第10章 系统级I/O

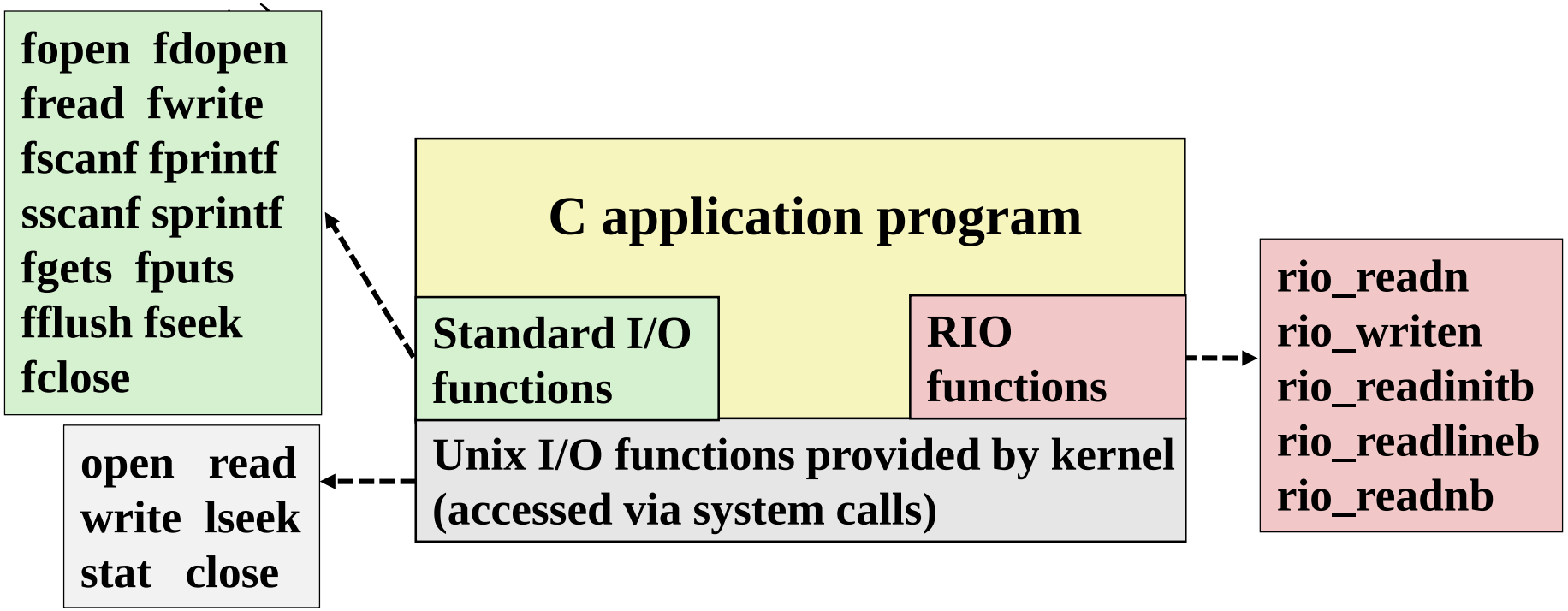
教　师：　郑贵滨
计算机科学与技术学院
哈尔滨工业大学

主要内容

- **Unix I/O**
- 元数据、共享和重定位
- 标准I/O
- RIO,健壮的I/O程序包(robust I/O)
- 结束语

主题: Unix I/O、C Standard I/O、RIO

- 两个集合:系统级、C语言级
- Robust I/O (RIO): 本课程的专门封装
- 好的编程做法: 处理错误检查、信号和“不足值”(short



Unix I/O 概述

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

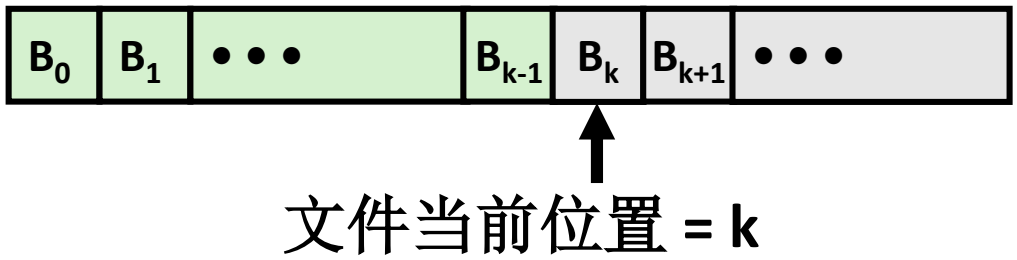
- Cool fact: 所有的I/O设备（网络、磁盘、终端）都被模型化为文件:
  - **/dev/sda2**（用户磁盘分区）
  - **/dev/tty2**（终端）

- 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）

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、关闭文件
    - **open()、close()**
  - 读、写文件
    - **read() 、 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()**

是异步信号安全的函数！



文件类型(File Types)

- 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): 包含一组链接的文件，每个链接都将一个文件名映射到一个文件/
  - 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
  - 命名通道（Named pipes (FIFOs)）
  - 符号链接（Symbolic links）
  - 字符和块设备（Character and block devices）

普通文件（Regular Files ）

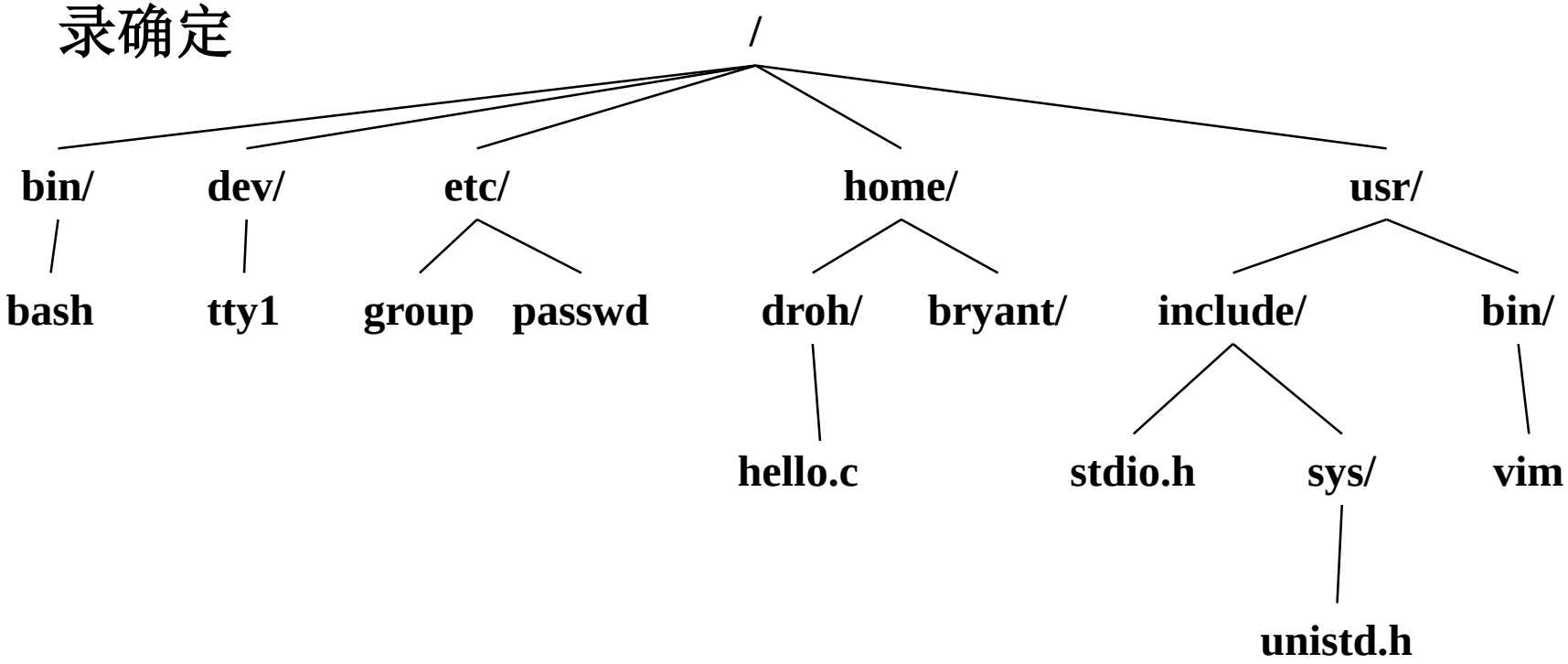
- 普通文件包含任意数据
- 应用程序常常要区分文本文件（*text files*）*和二进制文件*（*binary files*）
  - 文本文件：只包含 ASCII 或 Unicode字符的普通文件
  - 其他文件：二进制文件，如：目标文件, JPEG图像文件等等
  - 内核并不知道两者之间的区别
- Linux文本文件是文本行的序列
  - 文本行是一个字符序列，以一个新行符 (‘\n’)结束
    - 新行符为 0x0a，与 ASCII 的换行符 (LF) 是一样的
- 其他系统中的行结束标志
  - Linux和 Mac 操作系统 : '\n'(0x0a)
    - 换行 (LF)
  - Windows 和 因特网络协议: '\r\n'(0x0d 0x0a)
    - Carriage return (CR) followed by line feed (LF)
      回车换行

目录（Directories）

- 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

目录层次结构(Directory Hierarchy)

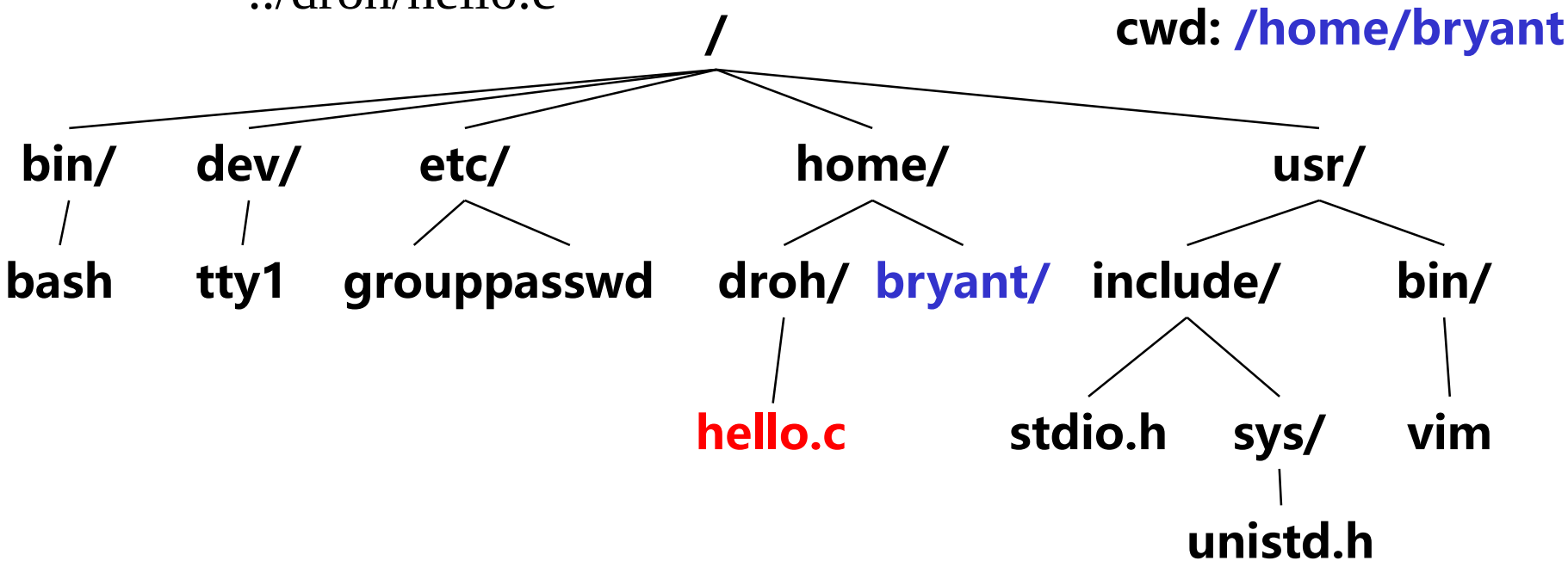
- 所有文件都组织成一个目录层次结构，由名为 / (斜杠)的根目录确定



- 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

路径名（Pathnames）

- 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.开头*，表示从当前工作目录开始的路径
    - ../droh/hello.c



开打文件

- 打开文件：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)

打开文件——细节

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

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

Returns: new file descriptor if OK, -1 on error

flags

- **O_RDONLY, O_WRONLY, O_RDWR (must have one)**
- **O_CREAT, O_TRUNC, O_APPEND (optional)**

mode

- **S_IRUSR, S_IWUSR, S_IXUSR**
- **S_IRGRP, S_IWGRP, S_IXGRP**
- **S_IROTH, S_IWOTH, S_IXOTH**

打开文件——细节

umask():

新建文件、新建目录都会有默认权限，例如：文件权限为-rw-rw-r--、目录权限为drwxrwxr-x。而umask的值则表明从当前进程的默认权限中去掉哪些权限，做为最终的默认权限值。

```
#define DEF_MODE    S_IRUSR | S_IWUSR |      \
                S_IRGRP | S_IWGRP |\
                   S_IROTH | S_IWOTH
#define DEF_UMASK    S_IWGRP | S_IWOTH
// IWGRP 是用户组用户的写权限,IWOTH 是非所有者或用户组用户的写权限

umask(DEF_UMASK);
fd = open ("foot.txt",
                O_CREAT|O_TRUNC|O_WRONLY, DEF_MODE);
/* permission: S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH*/
```

关闭文件

- 通知内核不再访问文件
- 关闭文件时，内核的操作：
  - 内核释放文件打开时创建的结构体
  - 内核将描述符释放给可用描述符池
    - 下次打开某个文件时，从该池中分配一个最小可用的描述符
- 进程终止时，内核的操作
  - 内核关闭所有打开的文件
  - 内核释放他们占用的内存资源

```
#include <unistd.h>
int close(int fd) ;
```

Returns: zero if OK, -1 on error

关闭文件

- 关闭一个已经关闭的文件，会导致程序(线程) 灾难
- 建议：总是检查返回码。

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

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

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!

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 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，并非错误!

简单的 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);
}
```

```
ssize_t Read(int fd, void *buf, size_t count) {
    ssize_t rc;
    if ( (rc = read(fd, buf, count)) < 0 )
            unix_error("Read error");
    return rc;
}
```

不足值(Short Counts)

- 出现“不足值”的几种情况:
  - 读时遇到EOF
  - 从终端读文本行
  - 读写网络套接字
  - 读磁盘文件 时，遇到**EOF**
  - 写磁盘文件时，**磁盘满或空间不足**
- 最好的做法就是始终考虑不足值

主要内容

- Unix I/O
- 元数据、共享和重定位
- 标准I/O
- RIO,健壮的I/O程序包(robust I/O)
- 结束语

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

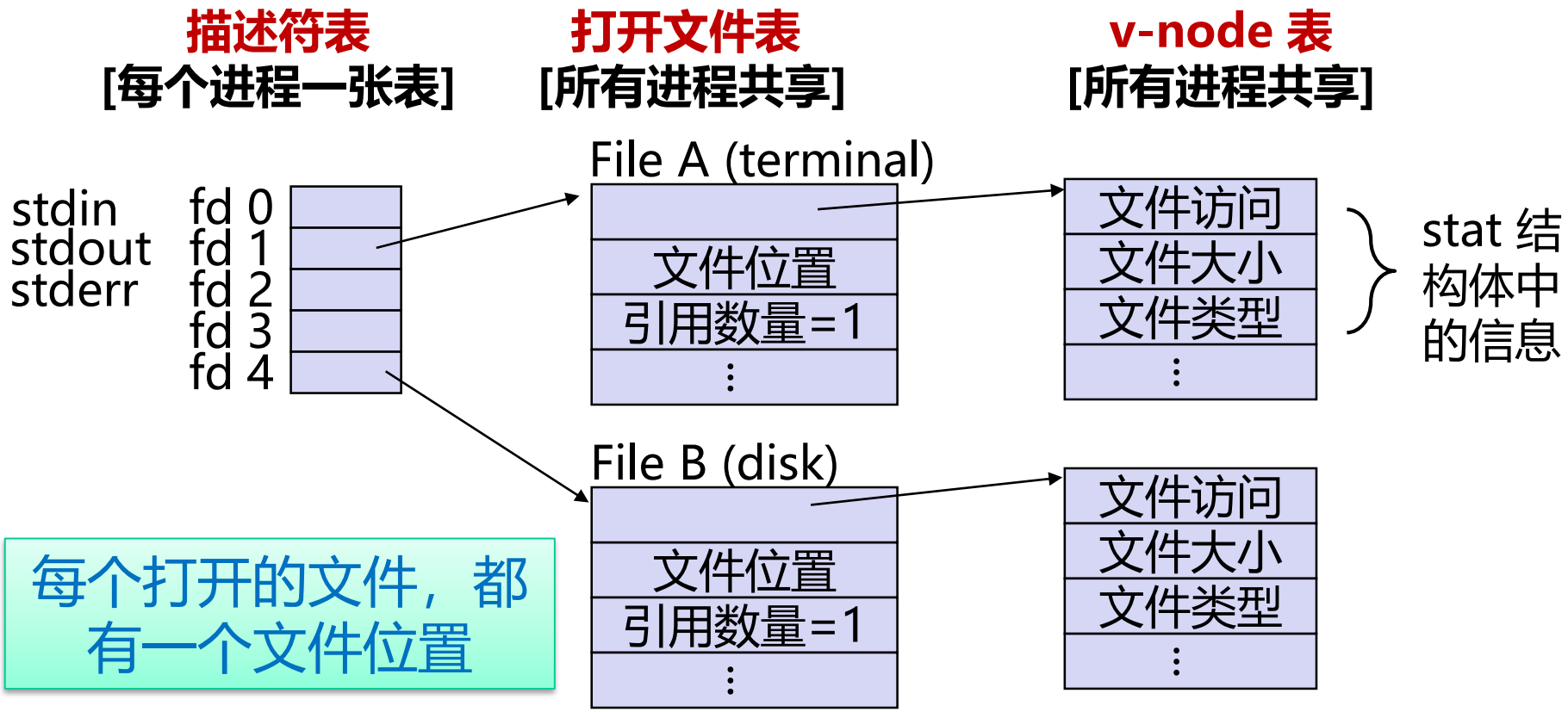
```
int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

    Stat(argv[1], &stat);// stat(filename, buf)
    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
```

Unix内核如何表示打开文件

- 两个运算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 指向打开磁盘文件



Unix内核如何表示打开文件

```
struct task_struct {
        /* Open file information: */
        struct files_struct     *files;
```

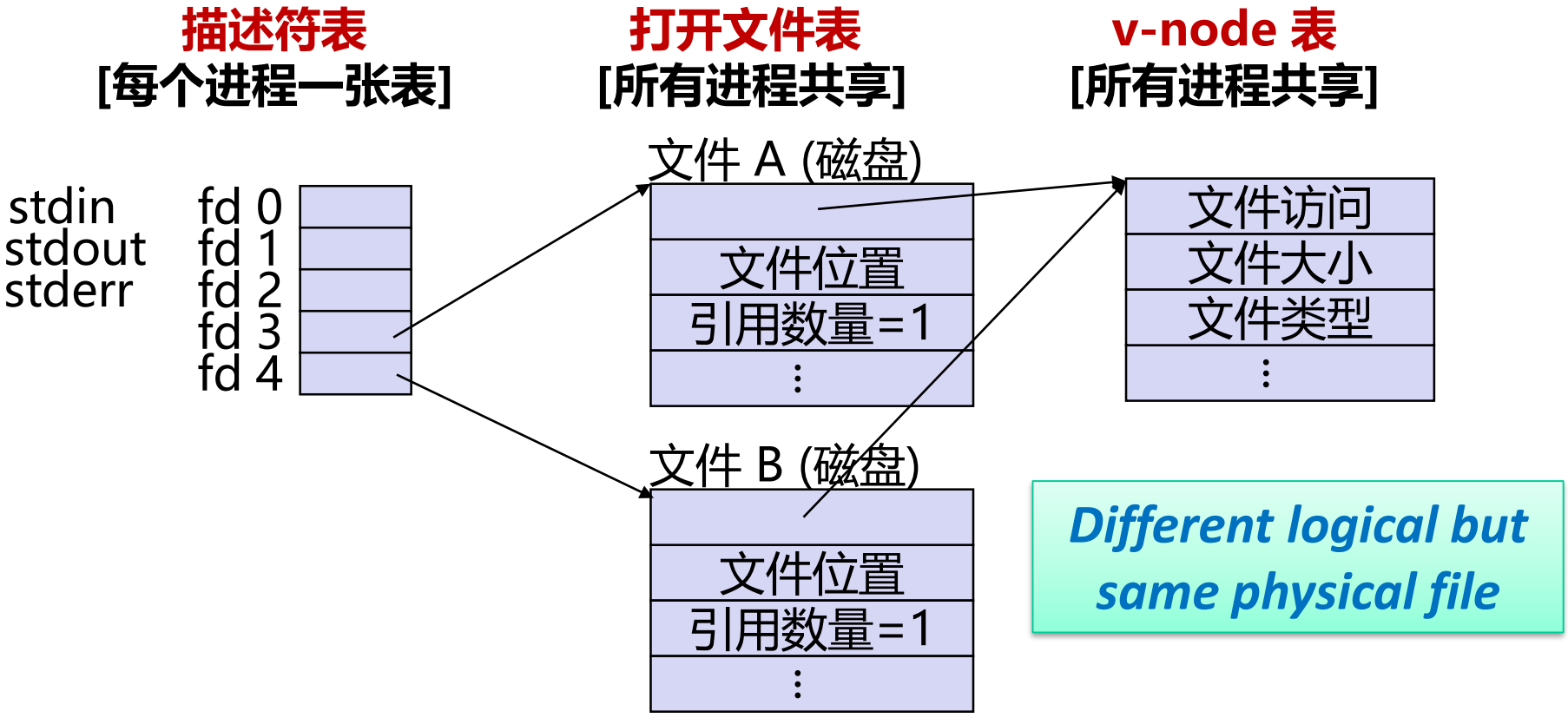
/usr/src/linux-headers-***/include/linux /sched.h Line:815

```
1.struct files_struct {
2.    atomic_t        count; //自动增量
3.    struct fdtable  *fdt;
4.    struct fdtable  fdtab;
5.    fd_set      close_on_exec_init;  //执行exec时需要关闭的文件描述符初值集合
6.    fd_set      open_fds_init;       //当前打开文件 的文件描述符屏蔽字
7.    struct file     * fd_array[NR_OPEN_DEFAULT];
                                  /*Linux中一个进程最多只能同时打开
                                    NR_OPEN_DEFAULT个文件*/
8.    spinlock_t      file_lock;  /* Protects concurrent writers.
                                   Nests inside tsk->alloc_lock */
9.};
```

```
/* /usr/src/linux-headers-4.15.0-39-generic/include/linux/fs.h Line 852*/
struct file {
    union {
        struct llist_node fu_llist;
        struct rcu_head  fu_rcuhead;
    } f_u;
    struct path             f_path;
    struct inode            *f_inode;       /* cached value */
    const struct file_operations    *f_op;

    /*
     * Protects f_ep_links, f_flags.
     * Must not be taken from IRQ context.
     */
    spinlock_t              f_lock;
    enum rw_hint                    f_write_hint;
    atomic_long_t                   f_count; //当前有多少个进程在使用该文件
    unsigned int                    f_flags;
    fmode_t                 f_mode;
    struct mutex                    f_pos_lock;
    loff_t                  f_pos;
    struct fown_struct      f_owner;
    const struct cred       *f_cred;
    struct file_ra_state    f_ra;
...
```

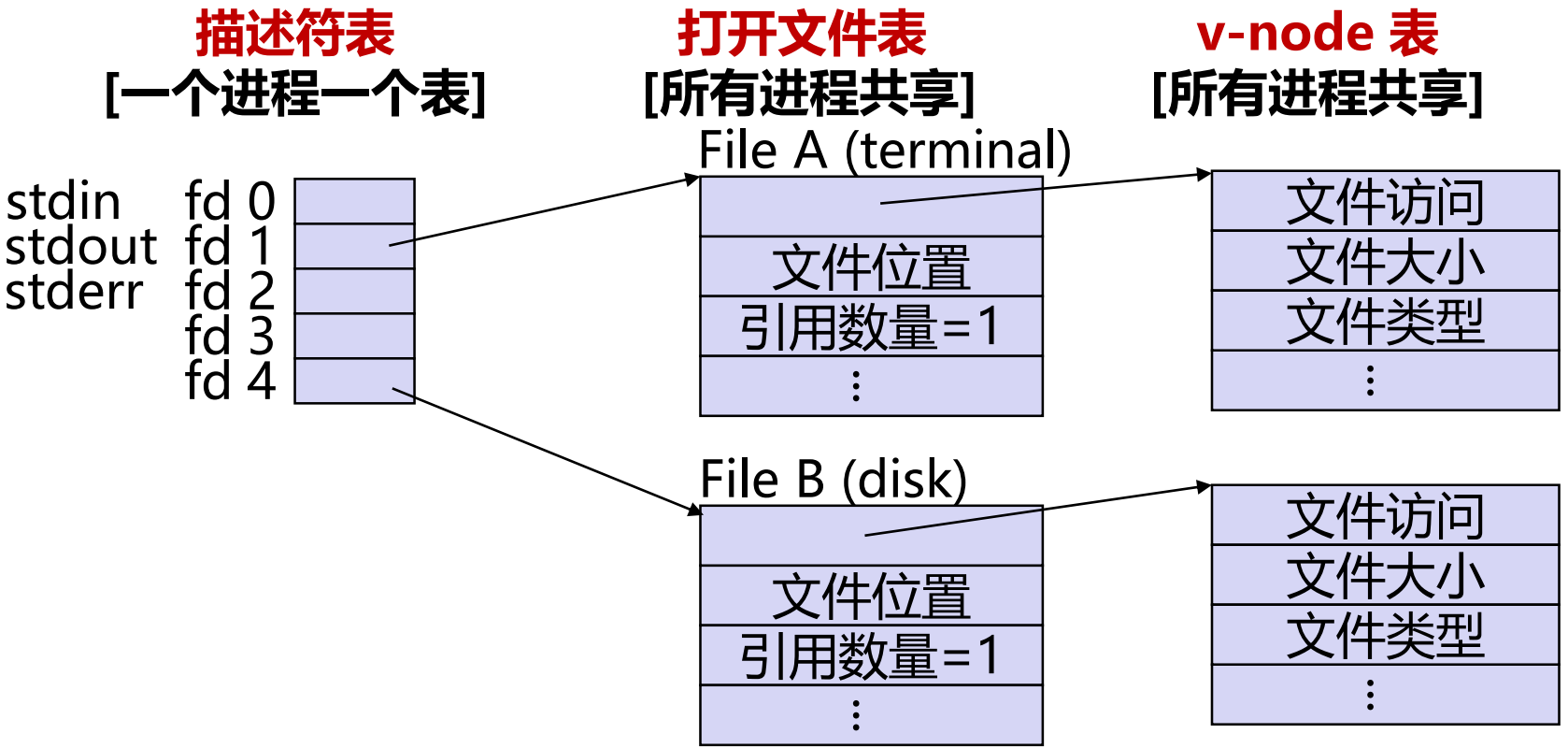
共享文件

- 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v-node表项（磁盘文件）
  - 例如，调用open函数对同一个filename打开两次



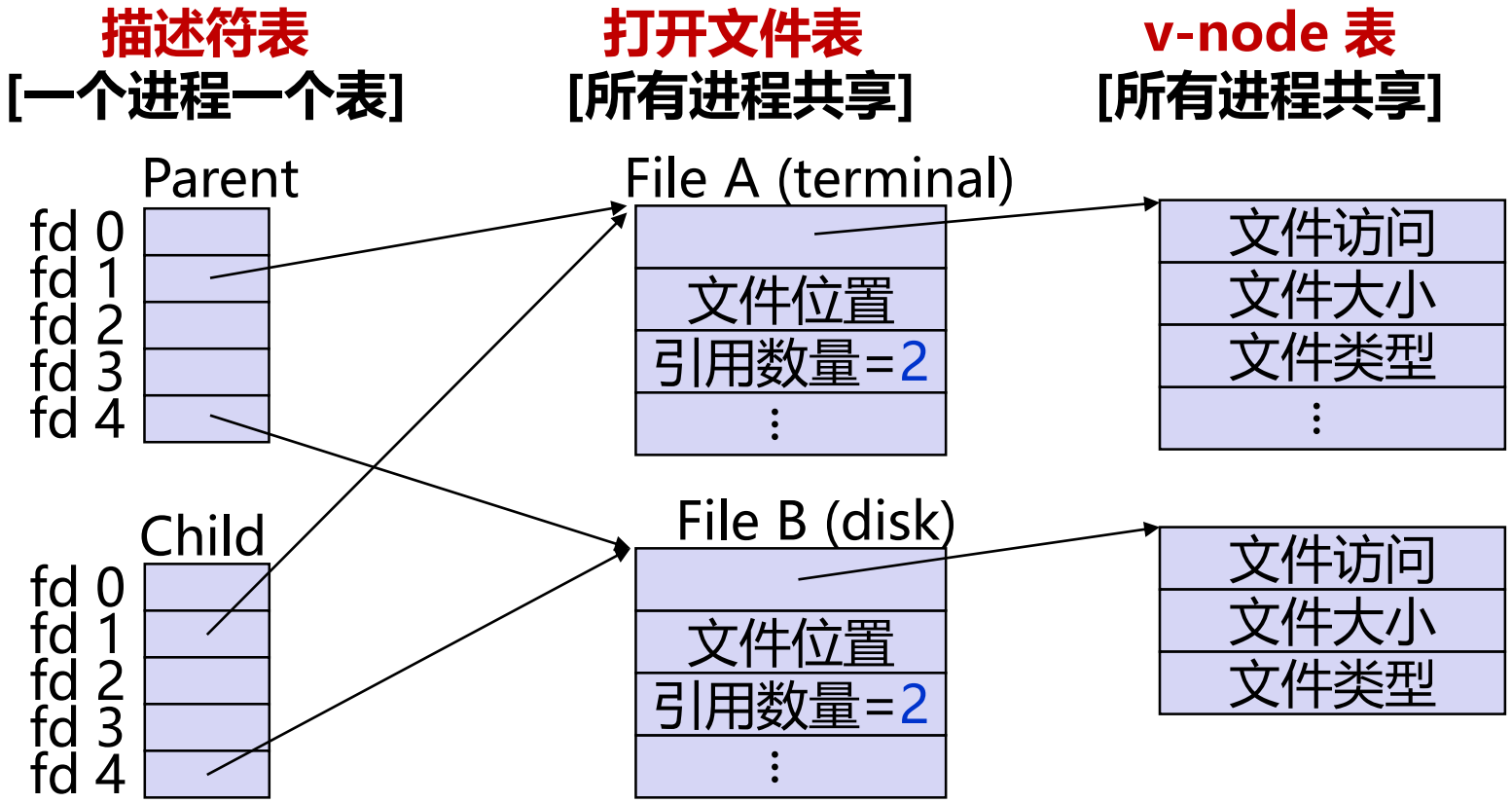
进程如何共享文件：fork

- **子进程继承父进程的打开文件**
  - exec函数不改变文件情况，共享相同的文件位置
  - 使用fcntl 函数可以改变打开文件的各种属性，如位置等
- **调用fork 之前:**



进程如何共享文件：fork

- 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

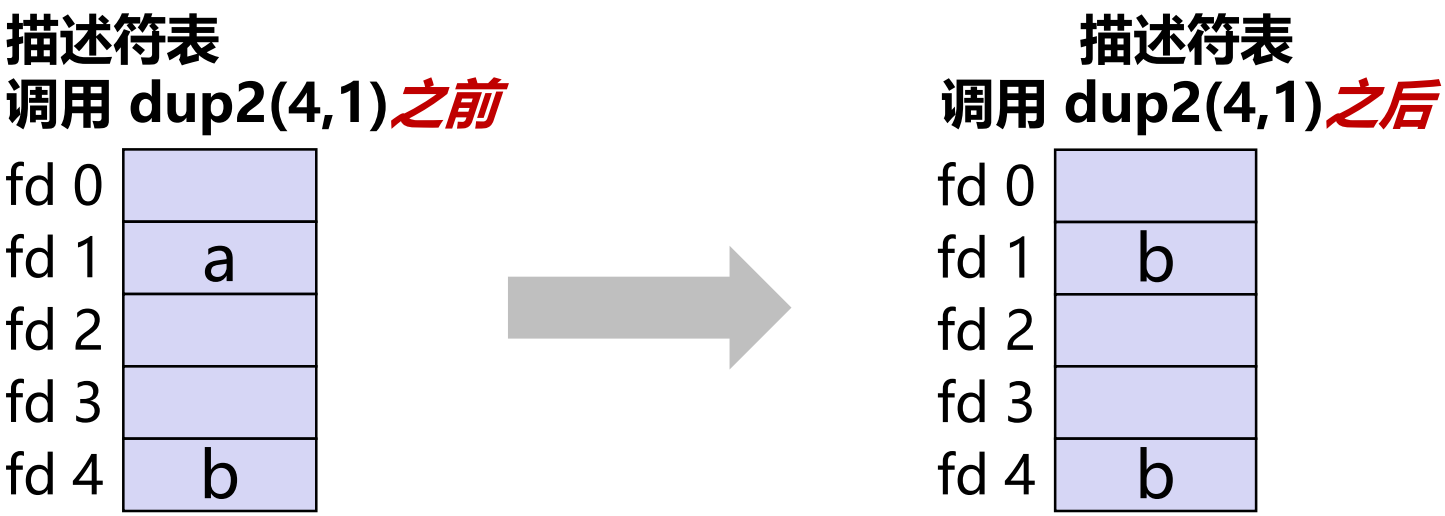


I/O重定向

- **问题: Unix内核如何实现 I/O 重定向?**

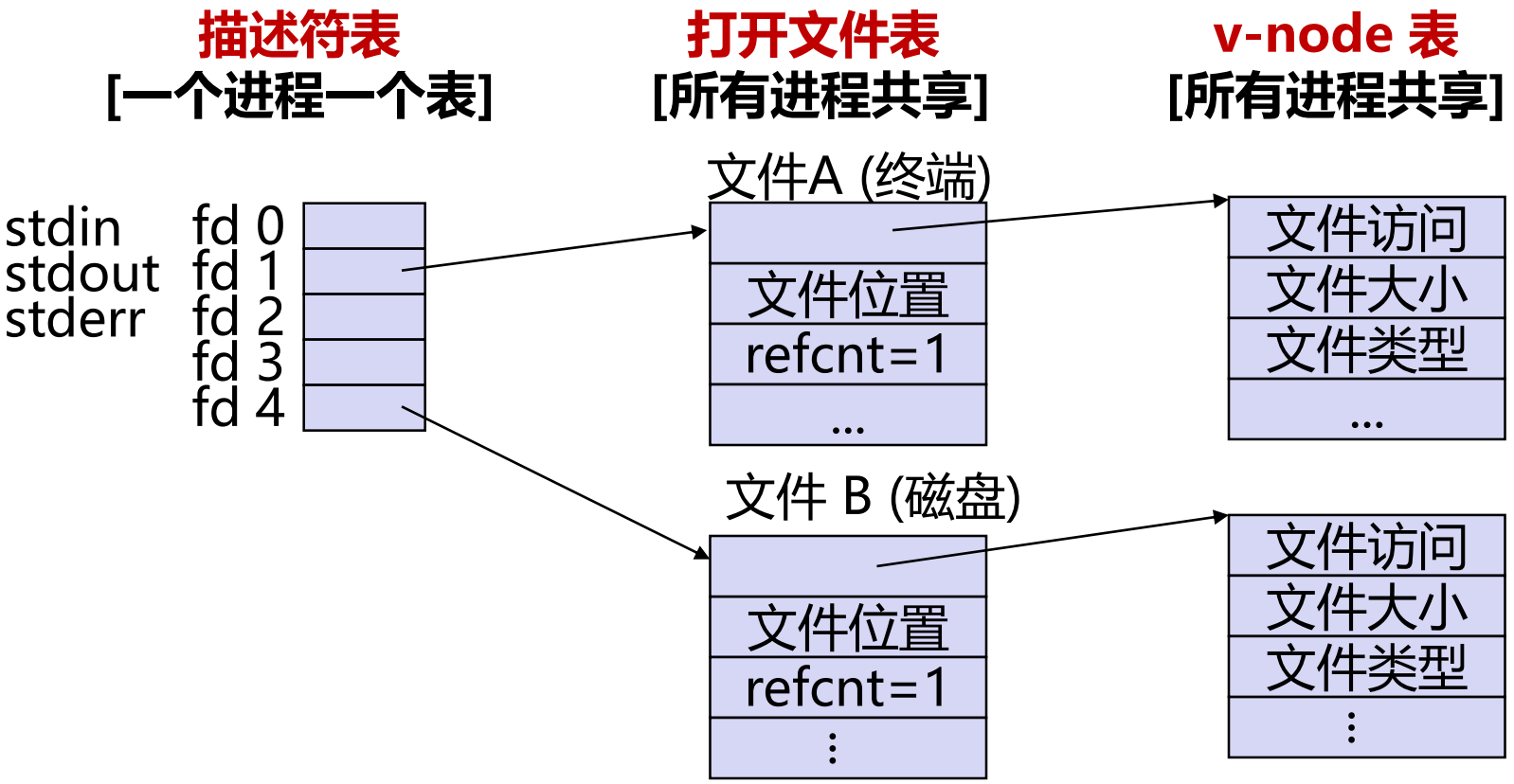
  linux> ls > foo.txt

- **回答: 调用 dup2(srcfd, dstfd)函数**
  - 复制描述符表表项 **srcfd** 到描述符表表项 **dstfd**，覆盖描述符表表项dstfd以前的内容。



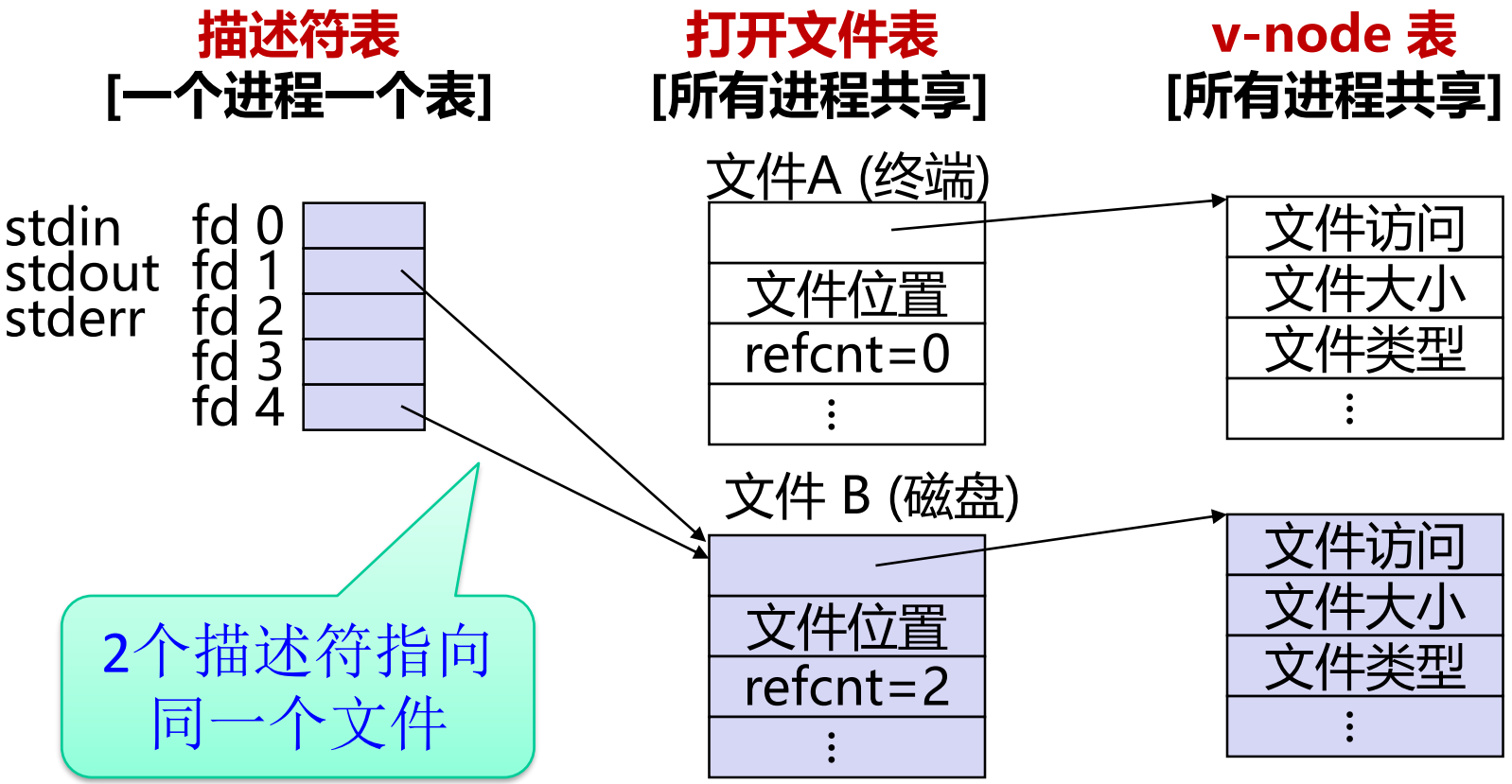
I/O重定向示例

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



I/O重定向示例(cont.)

- **步骤 #2: 调用 dup2(4,1)**
  - 使原本指向stdout的fd=1，指向了fd=4所指向的磁盘文件



文件描述符趣事(1)：I/O 重定向

```
#include "csapp.h"
int main(int argc, char *argv[]){
    int fd1, fd2, fd3;
    char c1, c2, c3;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    fd2 = Open(fname, O_RDONLY, 0);
    fd3 = Open(fname, O_RDONLY, 0);
    Dup2(fd2, fd3);
    Read(fd1, &c1, 1);
    Read(fd2, &c2, 1);
    Read(fd3, &c3, 1);
    printf("c1 = %c, c2 = %c, c3 = %c\n", c1, c2, c3);
    return 0;
}
```

ffiles1.c

- 如文件内容是"abcde"，程序的输出是?

文件描述符趣事(1)：I/O 重定向

```
#include "csapp.h"
int main(int argc, char *argv[]){
    int fd1, fd2, fd3;
    char c1, c2, c3;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    fd2 = Open(fname, O_RDONLY, 0);
    fd3 = Open(fname, O_RDONLY, 0);
    Dup2(fd2, fd3);
    Read(fd1, &c1, 1);
    Read(fd2, &c2, 1);
    Read(fd3, &c3, 1);
    printf("c1 = %c, c2 = %c, c3 = %c\n", c1, c2, c3);
    return 0;
}
```

dup2(oldfd, newfd)

ffiles1.c

- 如文件内容是"abcde"，程序的输出是? c1 = a, c2 = a, c3 = b

文件描述符趣事(2)：进程控制和I/O

```
#include "csapp.h"
int main(int argc, char *argv[]){
    int fd1;
    int s = getpid() & 0x1;
    char c1, c2;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    Read(fd1, &c1, 1);
    if(fork()){   /*Parent*/
        sleep(s);
        Read(fd1, &c2, 1);
        printf("Parent: c1 = %c, c2 = %c\n", c1, c2);
    } else {      /*Child*/
        sleep( 1 - s );
        Read(fd1, &c2, 1);
        printf("Child: c1 = %c, c2 = %c\n", c1, c2);
    }
    return 0;
}                                                    //ffiles2.c
```

- 如文件内容是"abcde"，程序的输出是?

文件描述符趣事(2)：进程控制和I/O

```
#include "csapp.h"
int main(int argc, char *argv[]){
    int fd1;
    int s = getpid() & 0x1;
    char c1, c2;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    Read(fd1, &c1, 1);
    if (fork()) { /* Parent */
        sleep(s);
        Read(fd1, &c2, 1);
        printf("Parent: c1 = %c, c2 = %c\n", c1, c2);
    } else { /* Child */
        sleep(1-s);
        Read(fd1, &c2, 1);
        printf("Child: c1 = %c, c2 = %c\n", c1, c2);
    }
    return 0;
}                                                    //ffiles2.c
```

Child: c1 = a, c2 = b
Parent: c1 = a, c2 = c

Parent: c1 = a, c2 = b
Child: c1 = a, c2 = c

走哪条路径？

- 如文件内容是"abcde"，程序的输出是?

文件描述符趣事(3)

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char *fname = argv[1];
    fd1 = Open(fname, O_CREAT|O_TRUNC|O_RDWR, S_IRUSR|S_IWUSR);
    Write(fd1, "pqrs", 4);
    fd3 = Open(fname, O_APPEND|O_WRONLY, 0);
    Write(fd3, "jklmn", 5);
    fd2 = dup(fd1);  /* Allocates descriptor */
    Write(fd2, "wxyz", 4);
    Write(fd3, "ef", 2);
    return 0;
}                                          //ffiles3.c
```

- 结果文件的内容是？

  pqrswxyznef

主要内容

- Unix I/O
- 元数据、共享和重定位
- RIO，健壮的I/O程序包(robust I/O)
- 标准I/O
- 结束语

RIO包(The RIO Package)

- RIO 是一个封装集，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
  - 无缓冲的二进制数据输入输出函数
    - **rio_readn**和 **rio_writen**
  - 带缓冲的输入函数(文本行、二进制数据)
    - **rio_readlineb** 和 **rio_readnb**
    - 带缓冲的 RIO 函数是线程安全的，可在同一个描述符上任意地交错调用
- 下载地址： http://csapp.cs.cmu.edu/3e/code.html
  → **src/csapp.c** and **include/csapp.h**

无缓存的RIO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
    - 能确定要读取的字节数时使用它
  - 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的实现

```
/* rio_readn - Robustly read n bytes (unbuffered) */
1  ssize_t rio_readn(int fd, void *usrbuf, size_t n) {
2    size_t nleft = n;
3    ssize_t nread;
4    char *bufp = usrbuf;
5    while (nleft > 0) {
6          if ((nread = read(fd, bufp, nleft)) < 0) {
7             if (errno == EINTR) /* Interrupted by sig handler return */
8                   nread = 0;      /* and call read() again */
9             else
10                  return -1;      /* errno set by read() */
11         }
12         else if (nread == 0)
13            break;              /* EOF */
14         nleft -= nread;
15         bufp += nread;
16   }
17   return (n - nleft);         /* Return >= 0 */
18 }
```

//csapp.c

rio_ writen的实现

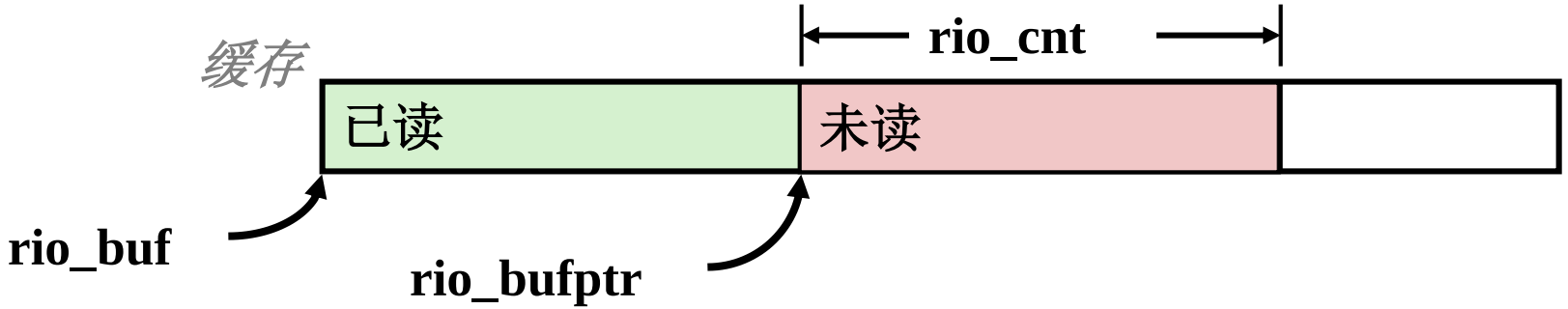
```
1  ssize_t rio_writen(int fd, const void *buf, size_t count)
2  {
3          size_t nleft = count;
4          ssize_t nwritten;
5          const char *ptr = buf;
6
7          while (nleft > 0) {
8              if ((nwritten = write(fd, ptr, nleft)) <= 0) {
9                  if (errno == EINTR)
10                     nwritten = 0; /* and call write() again */
11                 else
12                     return -1; /* errorno set by write() */
13             }
14             nleft -= nwritten;
15             ptr += nwritten;
16         }
17         return count - nleft;
18 }
```

带缓存的I/O——动机

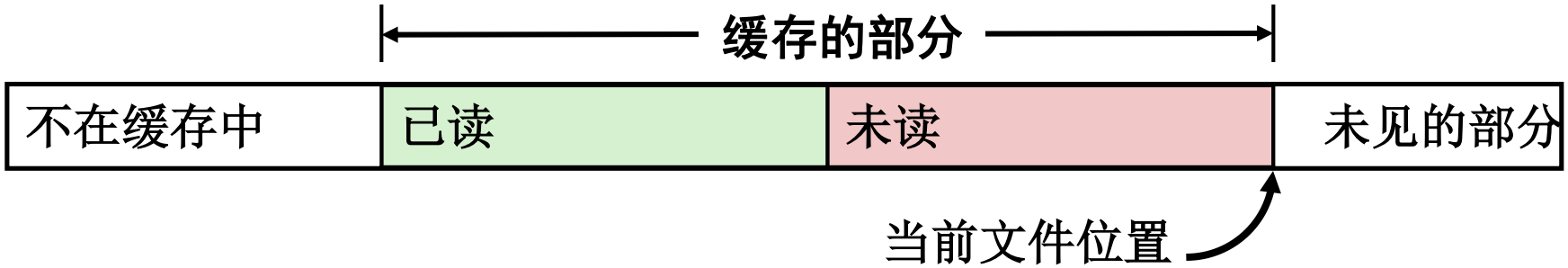
- 应用程序经常一次读/写1个字符
  - **getc, putc, ungetc**
  - **gets, fgets :** 读取一个文本行，到新行处停止
- 作为Unix I/O调用实现成本高
  - **read**和**write**需要调用 Unix 内核: > 10,000 时钟周期
- 解决办法: 带缓存的读
  - 使用 Unix **read**获取字节块
  - 用户的输入函数每次从缓存中取一个字节
    - 当缓存为空时重新填充

带缓存的I/O——实现

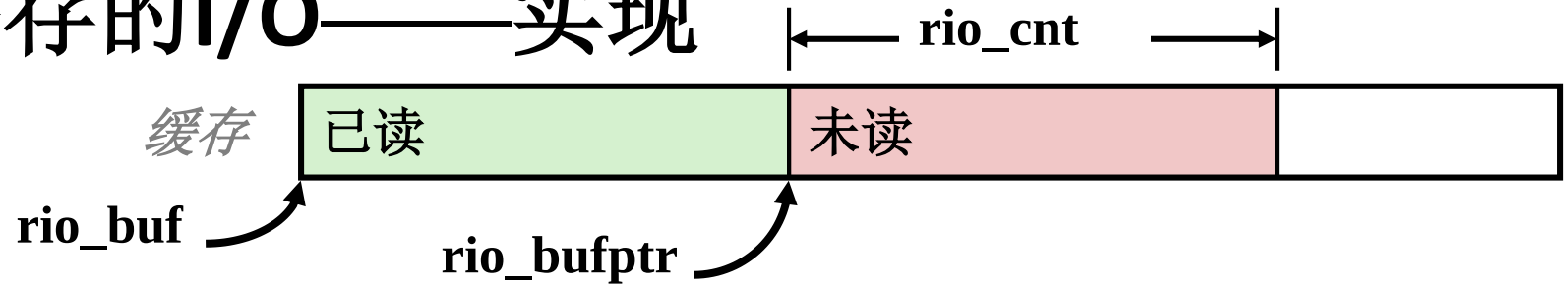
- 读文件
- 文件有关联的缓冲区，用来保存已经从文件中读出，但是还未被用户代码读取的字节



- 相当于在Unix 文件之上加了一层:



带缓存的I/O——实现



- 结构体

```
#define RIO_BUFSIZE 8192
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 ;
```

- 初始化

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

Robust I/O

- 带缓存的RIO函数
  - 高效地从文件中读取文本行和二进制数据，该文件的内容缓存在应用程序级缓存(缓冲区)中

```
#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 maxlen);
        returns: number of bytes read (0 if EOF), -1 on error
```

Robust I/O

- **rio_readnb 从文件 fd中最多读n个字节**
  - 停止条件
    - 已读**maxlen**字节 **(n字节)**
    - 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
- **rio_readlineb 从文件fd中读取最多maxlen字节的文本行，并存储在 usrbuf中**
    - 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**maxlen**字节
    - 遇到EOF
    - 遇到新行符 ('**\n**')

Robust I/O函数

- 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
```

```
1  static ssize_t rio_read(rio_t *rp, char *usrbuf, size_t n)
2  {//从内部缓冲区min(n, rio_cnt) 字节数据到 usrbuf中
3    int cnt = 0;
4
5    while (rp->rio_cnt <= 0) { /* refill if buf is empty */
6          rp->rio_cnt = read(rp->rio_fd, rp->rio_buf,
7                                        sizeof(rp->rio_buf));
8          if (rp->rio_cnt < 0) {
9              if (errno != EINTR)
10                 return –1 ;
11         }
12         else if (rp->rio_cnt  ==  0) /* EOF */
13             return 0;
14         else
15             rp->rio_bufptr = rp->rio_buf;  /* reset buffer ptr */
16   }/*…*/
```

```
17
18      /* Copy min(n, rp->rio_cnt) bytes
           from internal buf to user buf */
19      cnt = n ;
20    if ( rp->rio_cnt < n)
21       cnt = rp->rio_cnt ;
22    memcpy(usrbuf, rp->rio_bufptr, cnt) ;
23    rp->rio_buffer += cnt ;
24    rp->rio_cnt -= cnt ;
25    return cnt ;
26 }
```

```
1  ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n)
2  {// Robustly read n bytes (buffered)
3      size_t nleft = n;   ssize_t nread ;
4      char *bufp = usrbuf;
5      while (nleft > 0) {
6          if ((nread = rio_read(rp, bufp, nleft)) < 0) {
7              if ( errno = EINTR)
8                  /* interrupted by sig handler return */
9                  nread = 0;
10             else
11                 return –1;
12         }
13         else if (nread == 0)
14             break;
15         nleft -= nread;
16         bufp += nread;
17     }
18     return (n – nleft);
19 }
```

```
1  ssize_t rio_readlineb (rio_t *rp, void *usrbuf, size_t maxlen)
2  {// Robustly read a text line (buffered)
3      int n, rc;
4      char c, *bufp = usrbuf;
5      for (n=1; n < maxlen; n++) {
6          if ((rc = rio_read(rp, &c, 1)) == 1) {
7              *bufp++ = c;
8              if (c == '\n')
9                  break;
10         } else if (rc == 0) {
11             if (n== 1)
12                 return 0; /* EOF, no data read */
13             else
14                 break;
15         } else
16             return –1; /* error */
17     }
18     *bufp = 0 ;
19     return n ;
20 }
```

主要内容

- Unix I/O
- 元数据、共享和重定位
- RIO,健壮的I/O程序包(robust I/O)
- 标准I/O
- 结束语

C语言的标准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

- 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** 和 **fprintf**)

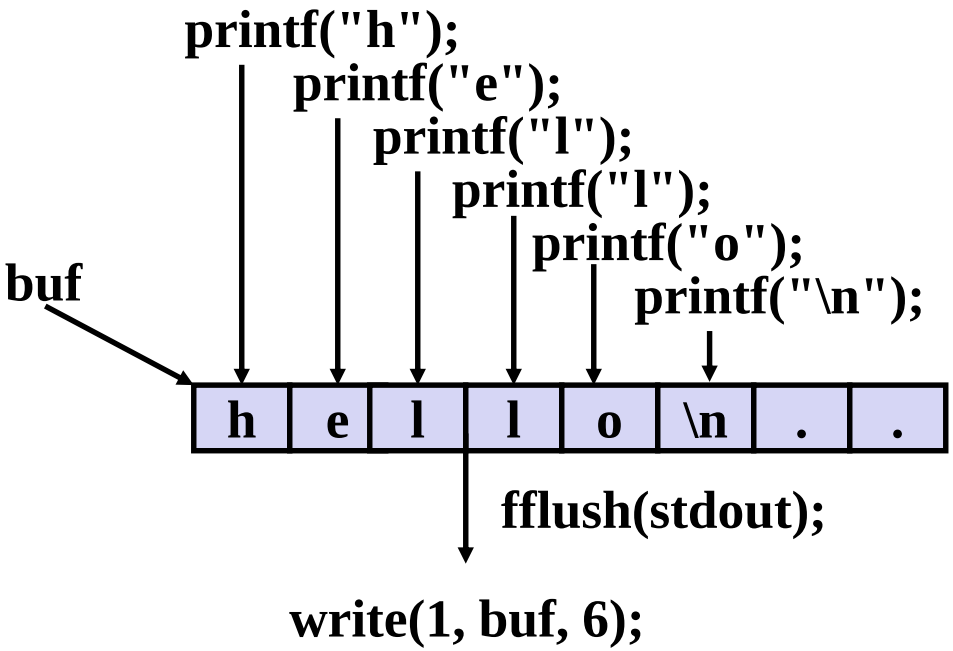
C语言的标准I/O流

- 标准 I/O库将一个打开的文件 模型化为**流**
  - 流类型FILE是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");
}
```

C语言的标准I/O：使用缓存

- 标准 I/O 函数使用带缓冲的I/O



- 缓冲区刷新到输出 fd:
  - 遇到'\n'、调用fflush、 exit或从main返回

C语言的标准I/O缓存的运作

- 使用一直很迷人的 Linux程序 `strace`， 可以看到缓冲的运作:

```
#include <stdio.h>

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

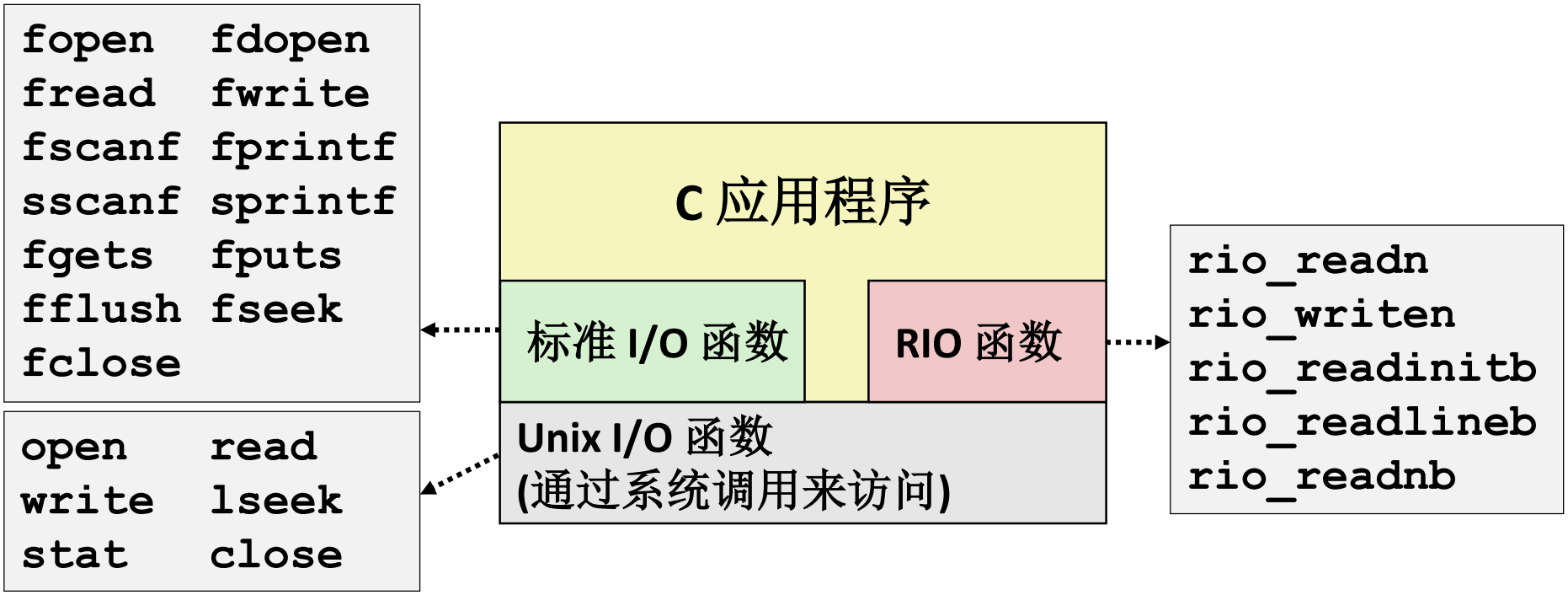
```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6)              = 6
...
exit_group(0)                       = ?
```

主要内容

- Unix I/O
- 元数据、共享和重定位
- 标准I/O
- RIO,健壮的I/O程序包(robust 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 是基于底层(low-level) Unix I/O 函数来实现的。



- 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
  - 发生不足值时容易出错
  - 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，标准I/O和RIO包均已解决

标准I/O的优点和缺点

- 优点:
  - 通过使用**buf**，减少**read**和**write**系统调用的次数，提高效率
  - 能自动处理不足值
- 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

- 一般规则：使用最高级别的I/O函数
  - 大多数 C 程序员只用标准 I/O函数就能完成所有工作
  - 但是，一定要明白我们所用的这些标准I/O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
  - 在需要绝对最高性能的极少数情况下
- 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

附录: 处理二进制文件Binary File

- 二进制文件
  - 任意字节值的序列!
  - 当然包括字节值0x00
- 二进制文件不可以使用文本、字符串类的函数
  - **面向文本的I/O函数: fgets, scanf, rio_readlineb**
    - 会把一些字节值(0x0d、0x0a)解释成 EOL
  - **字符串函数： strlen, strcpy, strcat**
    - 把字节值0x00看成字符串结束标识;
- **可以使用的函数**
  - **read**
  - **rio_readn (int fd, void *usrbuf, size_t n) : 封装的read，无buffer**
  - **rio_readnb (rio_t *rp, void *usrbuf, size_t n) : 封装的rio_read(封装的read)，有buffer**

访问文件夹

- 对目录/文件夹操作：读取其包含的条目(entries)
  - 结构体`dirent`包含文件夹条目的信息
  - 在遍历目录时，结构体DIR包含目录信息
    https://blog.csdn.net/zhuyi2654715/article/details/7605051#

```
#include <sys/types.h>
#include <dirent.h>
{
  DIR *directory;
  struct dirent *de;
  ...
  if (!(directory = opendir(dir_name)))
      error("Failed to open directory");
  ...
  while (0 != (de = readdir(directory))) {
      printf("Found file: %s\n", de->d_name);
  }
  ...
  closedir(directory);
}
```

For Further Information

- **The Unix bible:**
  - W. Richard Stevens & Stephen A. Rago, ***Advanced Programming in the Unix Environment***, 2nd Edition, Addison Wesley, 2005
    - Updated from Stevens's 1993 classic text

- **The Linux bible:**
  - Michael Kerrisk, The Linux Programming Interface, No Starch Press, 2010
    - Encyclopedic and authoritative