第10章 系统级I/O
System-Level I/O

100076202： 计算机系统导论

任课教师：
宿红毅　张艳　黎有琦　颜珂

原作者：
Randal E. Bryant and David R. O'Hallaron

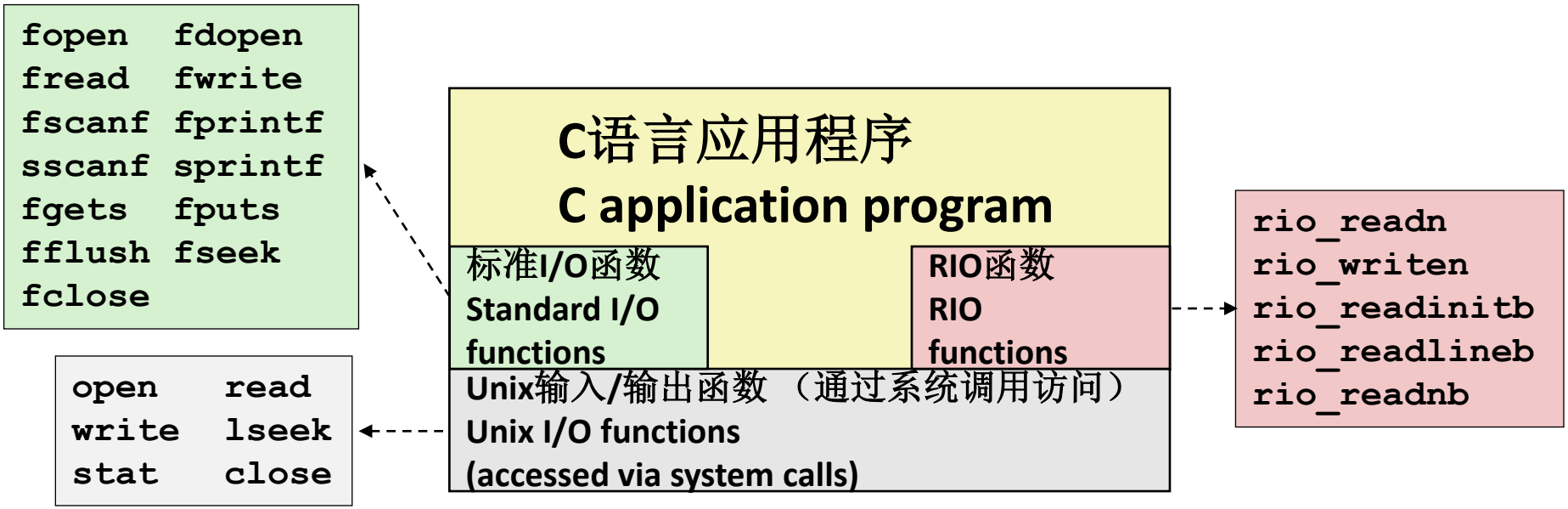
Carnegie Mellon University

议题

- **Unix输入和输出 Unix I/O**
- 元数据、共享和重定向 Metadata, sharing, and redirection
- 标准输入和输出 Standard I/O
- RIO（健壮输入/输出）包 RIO (robust I/O) package
- 闭幕词 Closing remarks

Unix输入/输出和C语言标准输入/输出
Unix I/O and C Standard I/O

- 两组接口：系统级和C语言级 **Two sets: system-level and C-level**
- 健壮输入/输出（RIO）：213课程特殊的包装函数 **Robust I/O (RIO): 213 special wrappers**
  良好的编码实践：处理错误检查、信号和“不足值” **good coding practice:** handles error checking, signals, and “short counts”



Unix输入/输出概述 Unix I/O Overview

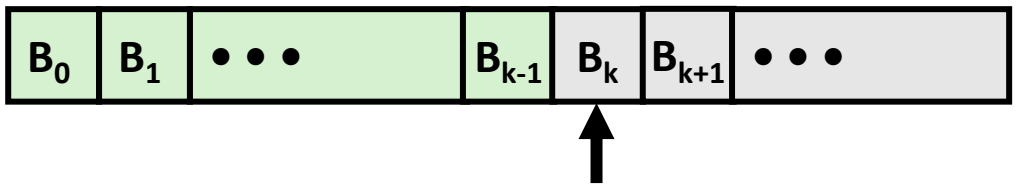
- Linux文件是m个字节的序列 **A Linux *file* is a sequence of *m* bytes:**
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$

- 很酷的事实：所有I/O设备都可以表示为文件 **Cool fact: All I/O devices are represented as files:**
  - **`/dev/sda2`** (用户磁盘分区 **`/usr`** disk partition)
  - **`/dev/tty2`** (终端 terminal)

- 甚至内核也表示为文件：**Even the kernel is represented as a file:**
  - **`/boot/vmlinuz-3.13.0-55-generic`** (内核映像 kernel image)
  - **`/proc`** (内核数据结构 kernel data structures)

Unix输入/输出概述 Unix I/O Overview

- 文件到设备的优雅映射允许内核导出简单接口，称为Unix I/O： **Elegant mapping of files to devices allows kernel to export simple interface called *Unix I/O:***
  - 打开和关闭文件 Opening and closing files
    - **`open()`** and **`close()`**
  - 读和写文件 Reading and writing a file
    - **`read()`** and **`write()`**
  - 改变当前文件位置 Changing the ***current file position*** (seek)
    - 指明下次读或写操作的文件偏移位置 indicates next offset into file to read or write
    - **`lseek()`**



当前文件位置为k

Current file position = k

文件类型 File Types

- 每个文件都有一种类型，指明其在系统中的角色 **Each file has a *type* indicating its role in the system**
  - *普通文件：*包含任意的数据 *Regular file:* Contains arbitrary data
  - *目录：*相关文件组的索引| *Directory:* Index for a related group of files
  - *套接字：*用于和另一台机器上的进程通信 *Socket:* For communicating with a process on another machine

- 其它超出我们讨论范围的文件类型 **Other file types beyond our scope**
  - *命名管道（先进先出） 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

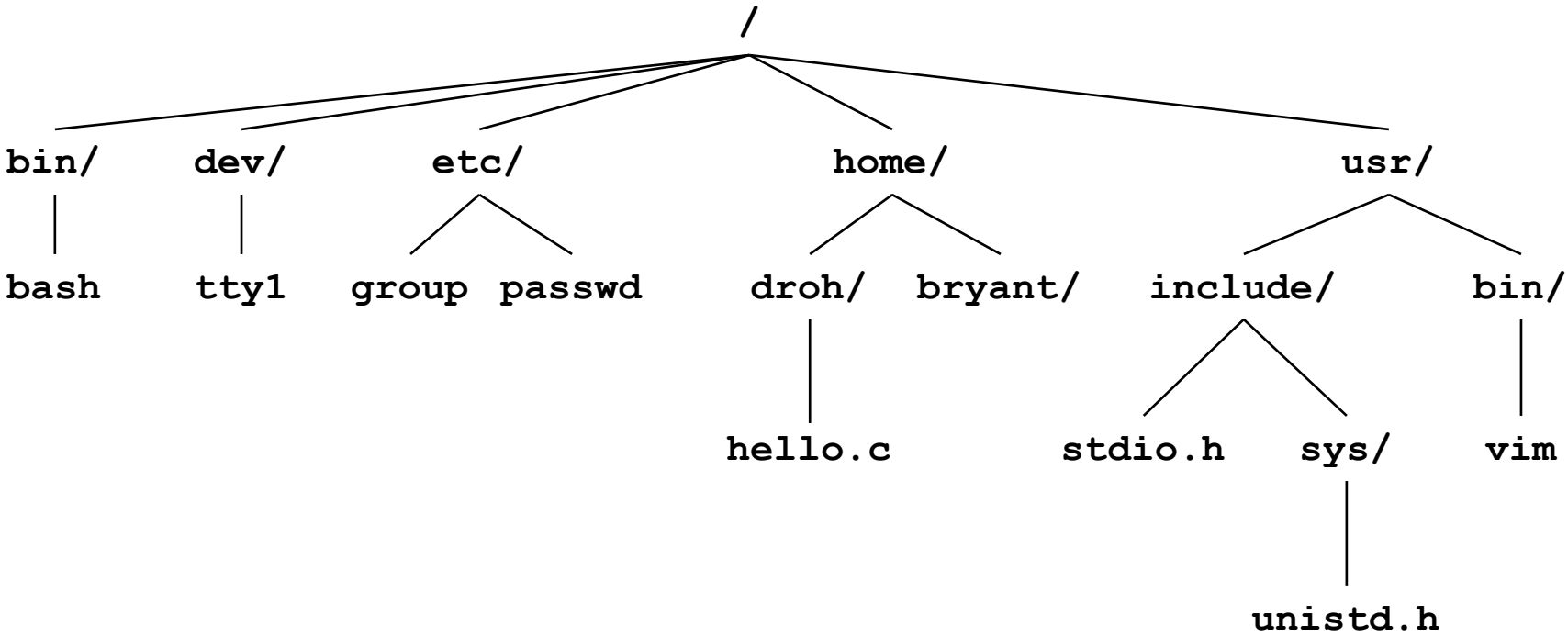
- 普通文件包含任意数据 **A regular file contains arbitrary data**
- 应用程序通常进一步分成*文本文件*和*二进制文件* **Applications often distinguish between *text files* and *binary files***
  - 文本文件是仅包含ASCII或统一编码字符的普通文件 Text files are regular files with only ASCII or Unicode characters
  - 二进制文件可以包含一切内容 Binary files are everything else
    - 例如，目标文件、JPEG图片 e.g., object files, JPEG images
  - 内核并不知道这种不同 Kernel doesn't know the difference!
- 文本文件是文本行序列 **Text file is sequence of *text lines***
  - 文本行是以换行字符（'\n'）结束的字符序列 Text line is sequence of chars terminated by *newline char* (**'\n'**)
    - 换行符为0xa，与ASCII换行字符（LF）相同 Newline is **0xa**, same as ASCII line feed character (LF)
- 其它系统中的行结束（EOL）指示符 **End of line (EOL) indicators in other systems**
  - Linux and Mac OS: **'\n'** (**0xa**)
    - 换行符 line feed (LF)
  - Windows and Internet protocols: **'\r\n'** (**0xd 0xa**)
    - 回车后跟换行 Carriage return (CR) followed by line

目录 Directories

- 目录由链接数组构成 **Directory consists of an array of *links***
  - 每个链接映射一个*文件名*到一个文件 Each link maps a *filename* to a file
- 每个目录包含至少两个条目 **Each directory contains at least two entries**
  - 点链接到自身 `.` (dot) is a link to itself
  - 点点在*目录层次*中链接到父目录（见下页幻灯片） `..` (dot dot) is a link to *the parent directory* in the *directory hierarchy* (next slide)
- 操纵目录的命令 **Commands for manipulating directories**
  - **`mkdir`**: 创建空目录 create empty directory
  - **`ls`**: 查看目录内容 view directory contents
  - **`rmdir`**: 删除空目录 delete empty directory

目录层次结构 Directory Hierarchy

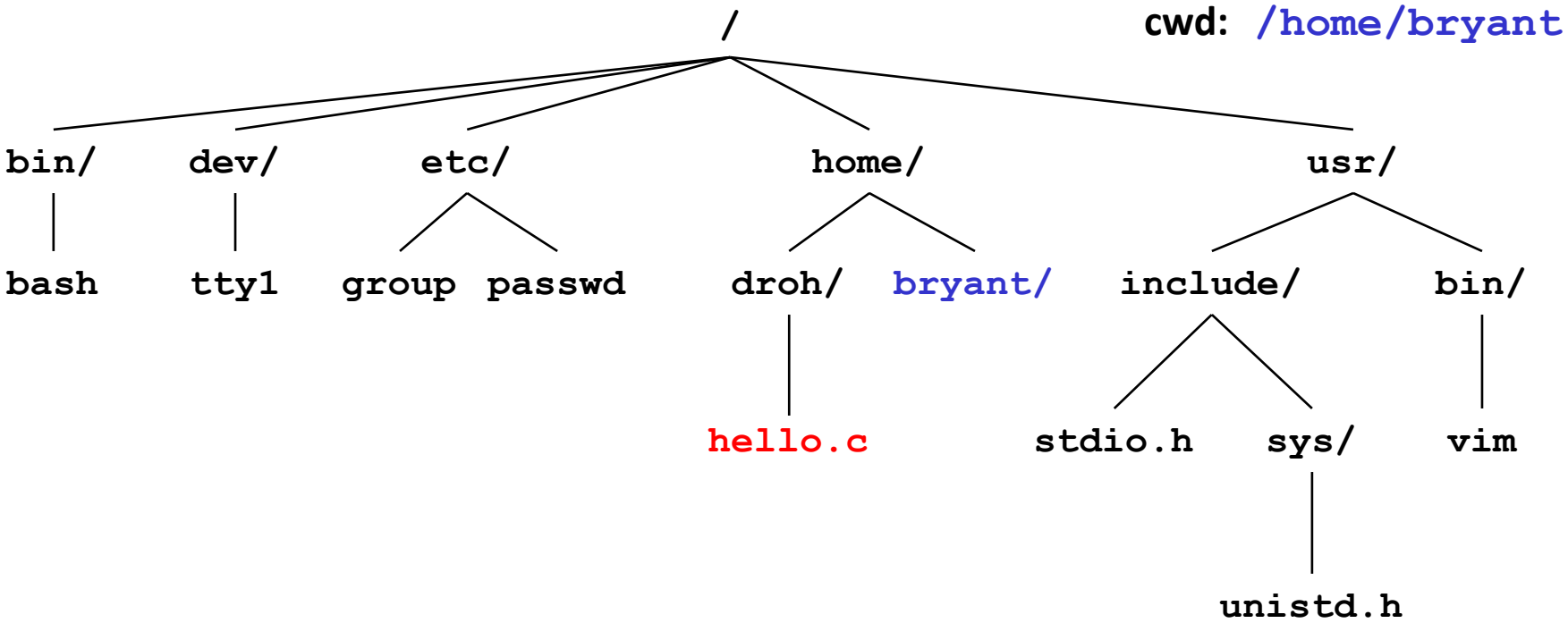
- 所有文件组织成层次结构，由名为/（斜杠）的根目录锚定 **All files are organized as a hierarchy anchored by root directory named / (slash)**



- 内核为每个进程维护*当前工作目录（**cwd**）* **Kernel maintains *current working directory (cwd)* for each process**
  - 使用cd命令改变当前目录 Modified using the **cd** command

路径名 Pathnames

- 在层级结构的文件位置由*路径名*指示 **Locations of files in the hierarchy denoted by *pathnames***
  - *绝对路径名*以斜杠开始并指示路径从根开始 *Absolute pathname* starts with '/' and denotes path from root
    - **`/home/droh/hello.c`**
  - *相对路径名*指示路径从当前工作目录开始 *Relative pathname* denotes path from current working directory
    - **`../droh/hello.c`**



打开文件 Opening Files

- 打开文件通知内核你正准备访问该文件 **Opening a file informs the kernel that you are getting ready to access that file**

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

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

- 返回一个小的整数标识*文件描述符* **Returns a small identifying integer *file descriptor***
  - `fd == -1` 指示发生错误 indicates that an error occurred
- **Linux shell创建的每个进程存活期都有三个打开文件，这些文件和终端设备相关联：Each process created by a Linux shell begins life with three open files associated with a terminal:**
  - 0: 标准输入 standard input (stdin)
  - 1: 标准输出 standard output (stdout)
  - 2: 标准错误 standard error (stderr)

关闭文件 Closing Files

- 关闭文件通知内核你完成了对该文件的访问 **Closing a file informs the kernel that you are finished accessing that file**

```
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在线程化的程序中导致灾难（稍后将详细介绍） **Closing an already closed file is a recipe for disaster in threaded programs (more on this later)**
- 寓意：始终检查返回代码，即使是对于close等看似良性的函数 **Moral: Always check return codes, even for seemingly benign functions such as `close()`**

读文件 Reading Files

- 读文件从当前文件位置复制若干字节到内存，然后更新文件位置 **Reading a file copies bytes from the current file position to memory, and then updates file position**

```
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**读到**buf**的字节数 **Returns number of bytes read from file `fd` into `buf`**
  - 返回类型ssize_t是有符号整数 Return type **`ssize_t`** is signed integer
  - **`nbytes`**小于零指示发生错误 **`nbytes < 0`** indicates that an error occurred
  - ***不足值***（nbytes小于buf大小）是有可能的，而且不是错误 ***Short counts*** (**`nbytes < sizeof(buf)`**) are possible and are not errors!

写文件 Writing Files

- 写文件从内存复制若干字节到当前文件位置，然后更新当前文件位置 **Writing a file copies bytes from memory to the current file position, and then updates current file position**

```
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写到文件fd的字节数 **Returns number of bytes written from `buf` to file `fd`**
  - **`nbytes`**小于零指示发生错误 **`nbytes < 0`** indicates that an error occurred
  - 和读文件一样，不足值有可能，而且不是错误 As with reads, short counts are possible and are not errors!

简单Unix输入/输出示例
Simple Unix I/O example

- 拷贝文件到标准输出，一次一个字节 Copying file to stdout, one byte at a time

```
#include "csapp.h"

int main(int argc, char *argv[])
{
    char c;
    int infd = STDIN_FILENO;
    if (argc == 2) {
        infd = open(argv[1], O_RDONLY, 0);
    }
    while(read(infd, &c, 1) != 0)
        write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

showfile1_nobuf.c

- 演示： Demo:

```
linux> strace ./showfile1_nobuf names.txt
```

关于不足值 On Short Counts

- 在这些情况下可能发生不足值： **Short counts can occur in these situations:**
  - 在读文件时遇到（文件结束）EOF　Encountering (end-of-file) EOF on reads
  - 从终端读文本行 Reading text lines from a terminal
  - 读和写网络套接字 Reading and writing network sockets

- 在这些情况下从不发生不足值 **Short counts never occur in these situations:**
  - 从磁盘文件读（除了EOF）Reading from disk files (except for EOF)
  - 写入磁盘文件 Writing to disk files

- 最佳做法是始终允许不足值 **Best practice is to always allow for short counts**

成熟的缓冲I/O代码
Home-Grown Buffered I/O Code

- 拷贝文件到标准输出，每次BUFSIZE字节 Copying file to stdout, BUFSIZE bytes at a time

```
#include "csapp.h"
#define BUFSIZE 64


int main(int argc, char *argv[])
{
    char buf[BUFSIZE];
    int infd = STDIN_FILENO;
    if (argc == 2) {
        infd = open(argv[1], O_RDONLY, 0);
    }
    while((nread = read(infd, buf, BUFSIZE)) != 0)
        write(STDOUT_FILENO, buf, nread);
    exit(0);
}
```

showfile2_buf.c

- 演示： Demo:

```
linux> strace ./showfile2_buf names.txt
```

议题

- Unix输入/输出 Unix I/O
- **元数据、共享和重定向 Metadata, sharing, and redirection**
- 标准输入/输出 Standard I/O
- RIO（健壮I/O）包 RIO (robust I/O) package
- 闭幕词 Closing remarks

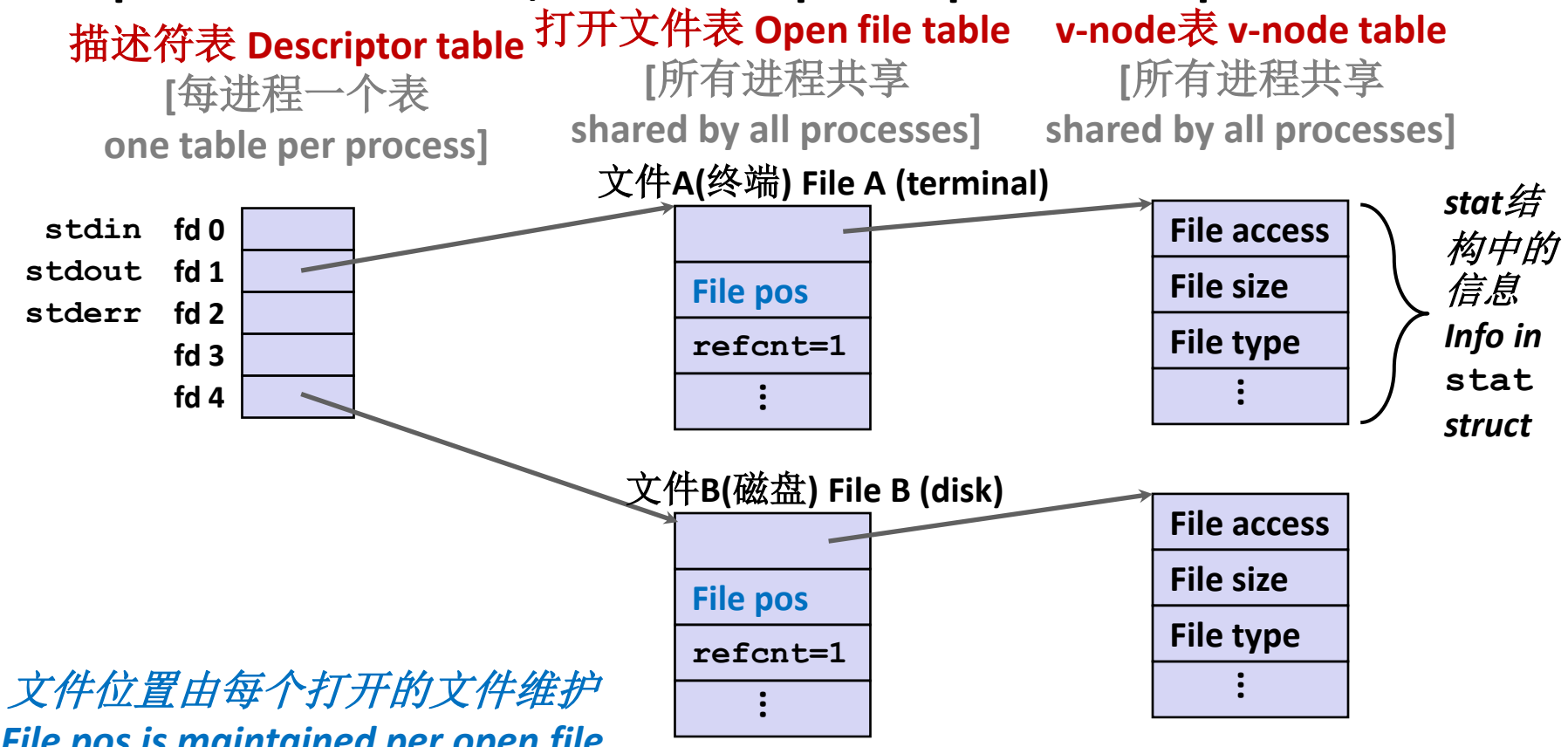
文件元数据 File Metadata

- *元数据* 是关于数据的数据，在本例中是文件数据 ***Metadata*** **is data about data, in this case file data**
- 每个文件的元数据由内核维护 **Per-file metadata maintained by kernel**
  - 用户用stat和fstat函数访问 Accessed by users with the **`stat`** and **`fstat`** functions

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

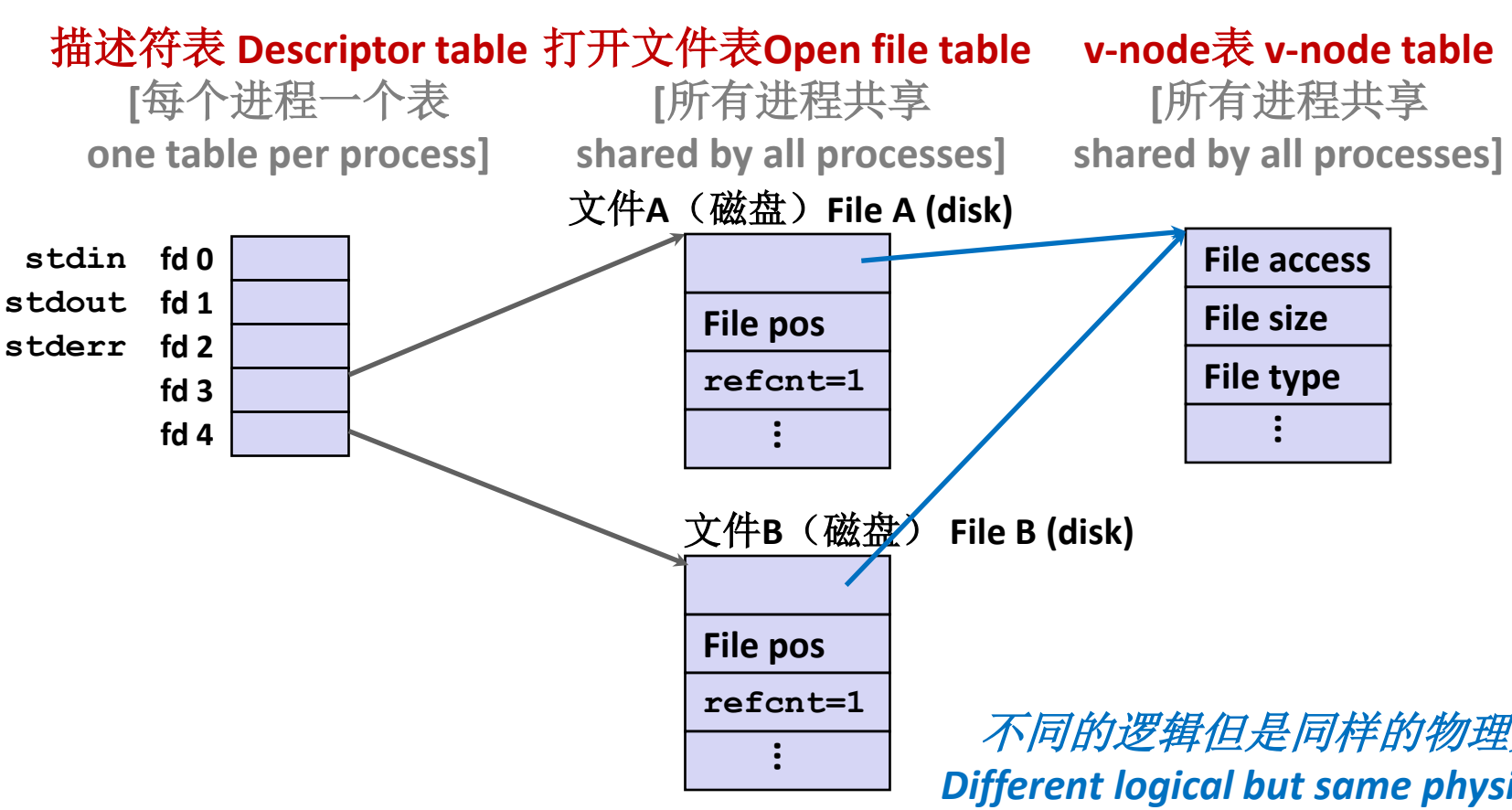
Unix内核如何表示打开文件
How the Unix Kernel Represents Open Files

- 两个描述符引用两个不同的打开文件。描述符1（标准输出）指向终端，描述符4指向打开的磁盘文件 Two descriptors referencing two distinct open files. Descriptor 1 (`stdout`) points to terminal, and descriptor 4 points to open disk file



文件共享 File Sharing

- 两个不同的描述符通过两个不同的打开文件表共享磁盘文件 **Two distinct descriptors sharing the same disk file through two distinct open file table entries**
  - 例如调用**open**两次，两次的**文件名**参数相同 e.g., calling **`open`** twice with the same **`filename`** argument

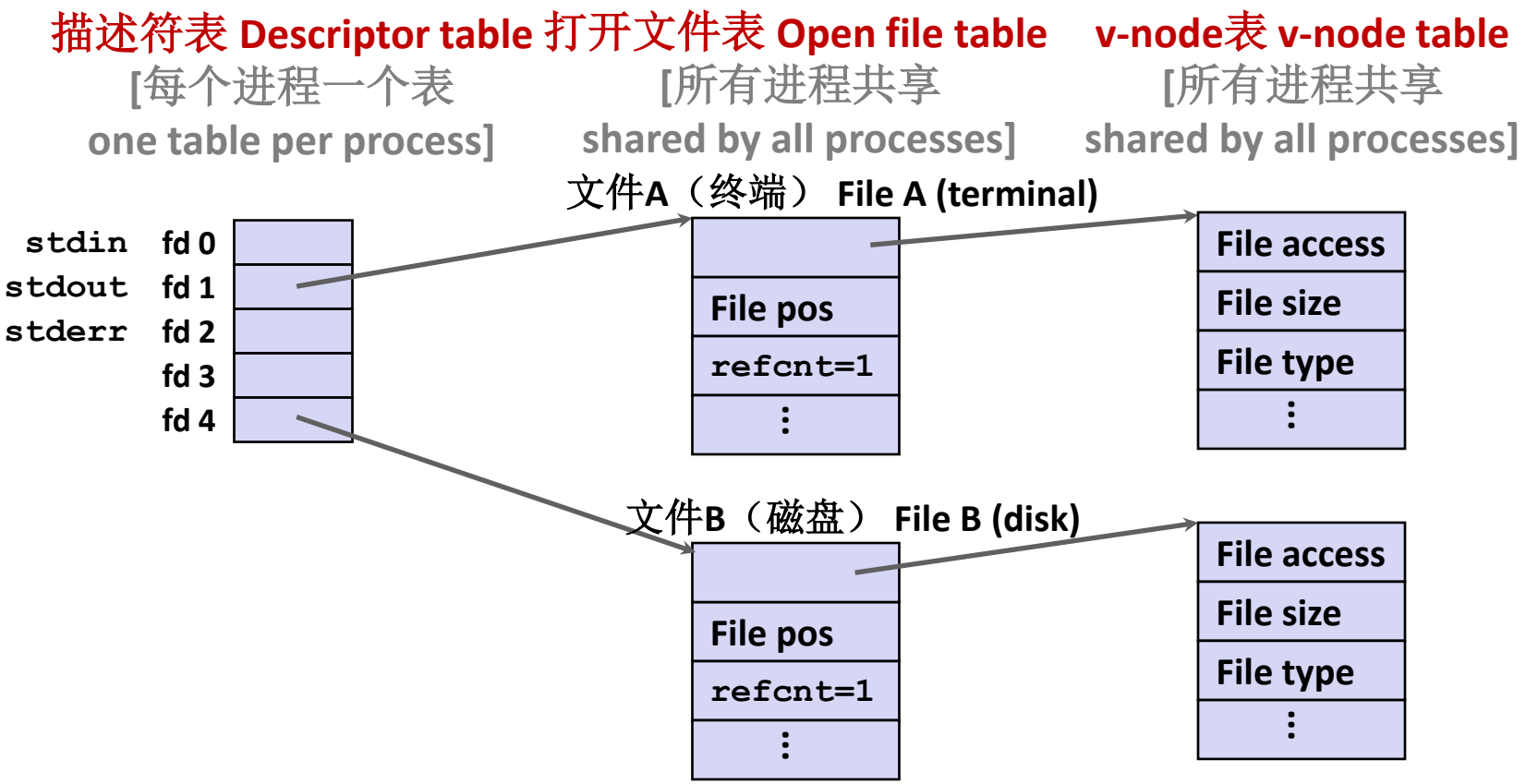


不同的逻辑但是同样的物理文件

Different logical but same physical file

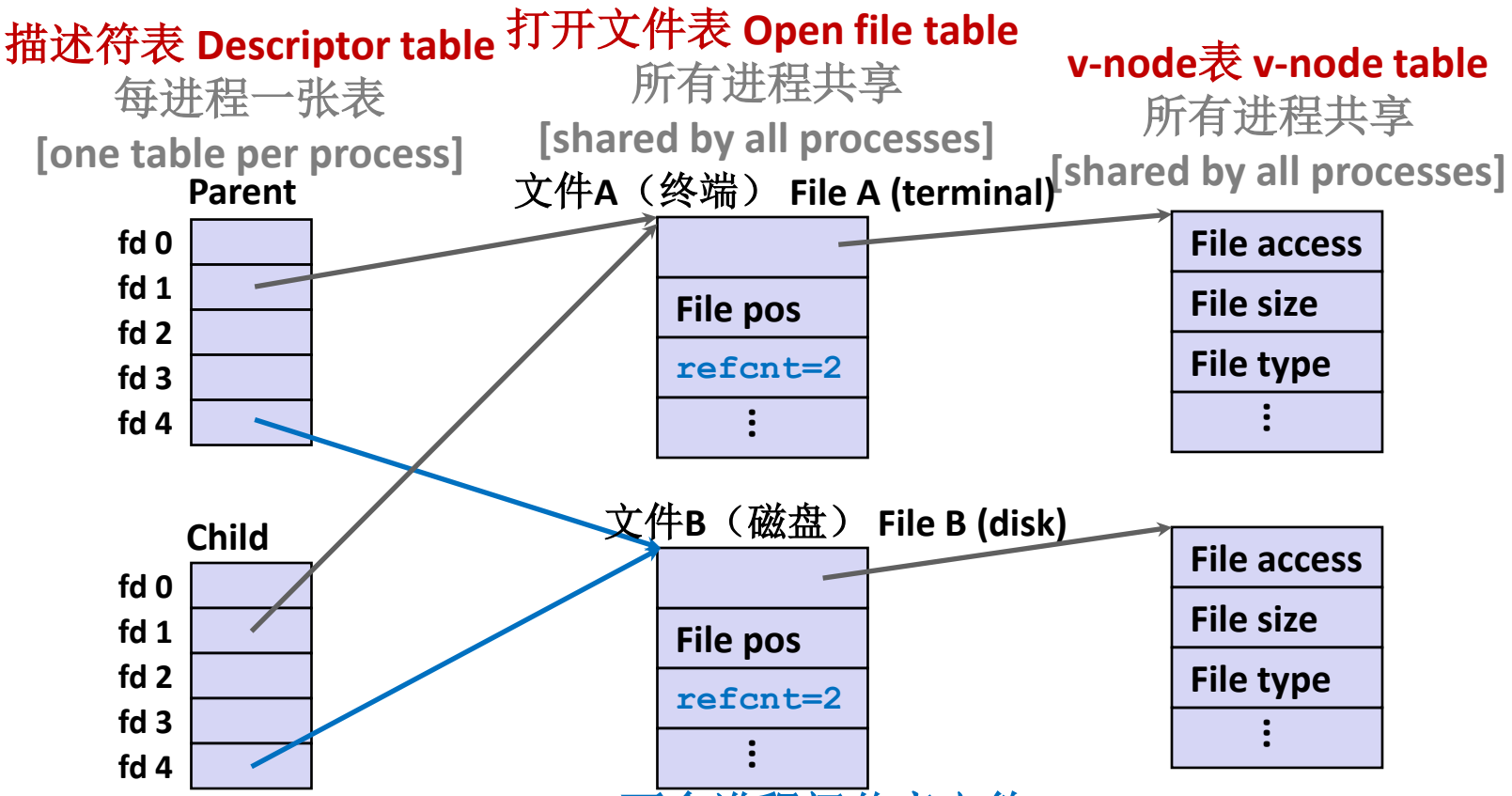
进程如何共享文件：fork
How Processes Share Files: `fork`

- 子进程继承其父进程的打开文件 **A child process inherits its parent's open files**
  - 注意：exec函数没有改变局面（使用fcntl函数改变） Note: situation unchanged by **`exec`** functions (use **`fcntl`** to change)
- 调用fork*之前 Before* **`fork`** call:



进程如何共享文件：fork
How Processes Share Files: `fork`

- 子进程继承其父进程的打开文件 A child process inherits its parent's open files
- 调用fork*之后 After* `fork`:
  - 子进程与父进程的表一样，而且每个refcnt加一 Child's table same as parent's, and +1 to each `refcnt`



两个进程间共享文件 File is shared between processes

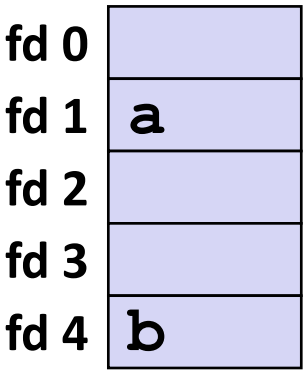
输入/输出重定向 I/O Redirection

- 问题：外壳如何实现I/O重定向？ **Question: How does a shell implement I/O redirection?**

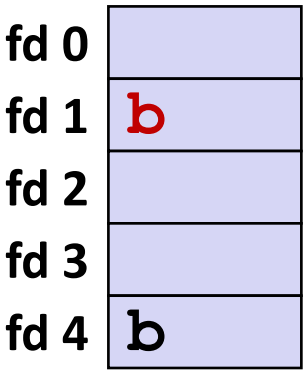
```
linux> ls > foo.txt
```

- 答案：通过调用**dup2(oldfd,newfd)**函数 **Answer: By calling the `dup2(oldfd, newfd)` function**
  - 复制（每个进程）描述符表条目oldfd到条目newfd Copies (per-process) descriptor table entry **`oldfd`** to entry **`newfd`**

描述符表 **Descriptor table**
调用之前 ***before*** **`dup2(4,1)`**



描述符表 **Descriptor table**
调用之后 ***after*** **`dup2(4,1)`**

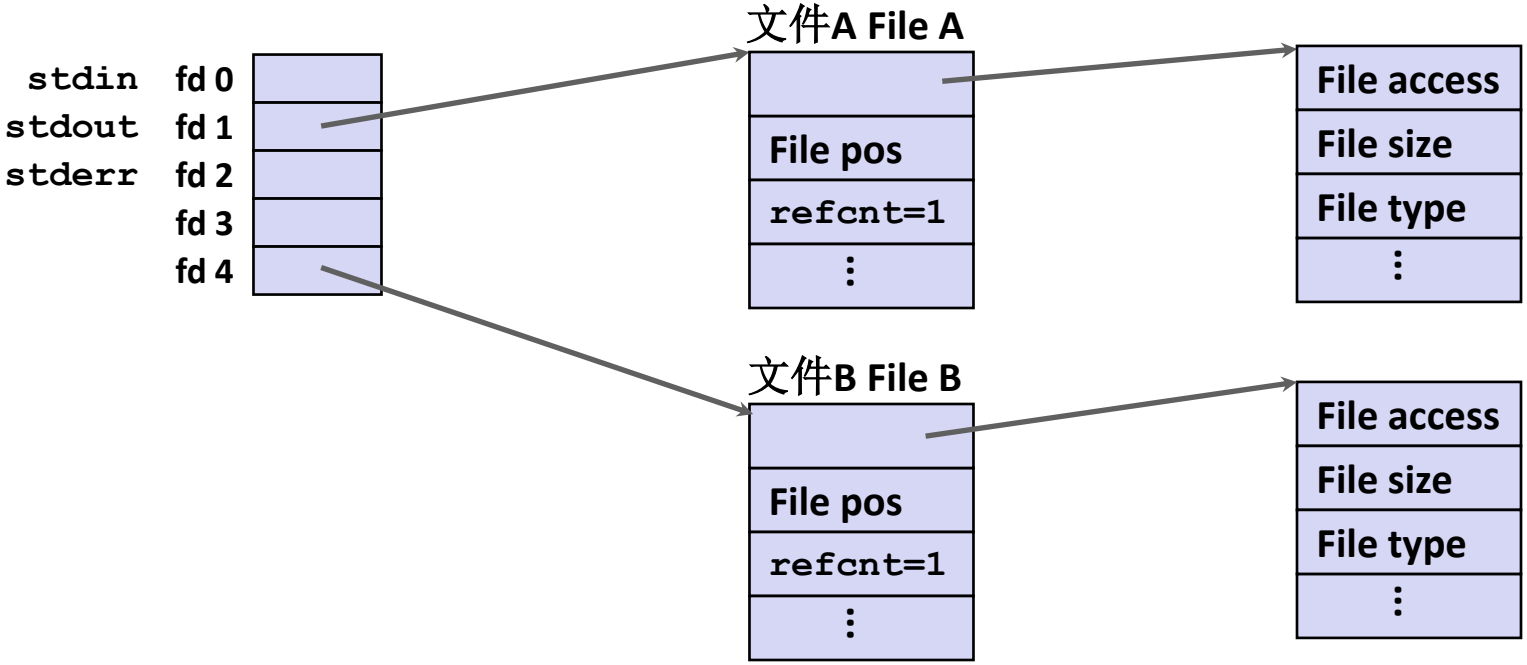


输入/输出重定向示例
I/O Redirection Example

- 步骤#1：打开文件，标准输出应该重定向到此文件 **Step #1: open file to which stdout should be redirected**
  - 在子进程执行外壳代码时发生，在exec调用之前 Happens in child executing shell code, before **`exec`**

描述符表 **Descriptor table** 打开文件表 **Open file table** v-node表 **v-node table**

每进程一张表 [one table per process] | 所有进程共享 [shared by all processes] | 所有进程共享 [shared by all processes]

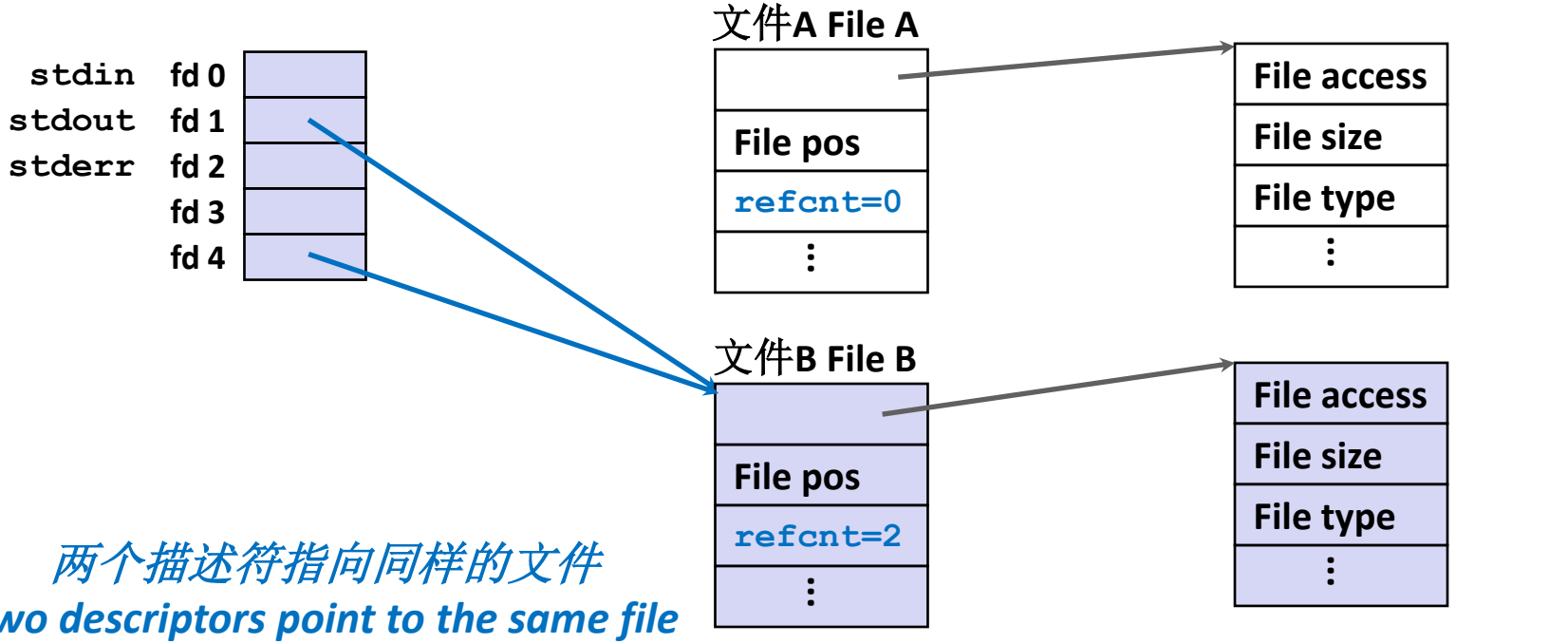


输入/输出重定向示例
I/O Redirection Example (cont.)

- 步骤#2：调用dup2(4,1) Step #2: call dup2(4,1)
  - 导致fd=1（标准输出）指向fd=4所指向的磁盘文件 Cause fd=1 (`stdout`) to refer to disk file pointed at by fd=4

描述符表 Descriptor table 打开文件表 Open file table v-node表 v-node table

每进程一张表 [one table per process] | 所有进程共享 [shared by all processes] | 所有进程共享 [shared by all processes]



两个描述符指向同样的文件

Two descriptors point to the same file

前奏：I/O和重定向示例
Warm-Up: I/O and Redirection Example

```
#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

- 文件包含“abcde”，该程序会输出什么？ What would this program print for file containing “abcde”?

前奏：I/O和重定向示例
Warm-Up: I/O and Redirection Example

```
#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

```
c1 = a, c2 = a, c3 = b
```

```
dup2(oldfd, newfd)
```

- 文件包含“abcde”，该程序会输出什么？ What would this program print for file containing “abcde”?

大师班：进程控制和I/O

Master Class: Process Control and I/O

```c
#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

- 文件包含“abcde”，该程序会输出什么？ What would this program print for file containing “abcde”?

大师班：进程控制和I/O

Master Class: Process Control and I/O

```c
#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
```

奖金：它往哪边走？

Bonus: Which way does it go?

- 文件包含“abcde”，该程序会输出什么？ **What would this program print for file containing “abcde”?**

议题

- Unix输入/输出 Unix I/O
- 元数据、共享和重定向 Metadata, sharing, and redirection
- **标准输入/输出 Standard I/O**
- RIO（健壮I/O）包 RIO (robust I/O) package
- 闭幕词 Closing remarks

标准输入/输出函数 Standard I/O Functions

- **C语言标准库（libc.so）包含高级*标准I/O*函数的集合**
  The C standard library (`libc.so`) contains a collection of higher-level *standard I/O* functions
  - 见附录B K&R文档 Documented in Appendix B of K&R

- **标准I/O函数的例子： Examples of standard I/O functions:**
  - 打开和关闭文件 Opening and closing files (**`fopen`** and **`fclose`**)
  - 读和写若干字节 Reading and writing bytes (**`fread`** and **`fwrite`**)
  - 读和写文本行 Reading and writing text lines (**`fgets`** and **`fputs`**)
  - 格式化读和写 Formatted reading and writing (**`fscanf`** and **`fprintf`**)

标准输入/输出流 Standard I/O Streams

- 标准I/O模型打开文件作为*流* **Standard I/O models open files as *streams***
  - 文件描述符和内存中缓冲区的抽象 Abstraction for a file descriptor and a buffer in memory

- **C语言程序开始运行时会有三个打开的流（在stdio.h中定义）**
  C programs begin life with three open streams (defined in `stdio.h`)
  - **`stdin`** (标准输入 standard input)
  - **`stdout`** (标准输出 standard output)
  - **`stderr`** (标准错误 standard error)

```
#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");
}
```

带缓冲的输入/输出：动机
Buffered I/O: Motivation

- 应用程序通常一次读/写一个字符 **Applications often read/write one character at a time**
  - `getc, putc, ungetc`
  - `gets, fgets`
    - 读文本行，一次读一个字符，直到读到换行符为止 Read line of text one character at a time, stopping at newline
- 实现为Unix I/O调用比较费时 **Implementing as Unix I/O calls expensive**
  - **读**和**写**需要Unix内核调用 **`read`** and **`write`** require Unix kernel calls
    - 大于1万个时钟周期 > 10,000 clock cycles

缓冲区
Buffer

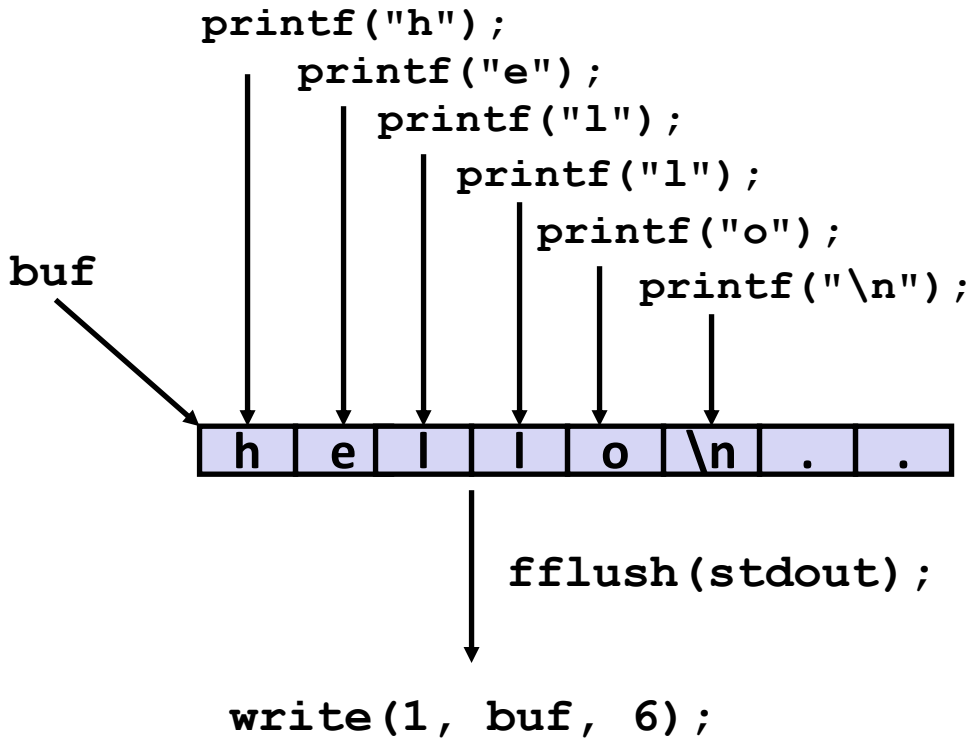
带缓冲的输入/输出：动机
Buffered I/O: Motivation

- 解决方案：带缓冲的读 **Solution: Buffered read**
  - 使用Unix **read**获取字节块 Use Unix **`read`** to grab block of bytes
  - 用户输入函数每次从缓冲区获取一个字节 User input functions take one byte at a time from buffer
    - 当缓冲为空时重新填充缓冲区 Refill buffer when empty

缓冲区
Buffer

在标准输入/输出中的缓冲
Buffering in Standard I/O

- 标准I/O函数使用带缓冲的I/O　Standard I/O functions use buffered I/O



- 在遇到“\n”时缓冲区刷新到输出文件描述符fd，调用fflush或exit，或从main返回　Buffer flushed to output fd on “\n”, call to `fflush` or `exit`, or return from `main`

标准输入/输出缓冲的作用
Standard I/O Buffering in Action

- 使用总是令人着迷的Linux strace程序，可以亲自看到这种缓冲的作用： You can see this buffering in action for yourself, using the always fascinating Linux `strace` program:

```
#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)                           = ?
```

标准输入/输出示例 Standard I/O Example

- 用标准输入/输出，一行一行地复制文件到标准输出 Copying file to `stdout`, line-by-line with stdio

```c
#include "csapp.h"
#define MLINE 1024

int main(int argc, char *argv[])
{
    char buf[MLINE];
    FILE *infile = stdin;
    if (argc == 2) {
        infile = fopen(argv[1], "r");
        if (!infile) exit(1);
    }
    while(fgets(buf, MLINE, infile) != NULL)
        fputs(buf, stdout);
    exit(0);
}
```

showfile3_stdio.c

- 演示 Demo:

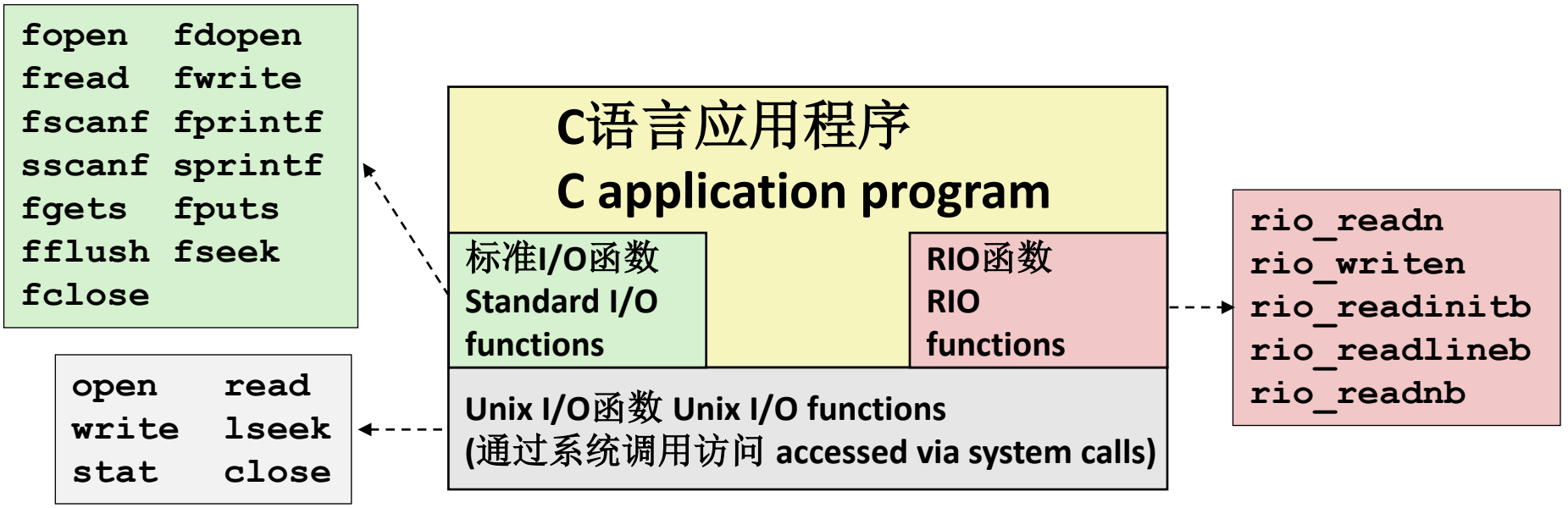
```
linux> strace ./showfile3_stdio names.txt
```

议题

- Unix输入/输出 Unix I/O
- 元数据、共享和重定向 Metadata, sharing, and redirection
- 标准输入/输出 Standard I/O
- **RIO（健壮I/O）包 RIO (robust I/O) package**
- 闭幕词 Closing remarks

Unix I/O、C语言标准I/O和RIO
Unix I/O, C Standard I/O, and RIO

- Unix I/O构建有两个不兼容的库 Two *incompatible* libraries building on Unix I/O
- 健壮I/O（RIO）：213课程特殊的包装器函数 Robust I/O (RIO): 213 special wrappers
  良好的编码实践：处理错误检查、信号和“不足值” **good coding practice:** handles error checking, signals, and “short counts”



Unix输入/输出简要回顾 Unix I/O Recap

```
/* Read at most max_count bytes from file into buffer.
   Return number bytes read, or error value */
ssize_t read(int fd, void *buffer, size_t max_count);
```

```
/* Write at most max_count bytes from buffer to file.
   Return number bytes written, or error value */
ssize_t write(int fd, void *buffer, size_t max_count);
```

- 在以下情况下可能会出现不足值： **Short counts can occur in these situations:**
  - 读取时遇到（文件结尾）EOF Encountering (end-of-file) EOF on reads
  - 从终端读取文本行 Reading text lines from a terminal
  - 读写网络套接字 Reading and writing network sockets
- 在以下情况下从不发生不足值： **Short counts never occur in these situations:**
  - 从磁盘文件读取（EOF除外） Reading from disk files (except for EOF)
  - 写入磁盘文件 Writing to disk files
- 最佳做法是始终允许不足值 **Best practice is to always allow for short counts**

RIO包（213/CS:APP包）
The RIO Package (213/CS:APP Package)

- RIO是一组包装器，可在应用程序中提供高效和健壮的I/O，例如易受不足值影响的网络程序 **RIO is a set of wrappers that provide efficient and robust I/O in apps, such as network programs that are subject to short counts**
- RIO提供两种不同的函数 **RIO provides two different kinds of functions**
  - 二进制数据的无缓冲输入和输出 Unbuffered input and output of binary data
    - **`rio_readn`** and **`rio_writen`**
  - 文本行和二进制数据的缓冲输入 Buffered input of text lines and binary data
    - **`rio_readlineb`** and **`rio_readnb`**
    - 缓冲RIO例程是线程安全的，可以在同一描述符上任意交错使用 Buffered RIO routines are thread-safe and can be interleaved arbitrarily on the same descriptor
- 从以下地址下载 **Download from** http://csapp.cs.cmu.edu/3e/code.html

无缓冲RIO输入和输出
Unbuffered RIO Input and Output

- 与Unix读写接口相同 Same interface as Unix **`read`** and **`write`**
- 特别适用于在网络套接字上传输数据 Especially useful for transferring data on network sockets

```
#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时返回不足值 **`rio_readn`** returns short count only if it encounters EOF
  - 只有当您知道要读取多少字节时才使用它 Only use it when you know how many bytes to read
- **`rio_writen`**从不返回不足值 **`rio_writen`** never returns a short count
- 对**`rio_readn`**和**`rio_writen`**的调用可以在同一描述符上任意交错使用 Calls to **`rio_readn`** and **`rio_writen`** can be interleaved arbitrarily on the same descriptor

rio_readn的实现 Implementation of rio_readn

```
/*
 * 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

缓冲RIO输入函数 Buffered RIO Input Functions

- 从部分缓存在内部内存缓冲区中的文件中有效读取文本行和二进制数据 **Efficiently read text lines and binary data from a file partially cached in an internal memory buffer**

```
#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);
```

返回：如果OK为读取字节数，遇到EOF为0，出错为-1

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 **rio_readlineb** reads a ***text line*** of up to **maxlen** bytes from file **fd** and stores the line in **usrbuf**
  - 特别适用于从网络套接字读取文本行 Especially useful for reading text lines from network sockets
- 停止条件 Stopping conditions
  - 读取到最大字节数 **maxlen** bytes read
  - 遇到EOF EOF encountered
  - 遇到换行符（"**\n**"） Newline ('**\n**') encountered

缓冲RIO输入函数(续)
Buffered RIO Input Functions (cont.)

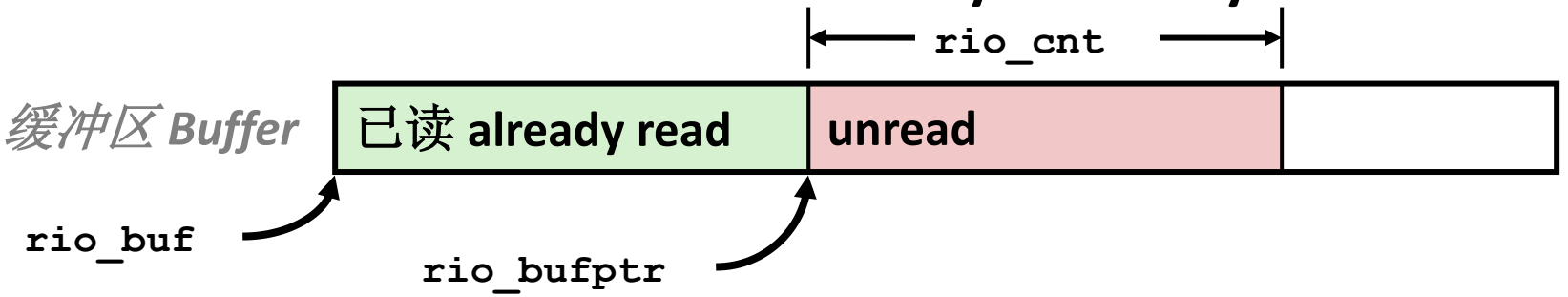
```
#include "csapp.h"

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);
```

返回：如果OK为读取字节数，遇到EOF为0，出错为-1
Return: num. bytes read if OK, 0 on EOF, -1 on error

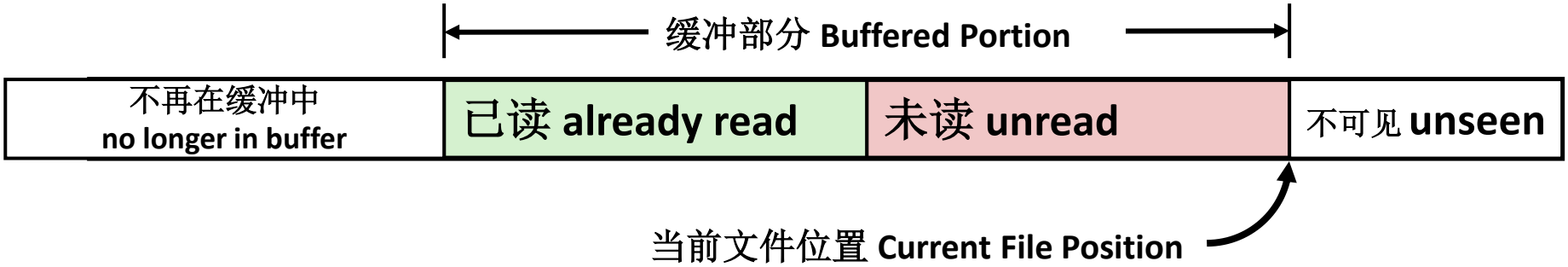
- **rio_readnb**从文件**fd**读取最多*n个字节* **rio_readnb** reads up to **n *bytes*** from file **fd**
- 停止条件 Stopping conditions
  - 读取到最大字节数 **maxlen** bytes read
  - 遇到EOF EOF encountered
- 对**rio_readlineb**和**rio_readnb**的调用可以在同一描述符上任意交错使用 Calls to **rio_readlineb** and **rio_readnb** can be interleaved arbitrarily on the same descriptor
  - 警告：不要与**rio_readn**调用交错使用 **Warning:** Don't interleave with calls to **rio_readn**

缓冲I/O：实现
Buffered I/O: Implementation

- 从文件读取 For reading from file
- 文件具有关联的缓冲区，用于保存已从文件读取但用户代码尚未读取的字节 File has associated buffer to hold bytes that have been read from file but not yet read by user code

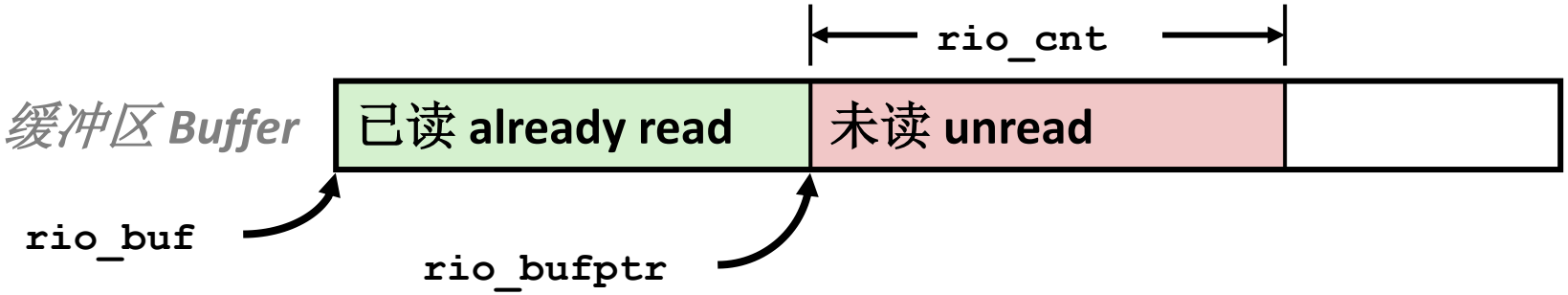


- Unix文件分层 Layered on Unix file:



缓冲I/O：声明 Buffered I/O: Declaration

- 结构包含所有信息 All information contained in `struct`




```
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;
```

标准I/O示例 Standard I/O Example

- 用rio逐行复制文件到标准输出 Copying file to `stdout`, line-by-line with rio

```c
#include "csapp.h"
#define MLINE 1024

int main(int argc, char *argv[])
{
    rio_t rio;
    char buf[MLINE];
    int infd = STDIN_FILENO;
    ssize_t nread = 0;
    if (argc == 2) {
        infd = Open(argv[1], O_RDONLY, 0);
    }
    rio_readinitb(&rio, infd);
    while((nread = rio_readlineb(&rio, buf, MLINE)) != 0)
        rio_writen(STDOUT_FILENO, buf, nread);
    exit(0);
}
```

showfile4_stdio.c

- 演示 Demo:

```
linux> strace ./showfile4_rio names.txt
```

议题

- Unix输入/输出 Unix I/O
- 元数据、共享和重定向 Metadata, sharing, and redirection
- 标准输入/输出 Standard I/O
- RIO（健壮I/O）包 RIO (robust I/O) package
- **闭幕词 Closing remarks**

标准输入/输出示例 Standard I/O Example

- 用mmap装载整个文件，复制文件到标准输出 Copying file to stdout, loading entire file with mmap

```
#include "csapp.h"


int main(int argc, char **argv)
{
    struct stat stat;
    if (argc != 2) exit(1);
    int infd = open(argv[1], O_RDONLY, 0);
    fstat(infd, &stat);
    size_t size = stat.st_size;
    char *bufp = mmap(NULL, size, PROT_READ,
                      MAP_PRIVATE, infd, 0);
    write(1, bufp, size);
    exit(0);
}
```

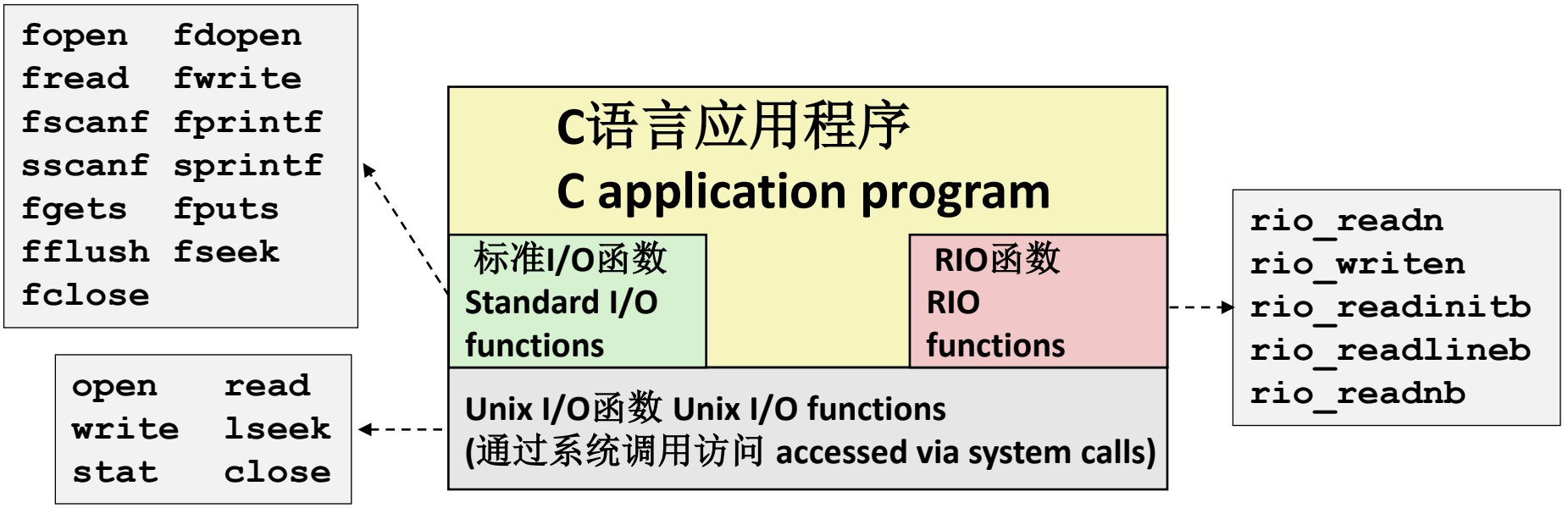
showfile5_mmap.c

- 演示 Demo:

```
linux> strace ./showfile5_mmap names.txt
```

Unix I/O与标准I/O与RIO比较
Unix I/O vs. Standard I/O vs. RIO

- 标准I/O和RIO使用低级Unix I/O实现 Standard I/O and RIO are implemented using low-level Unix I/O



- 在程序中应该使用哪一个？ Which ones should you use in your programs?

Unix I/O的优点和缺点
Pros and Cons of Unix I/O

- 优点 **Pros**
  - Unix I/O是最通用、开销最低的I/O形式 Unix I/O is the most general and lowest overhead form of I/O
    - 所有其他I/O包都使用Unix I/O函数实现 All other I/O packages are implemented using Unix I/O functions
  - Unix I/O提供了访问文件元数据的功能 Unix I/O provides functions for accessing file metadata
  - 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 Unix I/O functions are async-signal-safe and can be used safely in signal handlers

Unix I/O的优点和缺点
Pros and Cons of Unix I/O

- 缺点 **Cons**
  - 处理不足值很棘手，容易出错 Dealing with short counts is tricky and error prone
  - 高效读取文本行需要某种形式的缓冲，这也是一种棘手且容易出错的方法 Efficient reading of text lines requires some form of buffering, also tricky and error prone
  - 这两个问题都由标准I/O和RIO包解决 Both of these issues are addressed by the standard I/O and RIO packages

标准I/O的优缺点
Pros and Cons of Standard I/O

- 优点 **Pros:**
  - 缓冲通过减少读写系统调用的数量来提高效率 Buffering increases efficiency by decreasing the number of **`read`** and **`write`** system calls
  - 不足值自动处理 Short counts are handled automatically
- 缺点 **Cons:**
  - 不提供访问文件元数据的功能 Provides no function for accessing file metadata
  - 标准I/O函数不是异步信号安全的，不适用于信号处理程序 Standard I/O functions are not async-signal-safe, and not appropriate for signal handlers
  - 标准I/O不适用于网络套接字上的输入和输出 Standard I/O is not appropriate for input and output on network sockets
    - 对流的限制与对套接字限制互相冲突，极少有文档描述这些现象（CS:APP3e，第10.11节） There are poorly documented restrictions on streams that interact badly with restrictions on sockets (CS:APP3e, Sec 10.11)

选择输入/输出函数 Choosing I/O Functions

- 一般规则：使用您可以使用的最高级别I/O函数 **General rule: use the highest-level I/O functions you can**
  - 许多C程序员能够使用标准I/O函数完成所有工作 Many C programmers are able to do all of their work using the standard I/O functions
  - 但是，一定要理解使用的函数！ But, be sure to understand the functions you use!
- 何时使用标准I/O **When to use standard I/O**
  - 使用磁盘或终端文件时 When working with disk or terminal files
- 何时使用原始Unix I/O **When to use raw Unix I/O**
  - *信号处理程序内部，因为Unix I/O是异步信号安全的 Inside signal handlers, because Unix I/O is async-signal-safe*
  - 在极少数情况下，当需要绝对最高的性能时 In rare cases when you need absolute highest performance
- 何时使用RIO **When to use RIO**
  - *当读写网络套接字时 When you are reading and writing network sockets*
  - 避免对套接字使用标准I/O Avoid using standard I/O on sockets

旁注：使用二进制文件
Aside: Working with Binary Files

- 二进制文件 **Binary File**
  - 任意的字节序列 Sequence of arbitrary bytes
  - 包括字节值0x00 Including byte value 0x00
- **在二进制文件中*从不*应该使用的函数 Functions you should *never* use on binary files**
  - **面向文本的I/O：例如`fgets`、`scanf`、`rio_readlineb` Text-oriented I/O:** such as **`fgets`, `scanf`, `rio_readlineb`**
    - 解释EOF字符 Interpret EOL characters.
    - 使用`rio_readn`或`rio_readnb`这类函数代替 Use functions like **`rio_readn`** or **`rio_readnb`** instead
  - **字符串函数 String functions**
    - **`strlen`, `strcpy`, `strcat`**
    - 将字节值0（字符串结束）解释为特殊值 Interprets byte value 0 (end of string) as special

额外的幻灯片 Extra Slides

有趣的文件描述符（3）
Fun with File Descriptors (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

- 结果文件的内容是什么？ **What would be the contents of the resulting file?**
  - pqrswxyznef

访问目录 Accessing Directories

- 对目录推荐的操作仅有：读取目录条目 **Only recommended operation on a directory: read its entries**
  - **Dirent**结构包含关于目录条目的信息 **dirent** structure contains information about a directory entry
  - 当遍历目录条目时，DIR结构包含有关目录的信息 DIR structure contains information about directory while stepping through its entries

```
#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);
}
```

访问文件元数据的例子
Example of Accessing File Metadata

```
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

进一步的信息 For Further Information

- **Unix圣经 The Unix bible:**
  - **Unix环境高级编程** W. Richard Stevens & Stephen A. Rago, ***Advanced Programming in the Unix Environment***, 3rd Edition, Addison Wesley, 2013
    - 1993年Stevens经典版的更新 Updated from Stevens's 1993 classic text

- **Linux圣经 The Linux bible:**
  - Linux编程接口 Michael Kerrisk, The Linux Programming Interface, No Starch Press, 2010
    - 百科全书和权威 Encyclopedic and authoritative