第10章 系统级I/O

主要内容

- **Unix I/O**
- 用RIO包健壮地读写
- 读取文件元数据，共享和重定位
- 标准I/O
- 结束语

Unix I/O---OS的设备管理、文件管理

- 一个 Linux *文件* 就是一个 ***m*** 字节的序列:
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$

- 现实情况: 所有的I/O设备都被模型化为文件:
  - **/dev/sda2**（用户磁盘分区）
  - **/dev/tty2**（终端）

文件是对IO设备的抽象

- 甚至内核也被映射为文件:
  - **/boot/vmlinuz-3.13.0-55-generic**（内核映像）
  - **/proc**（内核数据结构）

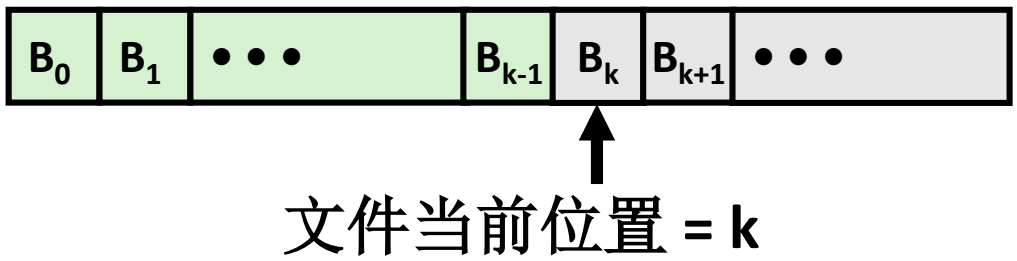
内核是操作系统代码常驻主存的部分。内核不是一个独立的进程,它是系统管理全部进程所用代码和数据结构的集合。

Linux中常见的文件操作命令

- mkdir 创建目录
- rmdir 删除目录
- cp 拷贝文件（目录）
- mv移动文件（目录）
- rm 删除文件（目录）
- touch 创建文件
- cat 查看文件

Unix I/O

- 这种将设备优雅地映射为文件的方式，允许Linux内核引出一个简单、低级的应用接口，称为*Unix I/O:*
  - 打开和关闭文件
    - **`open()`** and **`close()`**
  - 读写文件
    - **`read()`** and **`write()`**
  - 改变*当前的文件位置* (seek)
    - 指示文件要读写位置的偏移量
    - **`lseek()`**



File Types文件类型

- **每个Linux文件都有一个类型（type）来表明它在系统中的角色：**
  - 普通文件 (Regular file): 包含任意数据
  - 目录 (Directory): 一组链接文件的索引
  - 套接字 (Socket): 用来与另一个进程进行跨网络通信的文件
- **其他文件类型（不在讨论范畴）**
  - *命名通道（Named pipes (FIFOs)）*
  - *符号链接（Symbolic links）*
  - *字符和块设备（Character and block devices）*

Regular Files 普通文件

- 普通文件包含任意数据
- 应用程序常常要区分文本文件（*text files*）和二进制文件（*binary files*）
  - 文本文件是只包含 ASCII 或 Unicode字符的普通文件
  - 二进制文件是所有其他文件
    - 比如 目标文件, JPEG图像文件等等
  - 内核并不知道两者之间的区别
- **Linux文本文件是文本行的序列**
  - 文本行是一个字符序列，以一个新行符 (‘\n’)结束
    - 新行符为 0xa，与 ASCII 的换行符 (LF) 是一样的
- **其他系统中的行结束标志**
  - Linux和 Mac 操作系统 : ‘\n’ (0xa)
    - 换行 (LF，line feed)
  - Windows 和 因特网络协议: ‘\r\n’ (0xd 0xa)
    - Carriage return (CR) followed by line feed (LF)
      回车换行

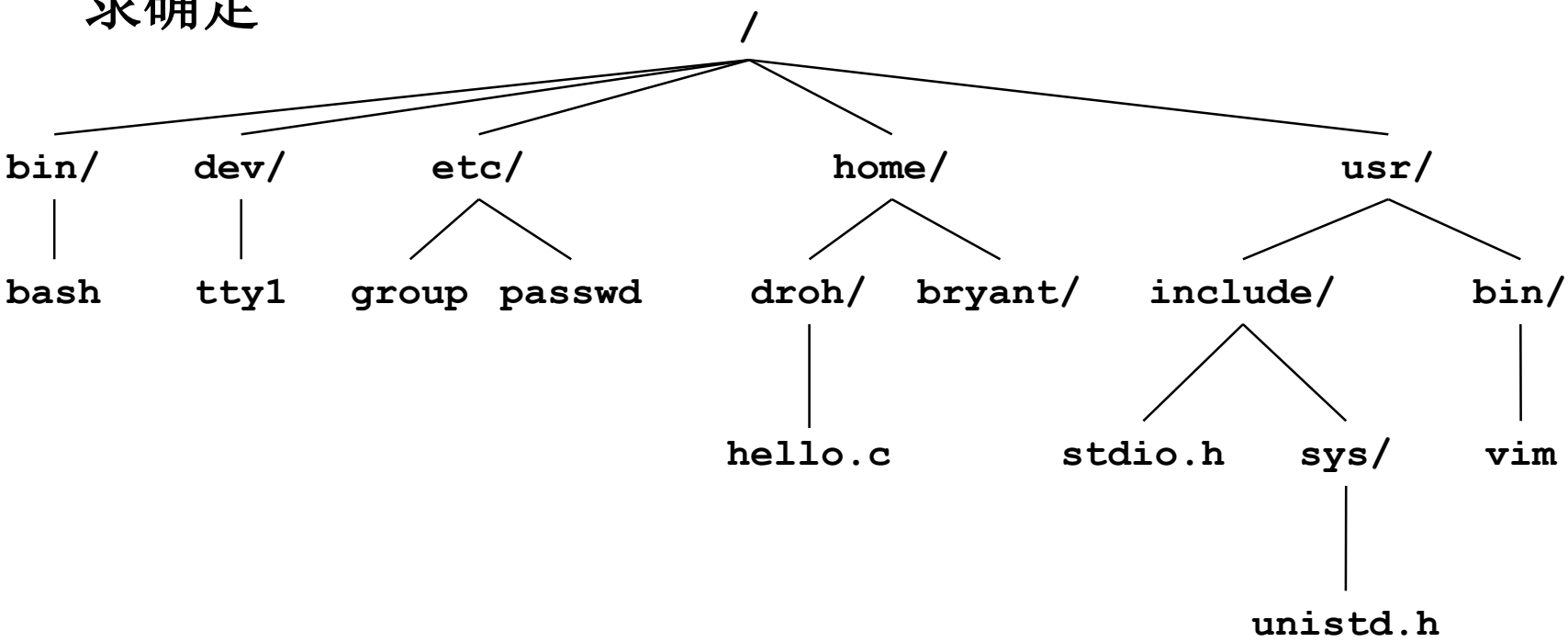
https://www.bilibili.com/video/BV1qP411f7jM/?vd_source=9b3ebd8f0d9db179d1b637e52d4b1303

Directories 目录

- 目录包含一组链接
  - 每个链接将一个文件名映射到一个文件
- 每个目录至少含有两个条目
  - . 是到该文件自身的链接
  - .. 是到目录层次结构中父目录的链接
- 操作目录命令
  - mkdir: 创建空目录
  - ls: 查看目录内容
  - rmdir: 删除空目录

Directory Hierarchy 目录层次结构

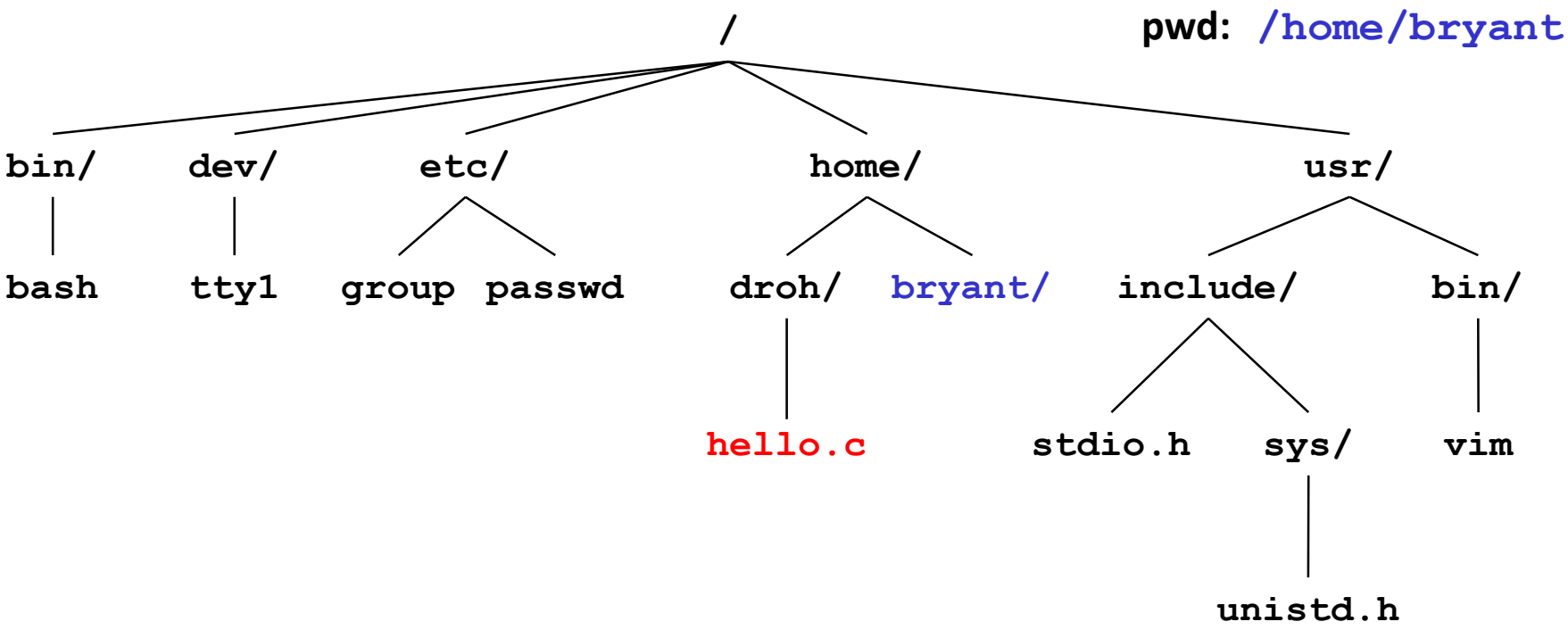
- 所有文件都组织成一个目录层次结构，由名为 / (斜杠)的根目录确定



- 内核为每个进程都保存着一个当前工作目录 (***current working directory (cwd)***)
  - 可以用cd命令来修改shell中的当前工作目录

Pathnames 路径名

- 目录层次结构中的位置用*路径名*来指定
  - *绝对路径名* 以 ‘/’ 开始，表示从根节点开始的路径
    - `/home/droh/hello.c`
  - *相对路径名* 以文件名开始，表示从当前工作目录开始的路径
    - `../home/droh/hello.c`



Opening Files 打开文件

- 打开文件是通知内核你准备好访问该文件

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

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

- 返回一个小的描述符数字---- *文件描述符*。返回的描述符总是在进程中当前没有打开的最小描述符。
  - `fd == -1` 说明发生错误
- Linux内核创建的每个进程都以与一个终端相关联的三个打开的文件开始:
  - 0: 标准输入 (stdin)
  - 1: 标准输出 (stdout)
  - 2: 标准错误 (stderr)

Closing Files 关闭文件

- 关闭文件是通知内核你要结束访问一个文件

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

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

- 关闭一个已经关闭的文件是导致线程程序灾难的一个因素
- 好习惯: 总是检查返回码，即使是看似良性的函数，比如 `close()`

Reading Files 读文件

- 读文件从当前文件位置复制字节到内存位置，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);
}
```

- 返回值表示的是实际传送的字节数量
  - 返回类型 **ssize_t** 是有符号整数
  - **nbytes < 0** 表示发生错误
  - *不足值（Short counts）*(**nbytes < sizeof(buf)**) 是可能的，不是错误！

Writing Files 写文件

- 写文件从内存复制字节到当前文件位置，然后更新文件位置

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

/* 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);
}
```

- 返回值表示的是从内存向文件fd实际传送的字节数量
  - **nbytes < 0** 表明发生错误
  - 同读文件一样, 不足值（short counts） 是可能的，并不是错误!

简单Unix I/O示例

- 一次一个字节地从标准输入复制到标准输出

```
#include "csapp.h"

int main(void)
{
    char c;

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

On Short Counts 不足值

- 出现“不足值”的几种情况:
  - Encountering (end-of-file) EOF on reads 读时遇到EOF
  - Reading text lines from a terminal 从终端读文本行
  - Reading and writing network sockets 读写网络套接字

- 正常情况:
  - 正常的读磁盘文件 （遇到 EOF出现不足值）
  - 正常的写磁盘文件（磁盘满也会出现不足值）

- 最好的解决办法就是一直允许不足值，反复处理不足值
  - 反复调用read和write处理不足值，直到所有需要的字节都传送完毕

主要内容

- Unix I/O
- 用RIO包健壮地读写
- 读取文件元数据，共享和重定位
- 标准I/O
- 结束语

The RIO Package RIO包（Robust）

- RIO是一个封装体，在像网络程序这样容易出现不足值的应用中，提供了方便、健壮和高效的I/O
- RIO提供两类不同的函数
  - 无缓冲的输入输出函数 Unbuffered input and output of binary data
    - **`rio_readn`**和 **`rio_writen`**
  - 带缓冲的输入函数 Buffered input of text lines and binary data
    - **`rio_readlineb`** 和 **`rio_readnb`**
    - 带缓冲的 RIO 输入函数是线程安全的，它在同一个描述符上可以被交错地调用
- 下载地址： http://csapp.cs.cmu.edu/3e/code.html

  → **`src/csapp.c`** and **`include/csapp.h`**

RIO的无缓冲的输入输出函数

- 使用与 Unix **read** 和 **write**相同的接口
- 对于在网络套接字上传输数据特别有用

```
#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** 在遇到 EOF时只能返回一个不足值
  - 只有当确定读取字节数时才使用它
- **rio_writen** 绝不会返回不足值
- 对同一个描述符，可以任意交错地调用**rio_readn** 和 **rio_writen**

rio_readn函数

```
/*
 * 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 */
}
```

csapp.c

errno 变量的值是一个整型数，每个值对应一种特定的错误类型。例如：
- EACCES：权限不足。
- ENOENT：没有这样的文件或目录。
- EINTR：被信号中断。
- ENOMEM：内存不足。

errno 变量通常由系统调用设置，但也有一些库函数可能会修改它。在使用 errno 时，需要注意以下几点：

1.线程安全：在多线程程序中，errno 是线程局部存储的，这意味着每个线程都有自己的 errno 副本，因此它是线程安全的。

2.错误检查：在使用系统调用或库函数后，应该立即检查 errno 的值，以确定操作是否成功。

3.错误处理：在捕获错误后，可以使用 perror() 或 strerror() 函数将 errno 值转换为可读的错误消息。

4.重置 errno：在某些情况下，你可能需要重置 errno 的值（例如，在忽略某些错误之后）。这可以通过将 errno` 设置为 0 来实现。

RIO的带缓冲的输入函数

- 高效地从内部内存缓冲区中缓存的文件中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中读取最大长度文本行，并存储在 **usrbuf**
    - 对于从网络套接字上读取文本行特别有用
  - **停止条件**
    - 已经读了最大字节数
    - 遇到EOF
    - 遇到新行符 (‘**\n**’)

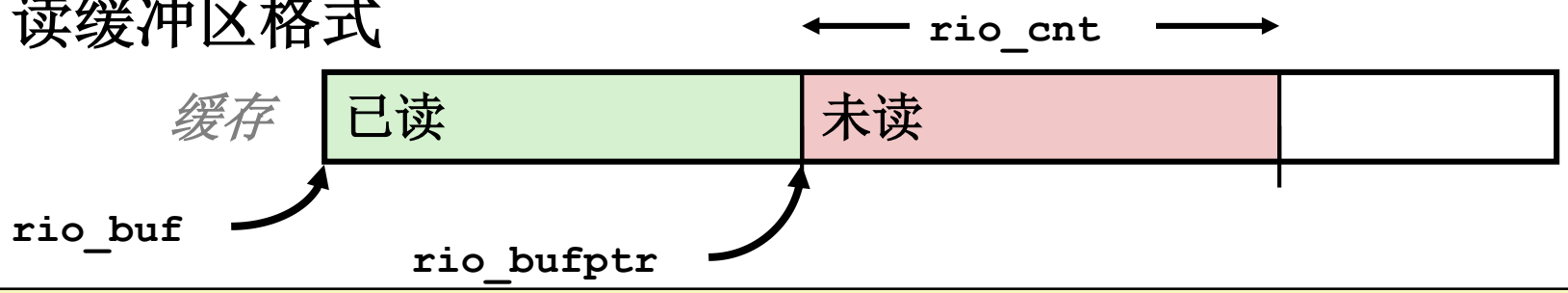
RIO的带缓冲的输入函数

```
#include "csapp.h"
//将描述符fd和地址rp处的一个类型为rio_t的读缓冲区联系起来
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** 最多读**n**个字节
- 停止条件
  - 已读最大字节数 **maxlen**
  - 遇到EOF
- 同一个描述符对 **rio_readlineb** 和 **rio_readnb** 的调用可以任意交叉进行
  - **警告**: 不要和 **rio_readn** 函数交叉使用

Buffered I/O: Declaration 带缓存的IO：声明

- 读缓冲区格式



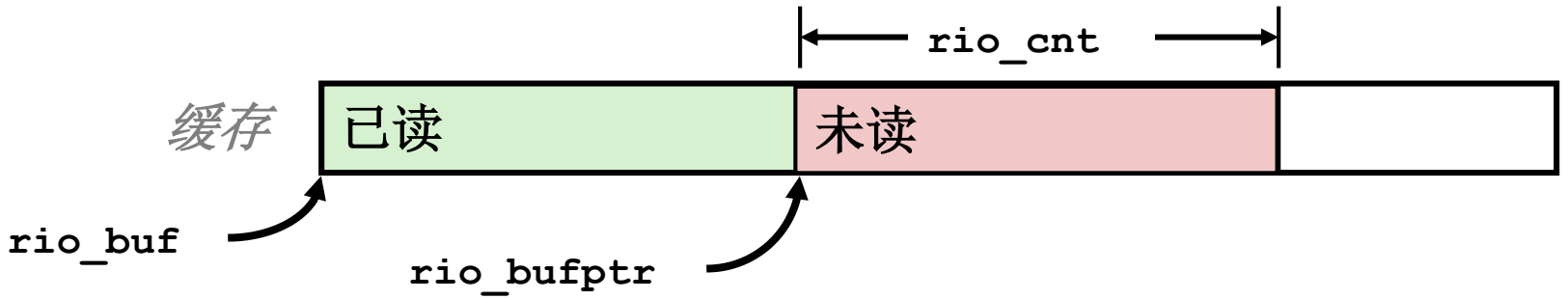

```
typedef struct {
    int rio_fd;                /* descriptor for this internal buf */
    int rio_cnt;               /* unread bytes in internal buf */
    char *rio_bufptr;          /* next unread byte in internal buf */
    char rio_buf[RIO_BUFSIZE]; /* internal buffer */
} rio_t;
```

- rio_readinitb函数

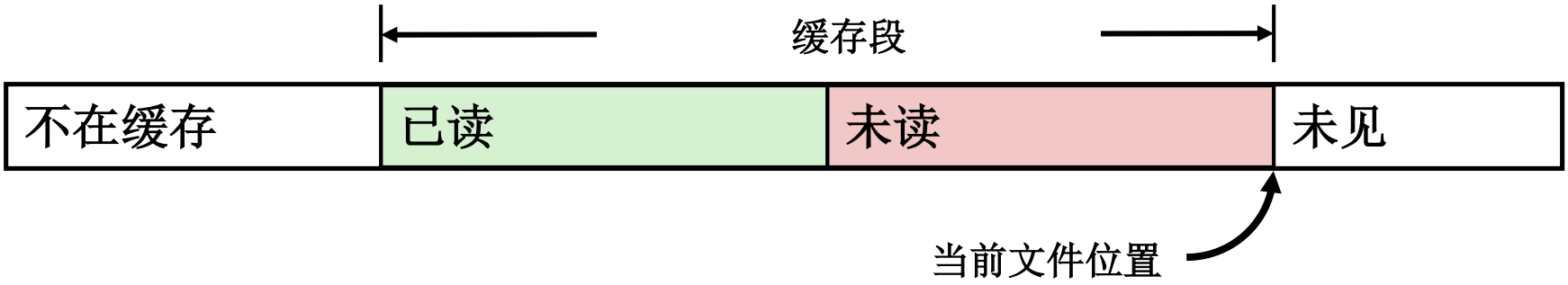
```
void rio_readinitb(rio_t *rp, int fd)
{
    rp->rio_fd = fd;
    rp->rio_cnt = 0;
    rp->rio_bufptr = rp->rio_buf;
}
```

Buffered I/O: Implementation 带缓冲I/O的应用

- 读文件
- 文件有关联的缓冲区来保存从文件中读取，但是还未被用户代码读取的字节



- Unix文件分层:



RIO函数示例

- 从标准输入复制一个文本文件到标准输出

```
#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

主要内容

- Unix I/O
- 用RIO包健壮地读写
- 读取文件元数据，共享和重定位
- 标准I/O
- 结束语

File Metadata 读取文件元数据

- *元数据 (Metadata)* 是关于文件的信息
- 每个文件的元数据都由内核来保存
  - 用户通过调用 **stat**和**fstat** 函数访问元数据

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* Device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* Protection and file type */
    nlink_t       st_nlink;    /* Number of hard links */
    uid_t         st_uid;      /* User ID of owner */
    gid_t         st_gid;      /* Group ID of owner */
    dev_t         st_rdev;     /* Device type (if inode device) */
    off_t         st_size;     /* Total size, in bytes */
    unsigned long st_blksize;  /* Blocksize for filesystem I/O */
    unsigned long st_blocks;   /* Number of blocks allocated */
    time_t        st_atime;    /* Time of last access */
    time_t        st_mtime;    /* Time of last modification */
    time_t        st_ctime;    /* Time of last change */
};
```

访问文件元数据示例

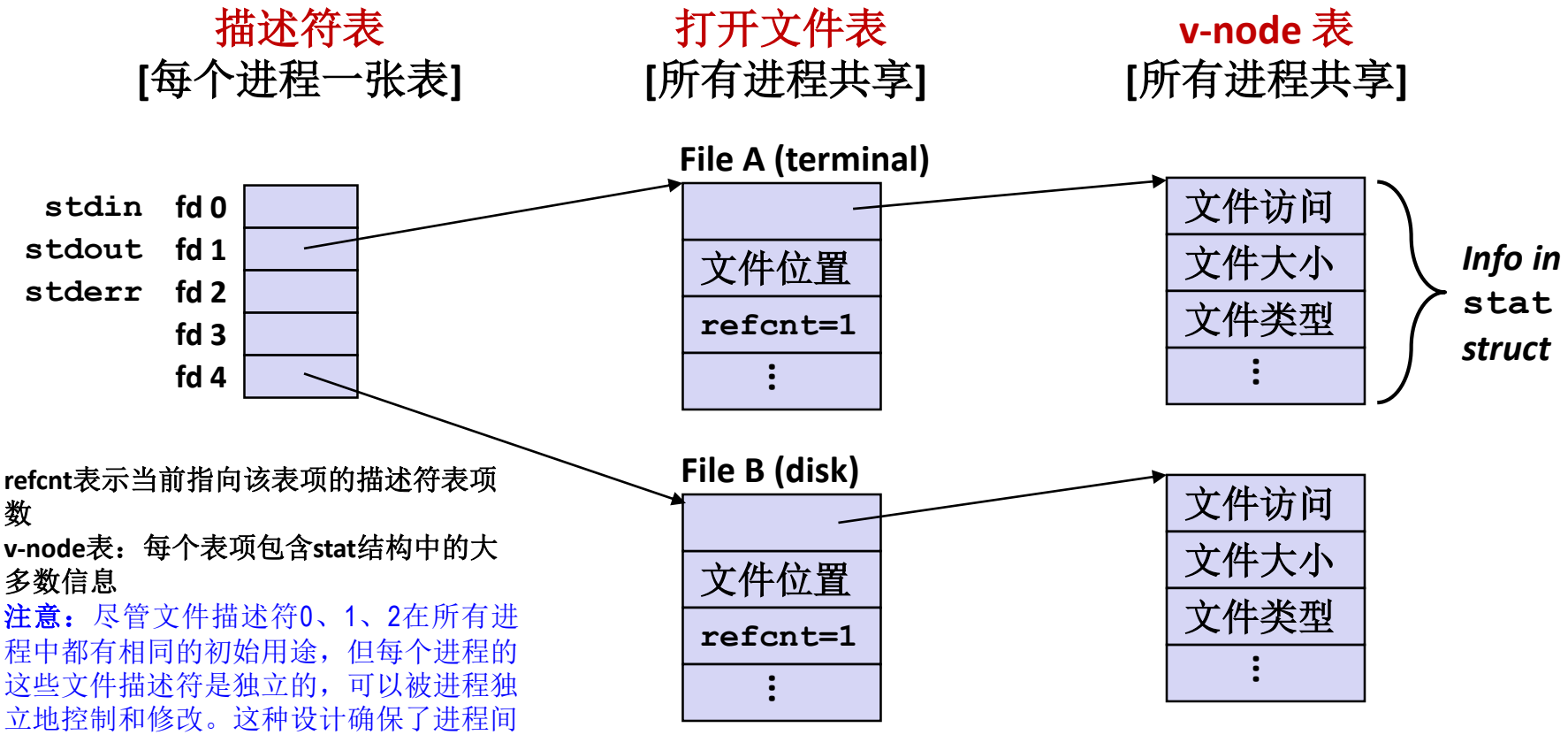
```
linux> ./statcheck statcheck.c
type: regular, read: yes
linux> chmod 000 statcheck.c
linux> ./statcheck statcheck.c
type: regular, read: no
linux> ./statcheck ..
type: directory, read: yes
```

```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

Unix内核如何表示打开文件

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



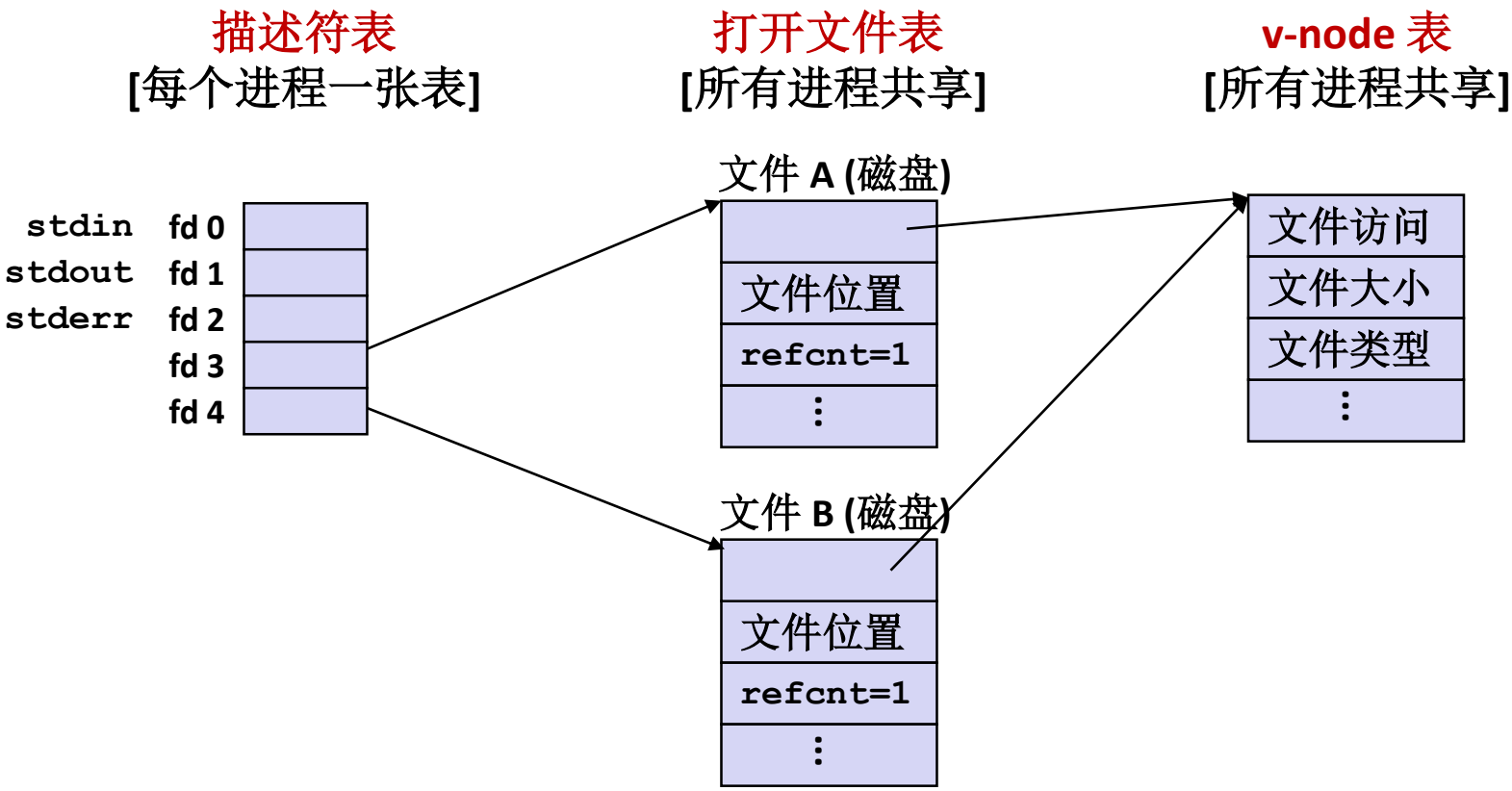
refcnt表示当前指向该表项的描述符表项数

v-node表：每个表项包含stat结构中的大多数信息

注意：尽管文件描述符0、1、2在所有进程中都有相同的初始用途，但每个进程的这些文件描述符是独立的，可以被进程独立地控制和修改。这种设计确保了进程间的隔离和灵活性。

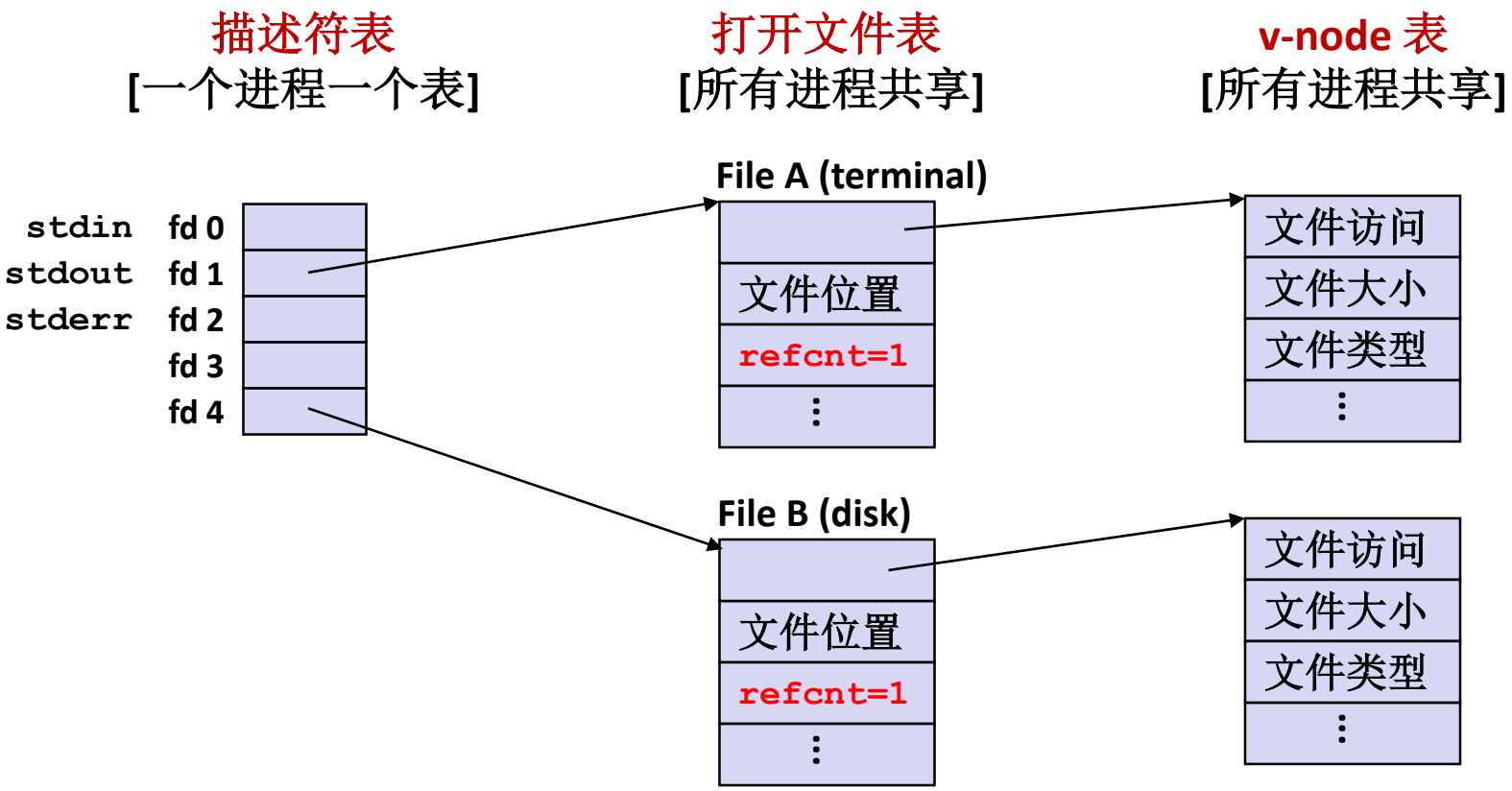
File Sharing 共享文件

- 两个不同的描述符通过两个不同的打开文件表表项来共享同一个磁盘文件
  - 例如，以同一个filename调用open函数两次



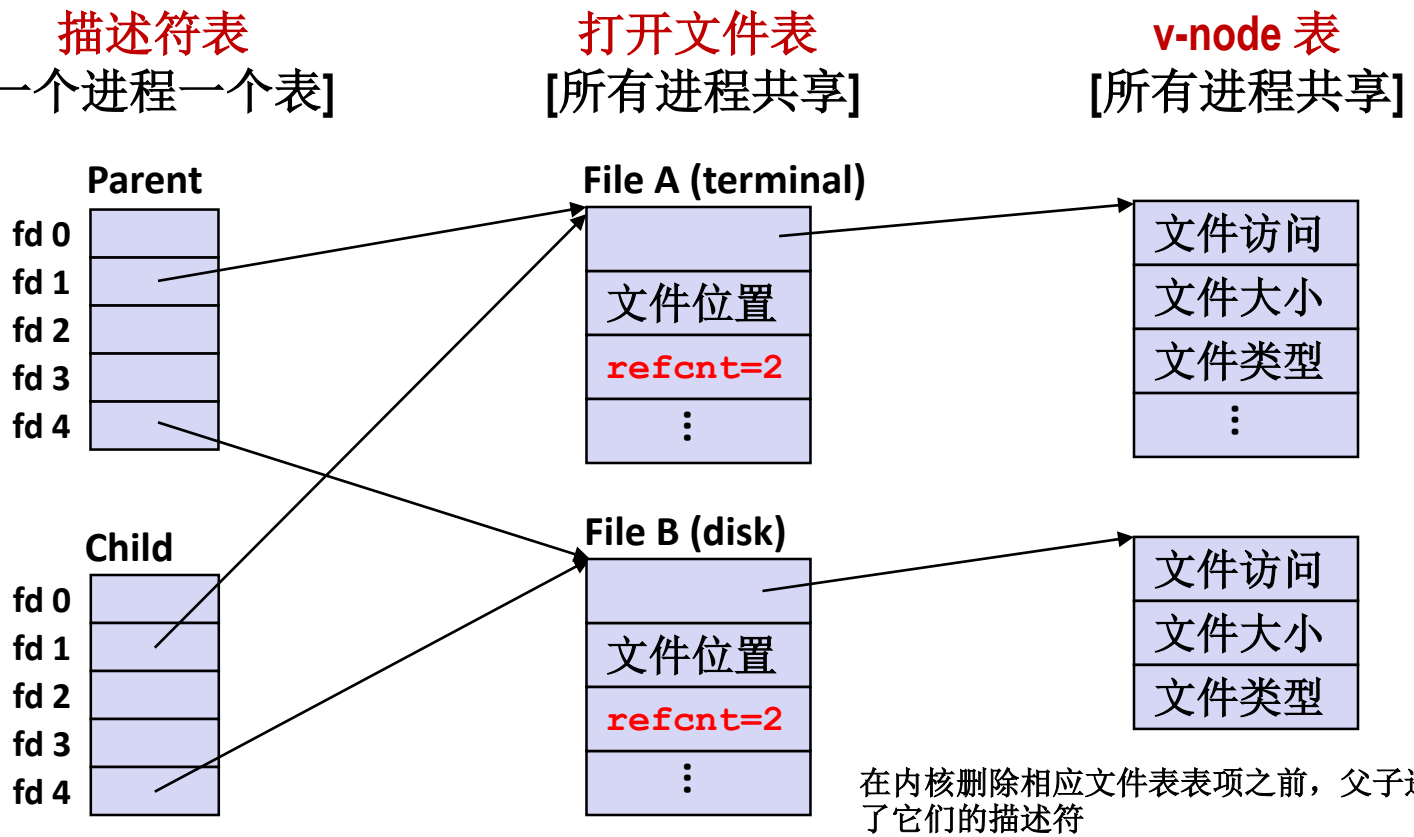
进程如何共享文件：fork

- 子进程继承父进程的打开文件
  - 注意: 共享相同的文件位置 (使用 fcntl 改变位置)
- 调用fork *之前*:



进程如何共享文件：fork（阅读教材P515）

- 子进程继承父进程的打开文件
- 调用fork *之后*:
  - 子进程的表与父进程的表相同, 每一个 refcnt +1



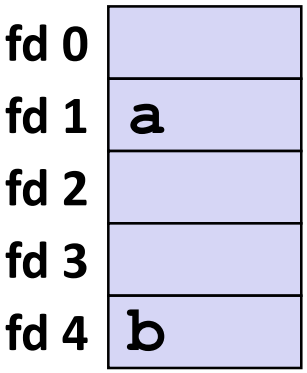
在内核删除相应文件表表项之前，父子进程必须都关闭了它们的描述符

I/O Redirection　　I/O重定向

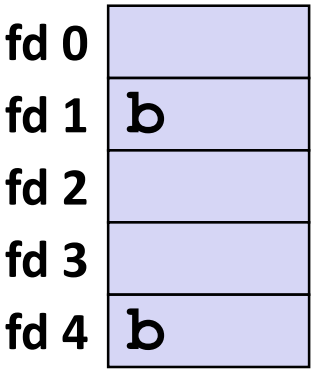
- 问题: Unix内核如何实现 I/O 重定向?
  - 允许用户将磁盘文件和标准输入输出联系起来
  - `linux>  ls > foo.txt`

- 回答: 通过调用 `dup2(oldfd, newfd)` 函数
  - 复制描述符表表项 `oldfd` 到描述符表表项 `newfd`，覆盖描述符表表项newfd以前的内容。（记住：左边描述符表项覆盖右边）

描述符表
调用 `dup2(4,1)` *之前*

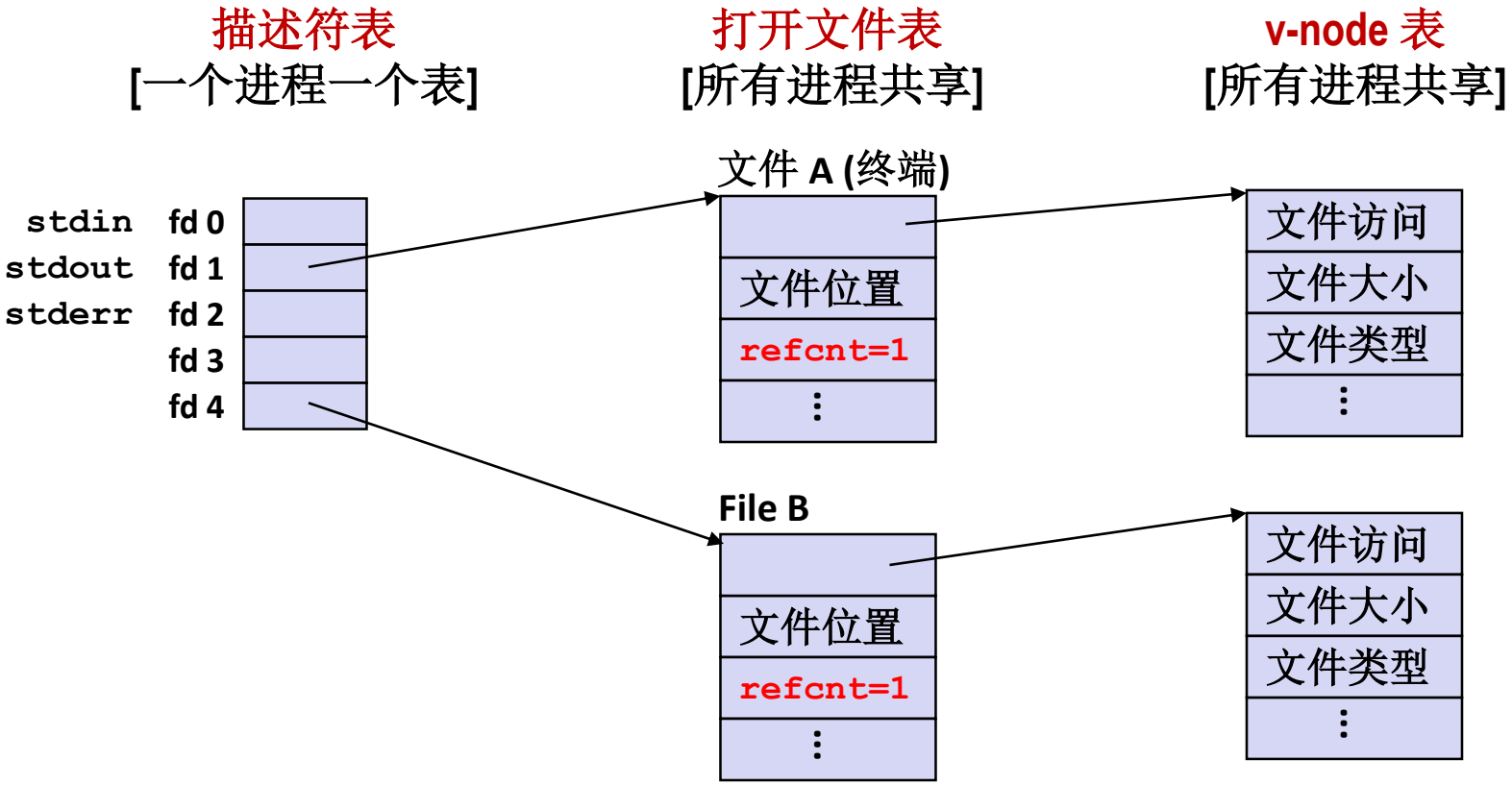


描述符表
调用 `dup2(4,1)` *之后*



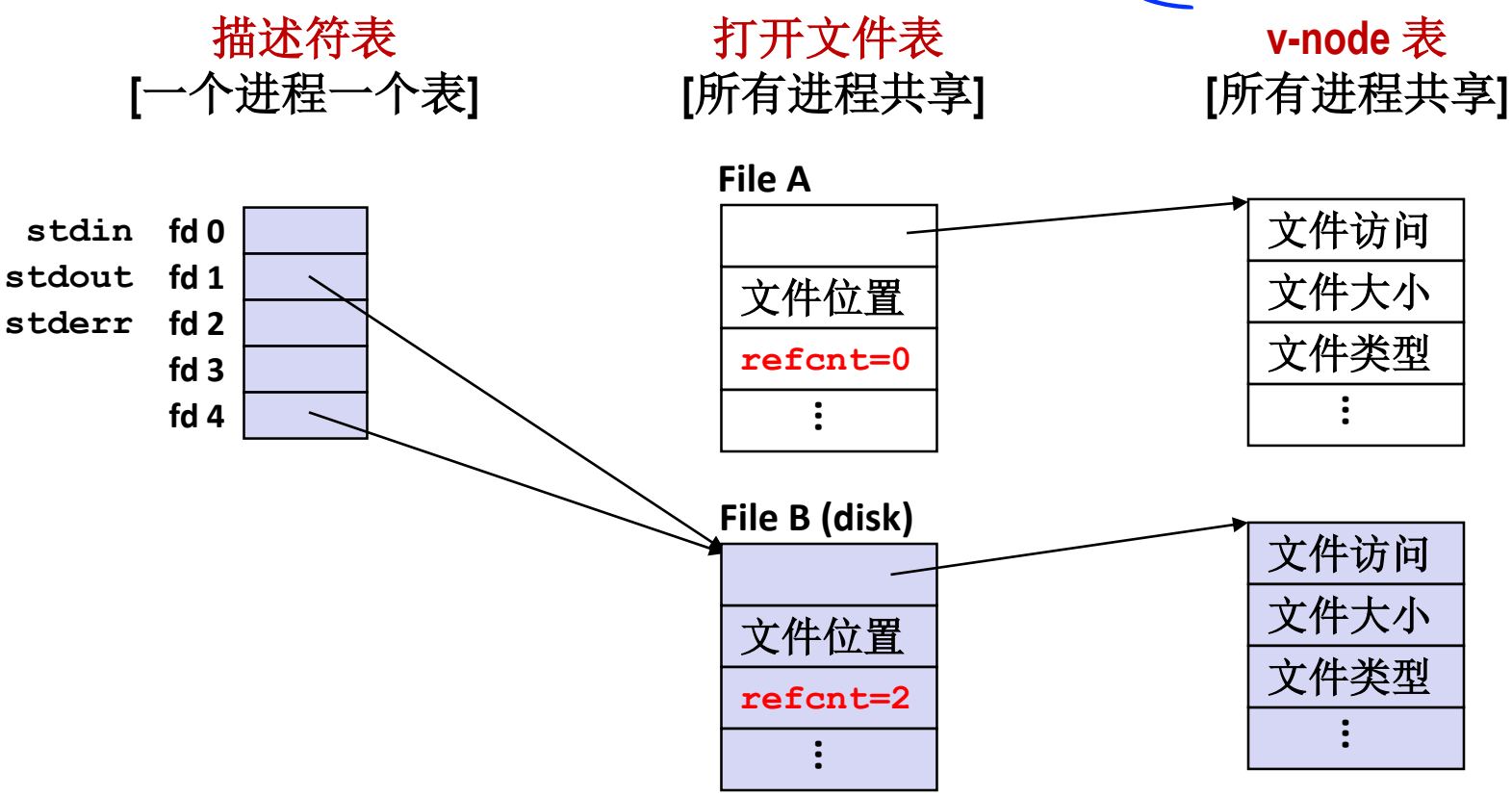
I/O Redirection Example

- 步骤 #1: 打开需重定位文件
  - 在调用dup2 (4, 1)之前



I/O Redirection Example

- 步骤 #2: 调用 `dup2(4,1)`
  - 修改描述符表项，使得 fd=1 (stdout) 指向 fd=4 所指向的磁盘文件
- 步骤 #3: 修改打开文件表项的引用计数



主要内容

- Unix I/O
- 用RIO包健壮地读写
- 读取文件元数据，共享和重定位
- **标准I/O**
- 结束语

Standard I/O Functions 标准I/O函数

- C语言定义了标准I/O库 (`libc.so`)，为程序员提供了 Unix *标准 I/O* 的较高级别的替代
  - 详见附录B中K&R的文章

- C语言的标准 I/O 函数示例:
  - 打开和关闭文件 (**`fopen`** 和 **`fclose`**)
  - 读和写字节 (**`fread`** 和 **`fwrite`**)
  - 读和写字符串 (**`fgets`** 和 **`fputs`**)
  - 格式化的读和写 (**`fscanf`** and **`fprintf`**)

Standard I/O Streams 标准I/O流

- 标准 I/O库将一个打开的文件 模型化为*流*
  - 对文件描述符和流缓冲区的抽象

- 每个C程序开始时都有三个打开的流(在stdio.h中定义)
  - **stdin** (standard input) 标准输入
  - **stdout** (standard output) 标准输出
  - **stderr** (standard error) 标准错误

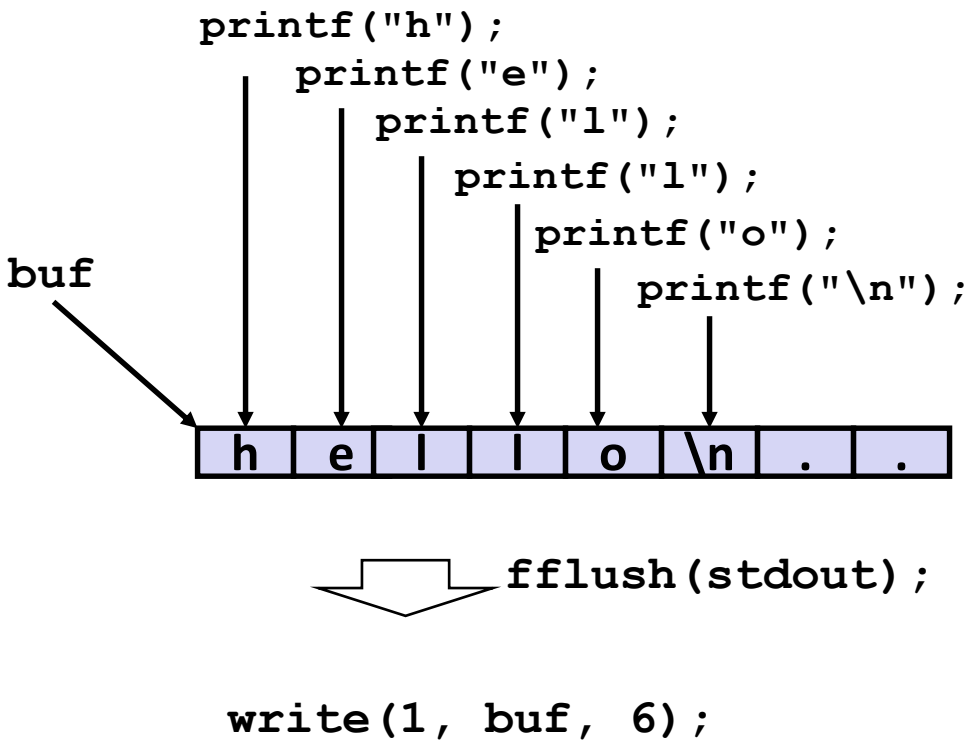
```
#include <stdio.h>
extern FILE *stdin;  /* standard input  (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error  (descriptor 2) */

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

Buffered I/O: Motivation 带缓冲I/O的动机

- 应用经常每次读/写一个字符
  - `getc, putc, ungetc`
  - `gets, fgets`
    - 每次读一行文本，到新行处停止
- 作为昂贵的 Unix I/O 调用来执行
  - 读和写需要调用 Unix 内核
    - > 10,000 时钟周期
- 解决: 带缓冲的读
  - 使用 Unix 读获取字符块
  - 用户输入函数每次从缓存取一个字节
    - 当缓存为空时重新填充

Buffering in Standard I/O 标准I/O的缓存

- 使用带缓冲的标准 I/O 函数



- 缓冲区刷新到输出 fd，当遇到 “\n”，调用fflush 或 exit，或从main返回 .

标准I/O缓冲区的作用

- 通过Linux的 `strace` 程序观察这种缓冲作用:

```
#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)                           = ?
```

为什么要用到fflush？

在 C 语言中，标准库通常使用缓冲区来提高 I/O 性能。当我们向文件写入数据时，数据首先被写入到缓冲区中，然后在适当的时候才被写入到文件中。如果我们没有调用 fflush，则**缓冲区中的数据可能会一直保留，直到程序结束或缓冲区满了才被写入到文件中**。这可能会导致数据丢失或不一致的情况。具体来说，如果我们向文件写入数据，但没有调用 fflush，则数据可能会一直保留在缓冲区中，直到程序结束或缓冲区满了才被写入到文件中。如果程序在数据被写入文件之前崩溃或意外终止，则数据可能会丢失。此外，如果程序在写入数据之前读取文件，则可能会读取到旧的数据，因为新的数据尚未被写入文件中。因此，为了确保数据被写入文件中，在写入数据后调用 fflush 来刷新缓冲区，以便将数据写入文件中。

fflush主要做的是借助系统调用write将_IO_write_ptr和_IO_write_base间的数据写入内核这个缓冲区（由C库维护），是实现在用户内存空间的。写流使用的是write系统调用，实际上进行一次系统调用的开销是很大的，要根据实际情况来选择使用，可以通过setbuf系列库函数来设置缓冲区或者禁用缓冲区。

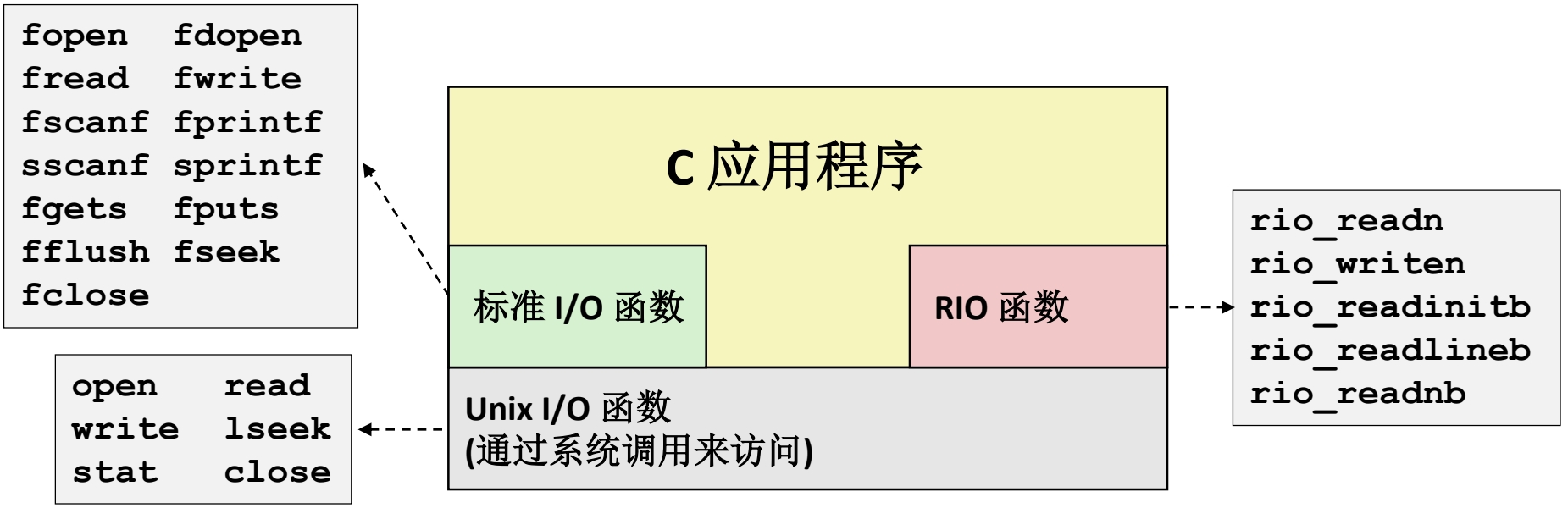
关于fflush请查看
https://yebd1h.smartapps.cn/pages/blog/index?blogId=129676454&_swebfr=1&_swebFromHost=hwquickapp

主要内容

- Unix I/O
- 用RIO包健壮地读写
- 读取文件元数据，共享和重定位
- 标准I/O
- 结束语

Unix I/O 、标准 I/O 和 RIO之间的关系

- 标准 I/O 和 RIO 是基于较低级的 Unix I/O 函数来实现的。



- 我该使用哪些I/O函数?

Unix I/O优点和缺点

- 优点
  - Unix I/O 是最通用、开销最低的I/O方式
    - 所有其他 I/O都是使用Unix I/O 函数来实现的
  - Unix I/O 提供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地使用
- 缺点
  - 处理不足值时容易出错
  - 有效地读取文本行需要某种形式的缓冲, 容易出错
  - 这两个问题都是由标准I/O和RIO包来解决

标准I/O的优点和缺点

- 优点:
  - 通过减少读和写系统调用的次数，从而减少间接带来的内存占用
  - 自动处理不足值
- 缺点:
  - 没有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, 不适合用于信号处理
  - 标准 I/O 不适合网络套接字的输入输出操作
    - 对流的限制和对套接字的限制有时候会互相冲突，而又很少有文档描述这些现象(CS:APP3e, Sec 10.11)

I/O函数的选择（详见书P639）

- **一般规则：使用最高级别的I/O函数**
  - 大多数 C 程序员在其整个职业生涯中只使用标准 I/O
  - 但是,他一定明白你所使用的函数!
- **什么时候使用标准 I/O**
  - 当使用磁盘文件和终端文件时
- **什么时候使用 Unix I/O**
  - 在信号处理程序中, 因为 Unix I/O 是异步信号安全的
  - 在极少数情况下，当你需要绝对最高的性能时
- **什么时候使用 RIO**
  - 当你准备读、写网络套接字时
  - 避免在套接字上使用标准I/O

习题

- 1. 若将标准输出重定向到文本文件file.txt，错误的是：
  - A. 需要先打开重定位的目标文件“file.txt”
  - B. 设“file.txt”对应的fd为4，内核调用dup2(1, 4)函数实现描述符表项的复制
  - C. 复制“file.txt”的打开文件表项，并修正fd为1的描述符
  - D. 修改“file.txt”的打开文件表项的引用计数

- B、C

解析：B选项正确的写法应该是dup2(4,1)，左边描述符表项覆盖右边。C选项错在最后“描述符”，因为“描述符”是不能随意修改的，而重定向的本质是描述符表项的复制。

Hope you enjoyed the CSAPP course!