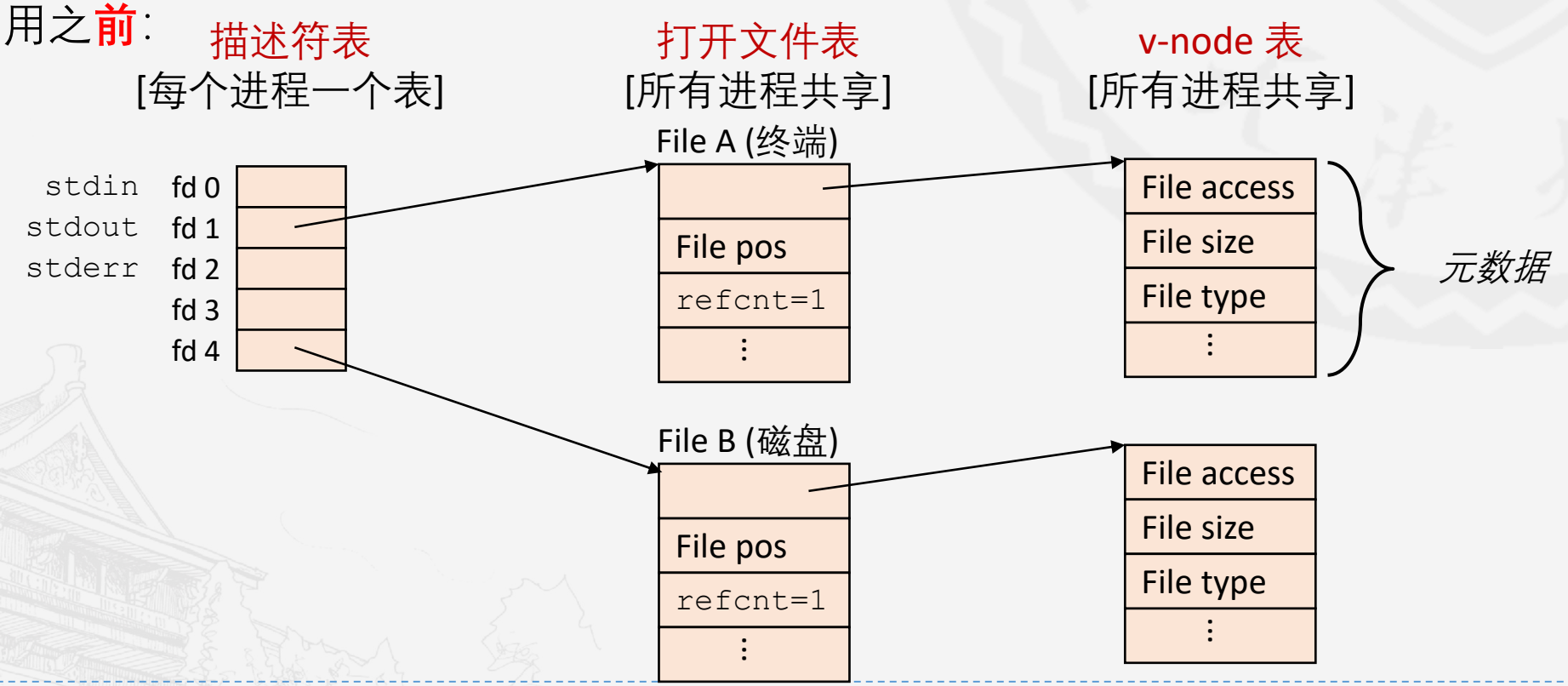
文件共享

- 两个不同描述符共享同一个磁盘文件，通过两个不同的打开文件表项
  - 例如：使用相同的文件名参数两次调用open



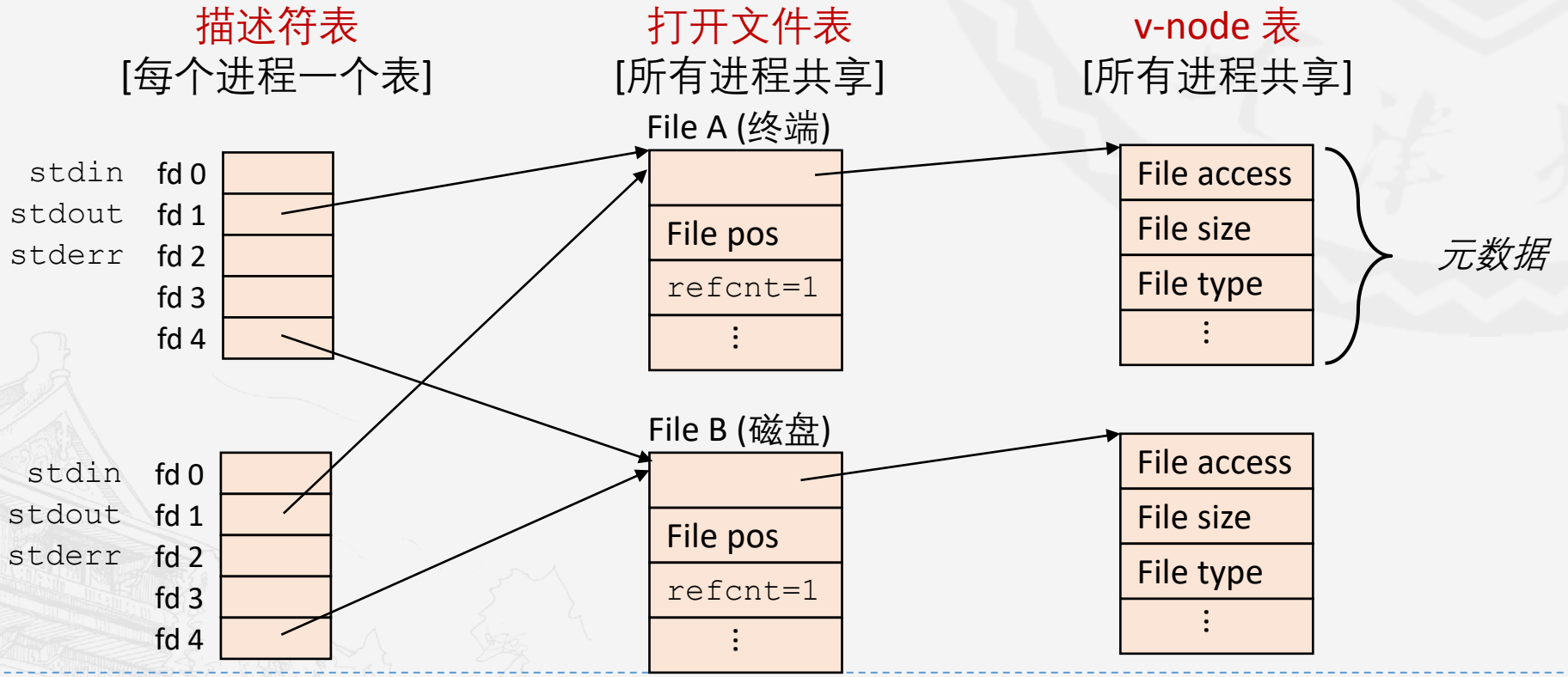
进程间共享文件：使用fork

- 子进程继承其父进程的打开文件
  - 注意：通过exec函数不会改变这种情况（可以使用fcntl来更改）
- 在fork调用之**前**：



进程间共享文件：使用fork（2）

- 子进程继承其父进程的打开文件
- 在fork之后，子进程的打开文件表与父进程相同，并且每个引用计数增加1。



元数据、共享和重定向

Metadata, Sharing, and Redirection

I/O重定向

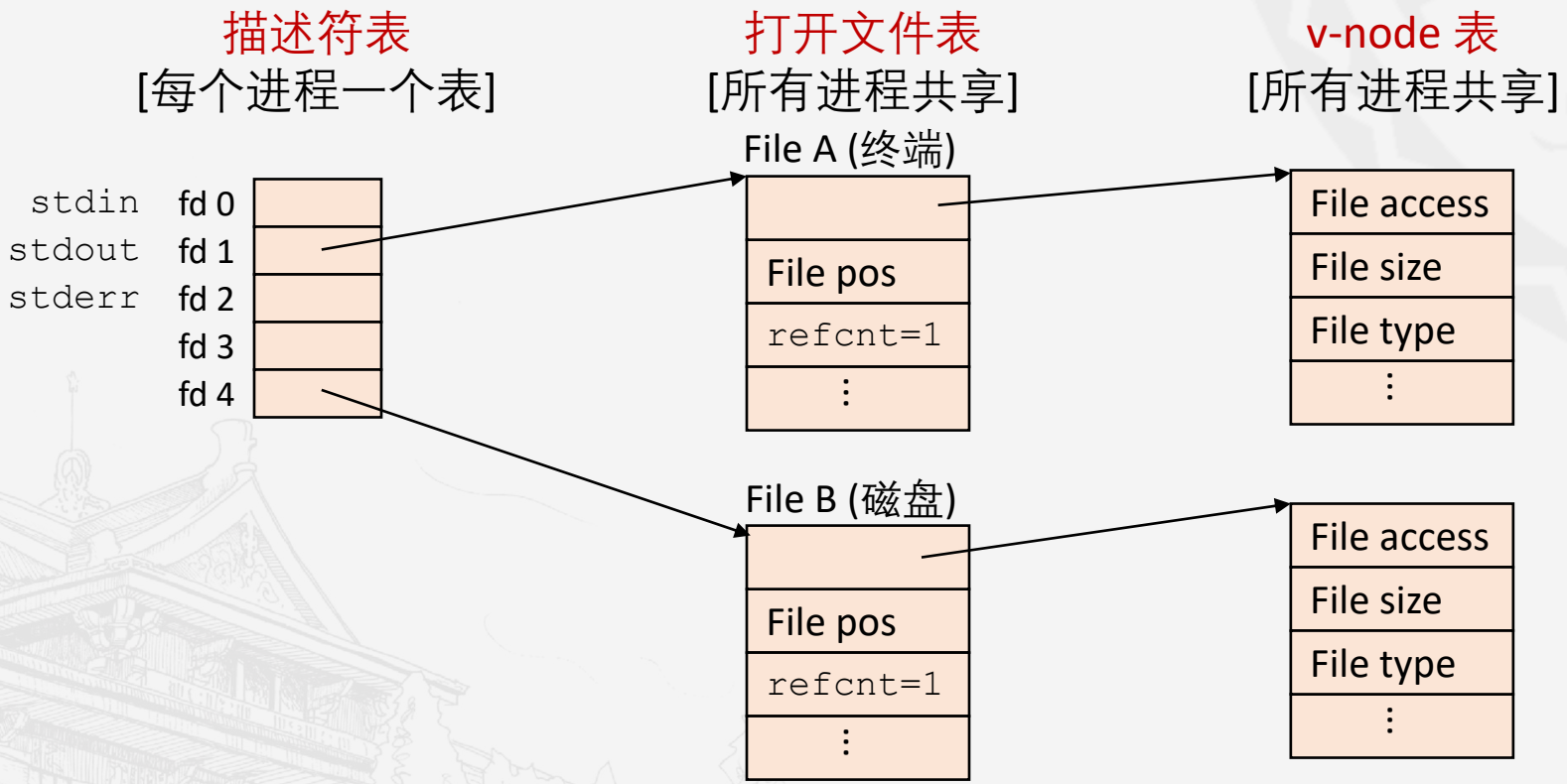
- 问题：在Shell中如何实现I/O重定向？

```
linux> ls > foo.txt
```

- 答案：通过调用dup2(oldfd, newfd)函数
  - 将（每个进程的）描述符表条目oldfd复制到条目newfd

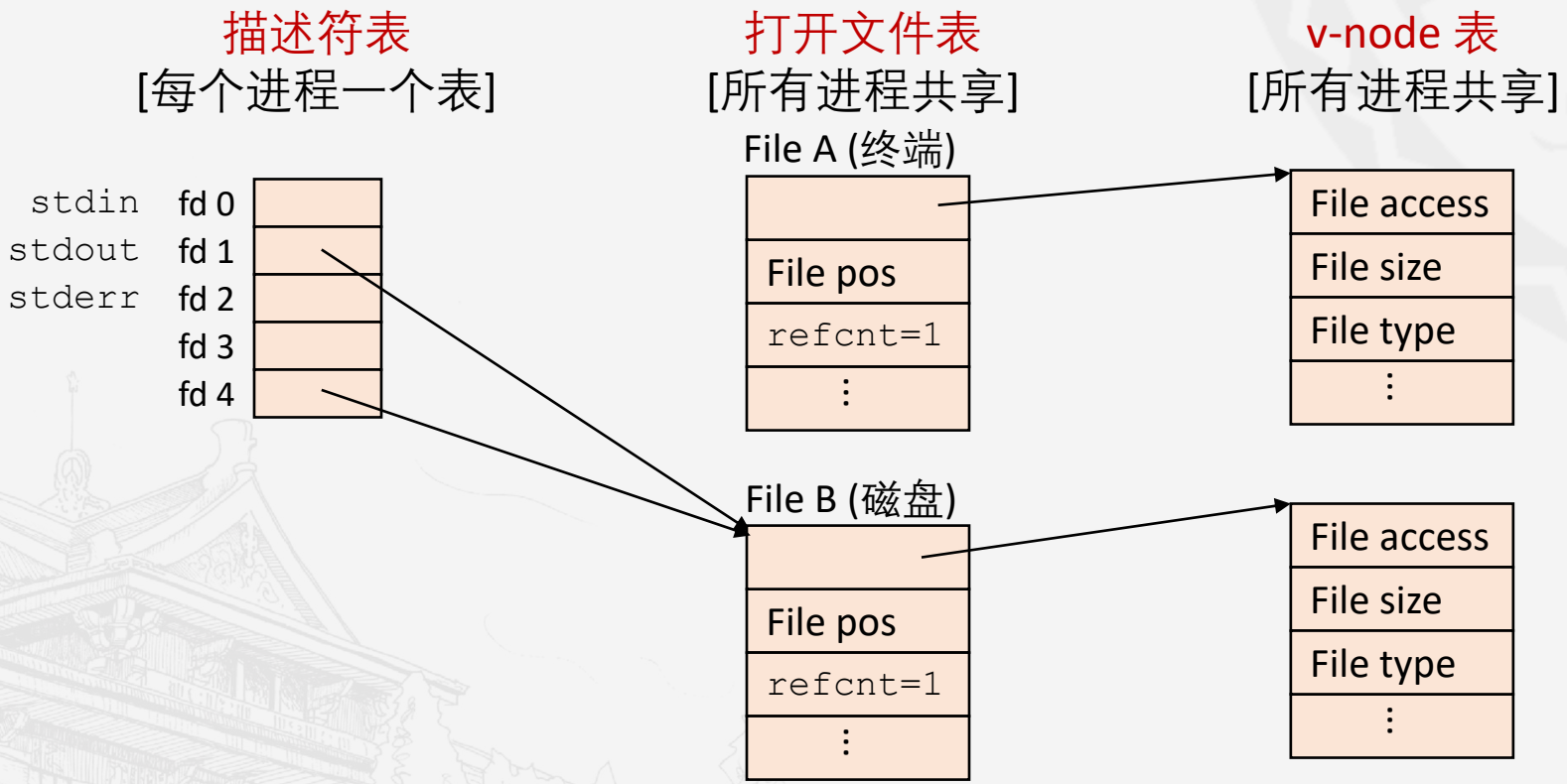
I/O重定向示例

- 步骤1：打开将标准输出重定向到的目标文件
  - 发生在shell fork调用后的子进程中，执行exec之前



I/O重定向示例（2）

- 步骤2：调用 **dup2(4,1)**
  - 将fd=1（标准输出）引用的文件指向fd=4所指向的磁盘文件



本章内容

Topic

- Unix I/O
- RIO包
- 元数据、共享和重定向
- 标准I/O
- 如何选择I/O函数

标准I/O

Standard I/O

- C标准库（libc.so）包含一系列更高级别的标准I/O函数
  - 详见附录B

- 标准I/O函数包括:
  - 文件的打开和关闭（fopen 和 fclose）
  - 基于字节的读取和写入（fread 和 fwrite）
  - 文本行的读取和写入（fgets 和 fputs）
  - 格式化的读取和写入（fscanf 和 fprintf）

标准I/O流

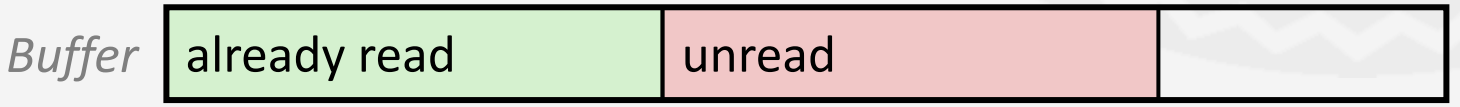
- 标准I/O将文件打开为流
  - 这是文件描述符和内存中的缓冲区的抽象

- C程序在开始时具有三个打开的流（在stdio.h中定义）：
  - stdin（标准输入）
  - stdout（标准输出）
  - stderr（标准错误）

```
#include <stdio.h>
extern FILE *stdin;  /* standard input  (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error  (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

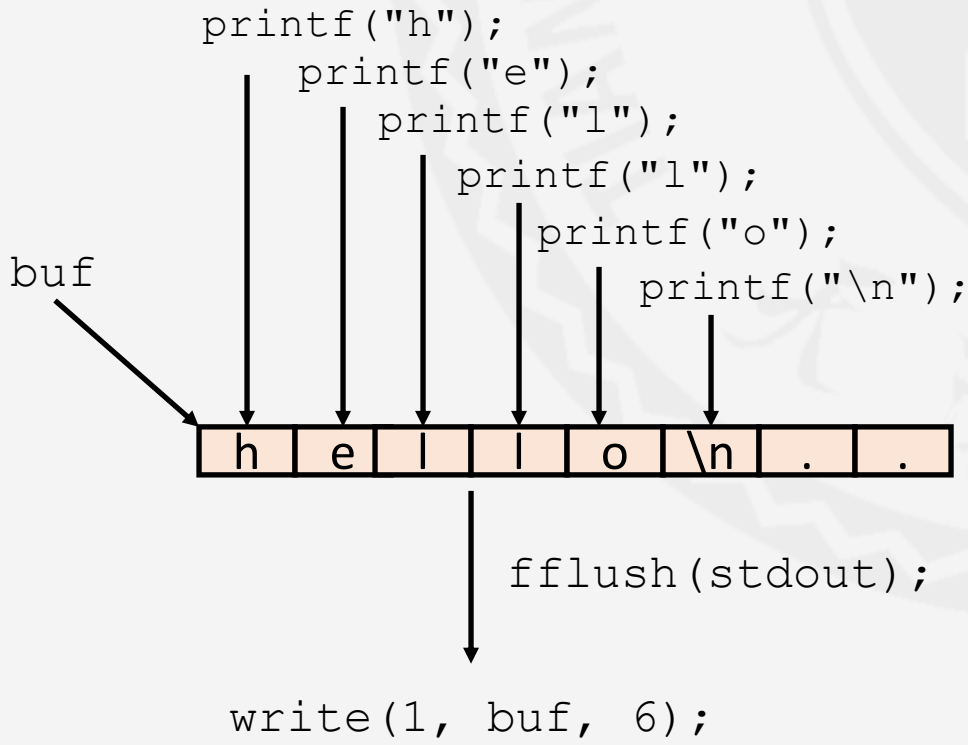
有缓冲的I/O：设计目标

- 应用程序通常以逐个字符的方式进行读写
  - getc、putc 和 ungetc
  - gets 和 fgets
    - 这些函数逐个字符地读取文本行，在遇到换行符时停止
  - 使用Unix I/O调用会降低性能
    - 因为 read 和 write 需要进行系统调用
    - > 10,000个时钟周期。
- 解决方案：使用缓冲读取
  - 利用Unix read 函数来抓取字节块
  - 用户输入函数从缓冲区逐个字节获取数据
    - 当缓冲区为空时重新填充缓冲区。



标准I/O中的缓冲区

- 标准I/O使用带缓冲区的I/O



- 在遇到“\n”换行符时、调用 fflush 函数、缓冲区满、 调用exit 函数、或从 main 函数返回时，缓冲区才被写入到文件中。

stdout与stderr的区别

- 猜一下在Linux中以下程序输出什么

```
int main()
{
    fprintf(stdout, “hello ");
    fprintf(stderr, “world!");
    return 0;
}
```

输出结果为：world!hello

```
int main()
{
    fprintf(stdout, “hello ");
    fprintf(stderr, “world!\n");
    return 0;
}
```

输出结果为：world!

hello

stdout与stderr的区别（2）

- stdout和stderr都用于标准输出，但是
  - stderr为无缓冲输出
  - stdout为有缓冲输出

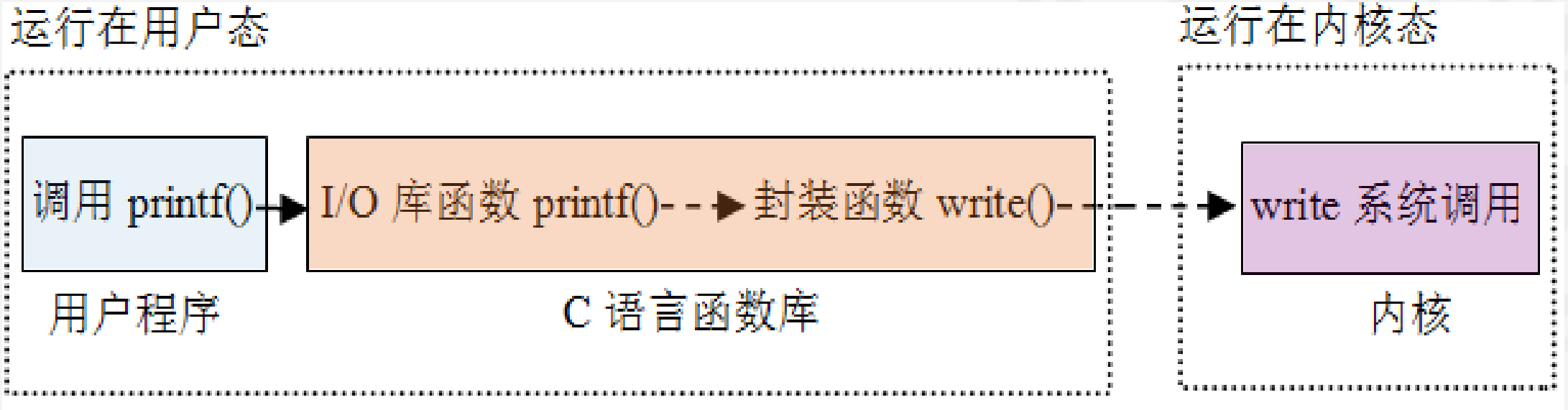
标准I/O缓冲区的工作方式

- 可以使用 Linux 中的 strace 程序来查看这种缓冲机制的实际运行情况。

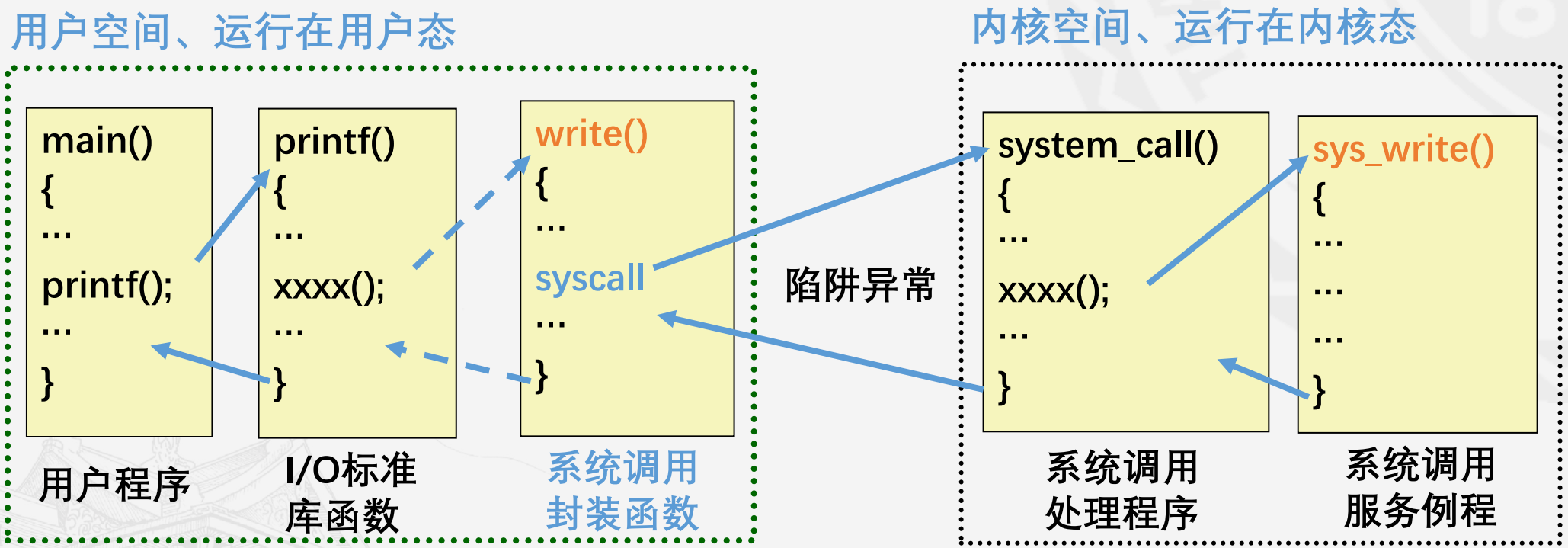
```
#include <stdio.h>

int main()
{
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("\n");
    fflush(stdout);
    exit(0);
}
```

```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6)                  = 6
...
exit_group(0)                           = ?
```

printf()函数的调用过程



printf()函数的调用过程（2）



本章内容

Topic

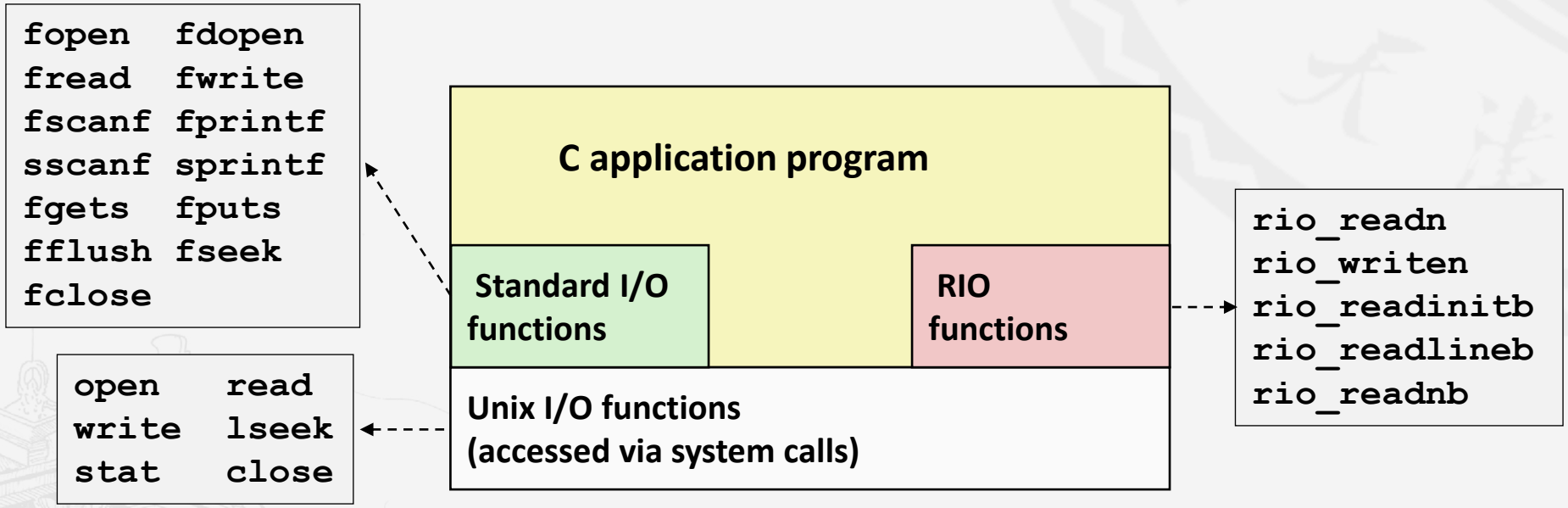
- Unix I/O
- RIO包
- 元数据、共享和重定向
- 标准I/O
- 如何选择I/O函数

如何选择I/O函数

How to Choose I/O Functions

Unix I/O vs. Standard I/O vs. RIO

- 标准I/O和RIO都是使用底层Unix I/O实现的。在程序中应该使用哪一个?



如何选择I/O函数

How to Choose I/O Functions

Unix I/O 的优缺点

- 优点：
  - Unix I/O 是最通用且开销最低的 I/O 形式。
  - 所有其他 I/O 包都是使用 Unix I/O 函数实现的。
  - Unix I/O 提供了访问文件元数据的函数。
  - Unix I/O 函数是异步信号安全的，可以在信号处理程序中安全使用。
- 缺点：
  - 处理短计数（short counts）比较棘手且容易出错。
  - 高效地读取文本行需要一定形式的缓冲，这也是比较棘手且容易出错的。
  - 这两个问题都可以通过标准I/O和RIO包来解决。

标准 I/O 的优缺点

- 优点：
  - 使用缓冲区提高了效率，通过减少read和write调用的次数。
  - 短计数会被自动处理。
- 缺点：
  - 不提供访问文件元数据的函数。
  - 标准I/O函数不是异步信号安全的，并且不适用于信号处理程序。
  - 标准I/O 不适用于网络套接字的输入和输出。
    - 流操作和套接字操作有冲突，而且文档不完善

如何选择I/O函数

How to Choose I/O Functions

- **一般原则：尽可能使用最高级别的I/O函数**
  - 许多C程序员能够完全使用标准I/O函数完成所有工作
  - 但是，请确保你理解所使用的函数!
- **何时使用标准I/O：**
  - 当与磁盘或终端文件一起工作时。
- **何时使用Unix I/O：**
  - 在信号处理程序中，因为Unix I/O是异步信号安全的。
  - 在极少数需要绝对最高性能的情况下。
- **何时使用RIO：**
  - 当您需要读写网络套接字时。
  - 避免在套接字上使用标准I/O。