汇编语言程序设计

第16讲 文件系统IO

Unix文件

- UNIX文件是一个m个字节的序列
  - B_0 , B_1 , …. , B_k , …. , B_{m-1}
- 所有I/O设备被表示为文件
  - /dev/sda2　(/usr 磁盘分区)
  - /dev/tty2　(终端)
- 甚至kernel也表达为一个文件
  - /dev/kmem　(kernel内存映像)
  - /proc　　(kernel数据结构)

Linux 内核提供了一种通过 /proc 文件系统在运行时访问内核内部数据结构、改变内核设置的机制；
proc文件系统是一个伪文件系统，它只存在内存当中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口

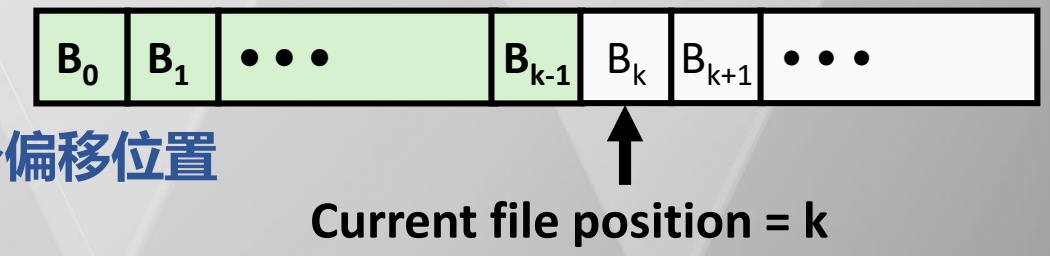
Unix文件类型

- 普通文件
  - 包含user/app数据的文件(binary, text, whatever)
  - OS不了解任何文件格式，只当做字节序列，类似主存
- 目录文件
  - 包含其他文件的名字和位置的文件
- 字符设备和块设备文件
  - 终端 (字符设备) 和磁盘 (块设备)
    - 字符设备：此类设备支持按字节/字符来读写数据；提供连续的数据流，应用程序可以顺序读取，通常不支持随机存取
    - 块设备：应用程序可以随机访问设备数据，程序可自行确定读取数据的位置

……

Unix I/O

- 主要特征
  - 将文件以“优雅”的方式映射到设备，对外提供简单IO访问接口
    - 重点: 所有的输入和输出以一种一致且单一的方式处理
- 基本UNIX IO操作 (系统调用)
  - 打开和关闭文件
    - open()and close()
  - 读写文件
    - read() and write()
  - 改变当前文件的位置 (seek)
    - 指示文件中将要读取或写入的下一个偏移位置
    - lseek()



打开文件

- 告知kernel准备访问该文件
  - 返回一个小的整数，文件描述符
    - fd == -1 表示出错

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

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

  - 每个由UNIX SHELL创建的进程以三个与终端关联的打开文件开始
    - 0: standard input
    - 1: standard output
    - 2: standard error

关闭文件

- 通知kernel准备结束访问该文件

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

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

- 检查close的返回码是一个好习惯，因为可能之前的write操作的错误是在最终的close时第一次报告
  - 启示: 经常检查返回码，甚至包括对close这样看上去没有危害的调用也要检查

Tips-1：当一个进程终止时，内核对该进程所有尚未关闭的文件描述符调用close关闭；

Tips-2：成功的关闭并不确保数据已经成功地保存到磁盘，因为内核会延迟写操作。如果需要确保，请在关闭前使用*fsync*

读文件

- 从当前文件位置拷贝数据到内存中，并且更新该位置

```
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);
}
```

  - 从文件fd中读取数据放入buf
    - 返回类型ssize_t 是有符号整数
    - nbytes < 0 表示错误发生
    - 返回的读取字节数少于预期是可能的，并不是错误 (nbytes < sizeof(buf))

写文件

- 从内存中拷贝数据到当前文件位置，并且更新该位置

```
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写入文件fd的字节数
  - nbytes < 0 表示错误发生
  - 和读操作一样，可能也不一定写入预期的字节数

简单的UNIX IO例子

- 从标准输入复制数据到标准输出，每次一个字节

```
#include "csapp.h"

int main(void)
{
    char c;

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

http://csapp.cs.cmu.edu/public/code.html

gcc -I../include ../src/csapp.c xxx.c -lpthread -o xxx.exe

处理读写“不足”

- 实际读写字节数与预期相比不足在这些情况下发生:
  - 读到了文件尾 （EOF）
  - 从终端读入文本行
  - 从网络socket读写数据
- 读写不足不会在以下情况发生:
  - 从磁盘文件读取数据（除了EOF）
  - 向磁盘文件写入数据
- 一种在代码中处理不足值的方式:
  - 使用RIO库(Robust I/O)

RIO包

- RIO是一组封装好的调用，为应用程序提供有效且鲁棒的IO，比如读写字节数未定的网络程序
- RIO提供两种不同的功能
  - 无缓存输入和二进制数据输出
    - rio_readn and rio_writen
  - 二进制数据以及文件行的缓存输入
    - rio_readlineb and rio_readnb
  - 下载http://csapp.cs.cmu.edu/public/code.html
    - 里头的src/csapp.c and include/csapp.h

文件元数据

- **Metadata （元数据）是有关数据的数据，此处指针对文件的元数据**
  - **每个文件的元数据由内核维护 / 用户通过stat和fstat调用来进行访问**

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

访问文件元数据的例子

```
/* statcheck.c - Querying and manipulating a file's meta data */
#include "csapp.h"

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

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

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

statcheck.c

参 数	说 明	参 数	说 明
S_IRUSR	所有者拥有读权限	S_IXGRP	群组拥有执行权限
S_IWUSR	所有者拥有写权限	S_IROTH	其他用户拥有读权限
S_IXUSR	所有者拥有执行权限	S_IWOTH	其他用户拥有写权限
S_IRGRP	群组拥有读权限	S_IXOTH	其他用户拥有执行权限
S_IWGRP	群组拥有写权限		

访问目录

- 对目录操作: 读取其条目
  - dirent 结构包含有关一个目录条目的信息
  - DIR 结构包含当进入其条目后的相关目录信息

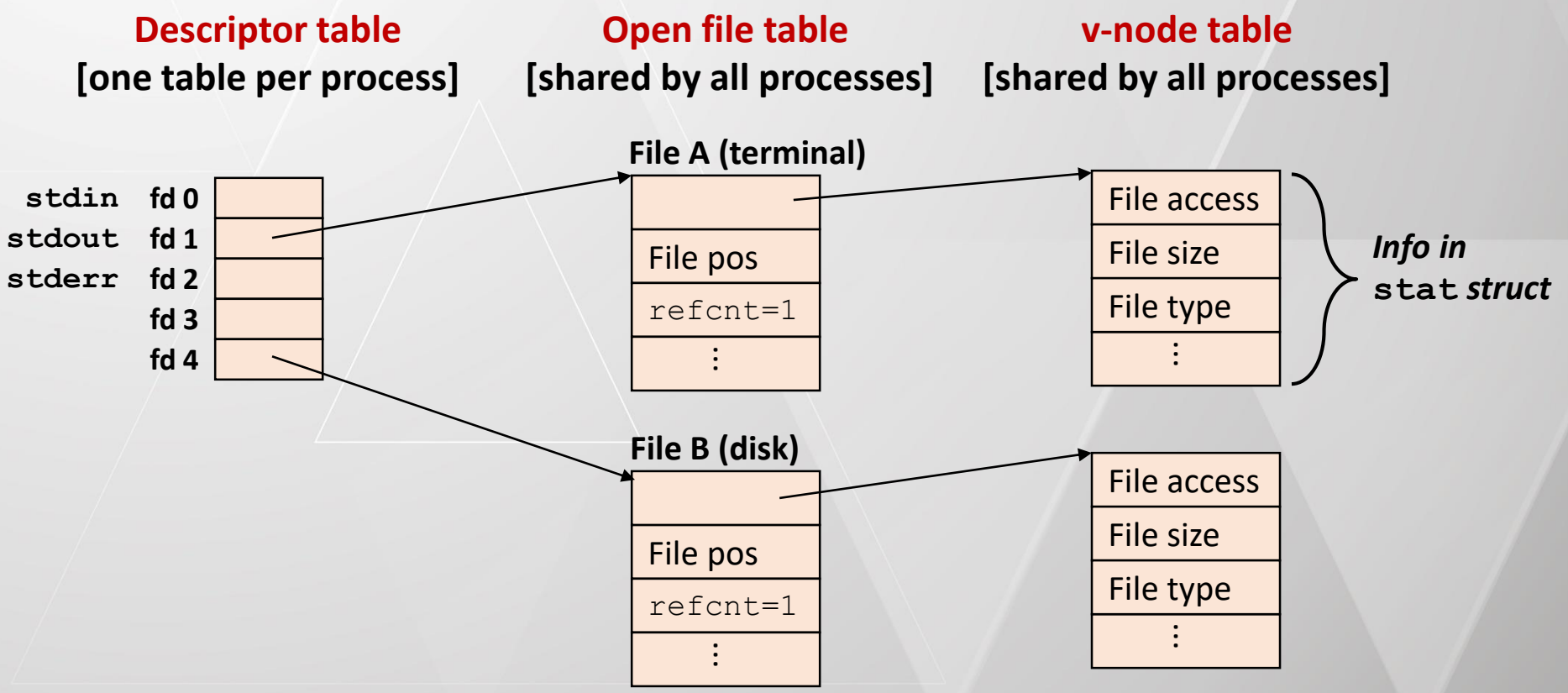
```
#include <sys/types.h>
#include <dirent.h>

{
  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);
}
```

```
struct dirent
{
   long d_ino;                              /*inode number   索引节点号 */
   off_t d_off;       /* offset to this dirent    在目录文件中的偏移 */
   unsigned short d_reclen;     /* length of this d_name 文件名长 */
   unsigned char d_type;          /* the type of d_name   文件类型 */
   char d_name [NAME_MAX+1];/* file name(null结尾)文件名最长255字符*/
}
```

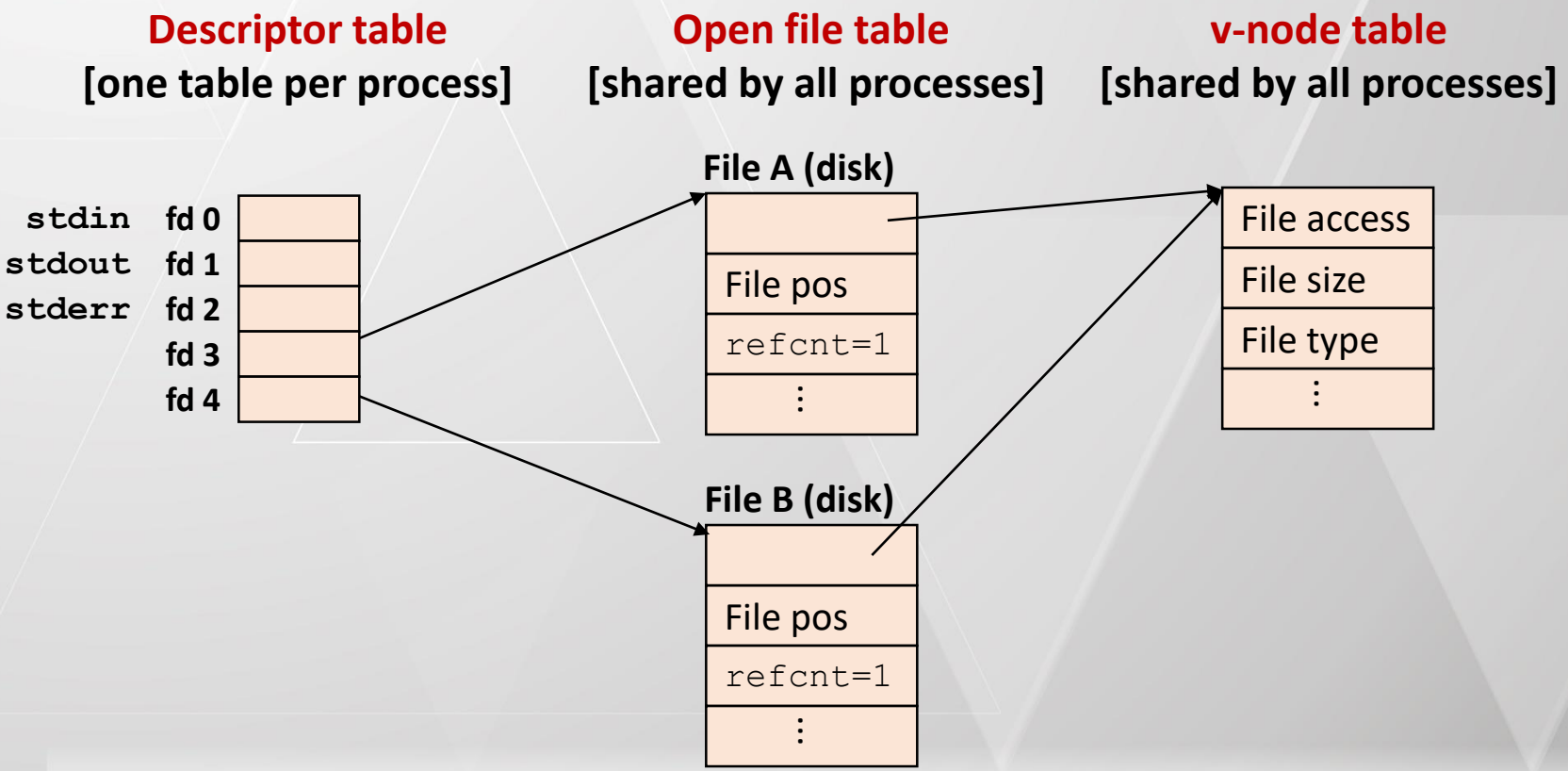
Kernel如何表示打开的文件

- 引用两个不同磁盘文件的两个文件描述符
  - 描述符1 (stdout) 指向终端, 描述符4 指向一个打开的磁盘文件



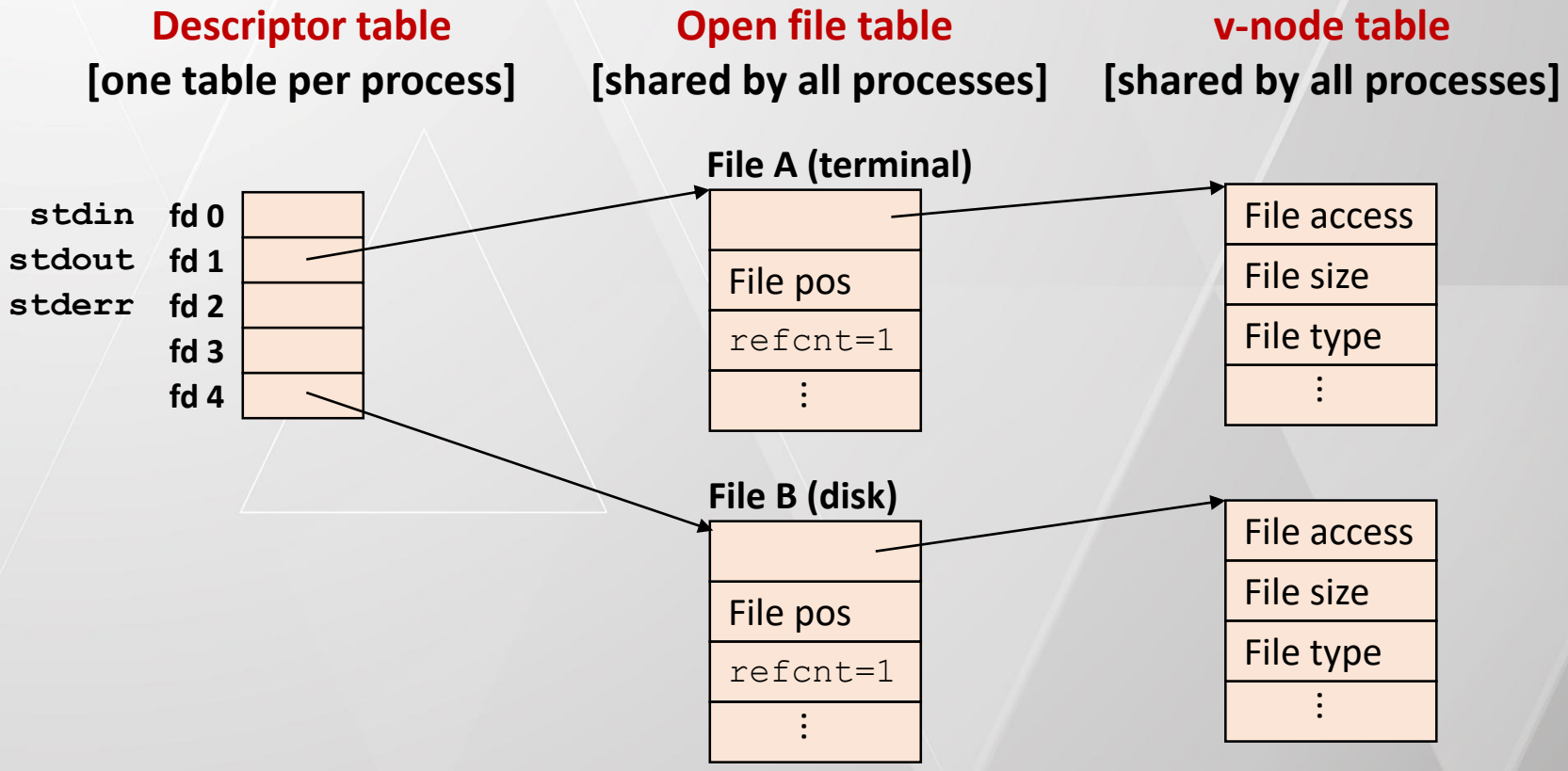
文件共享

- 两个独立的描述符通过两个独立的打开文件表条目共享同一磁盘文件
  - 比如，对同一个filename调用两次open调用



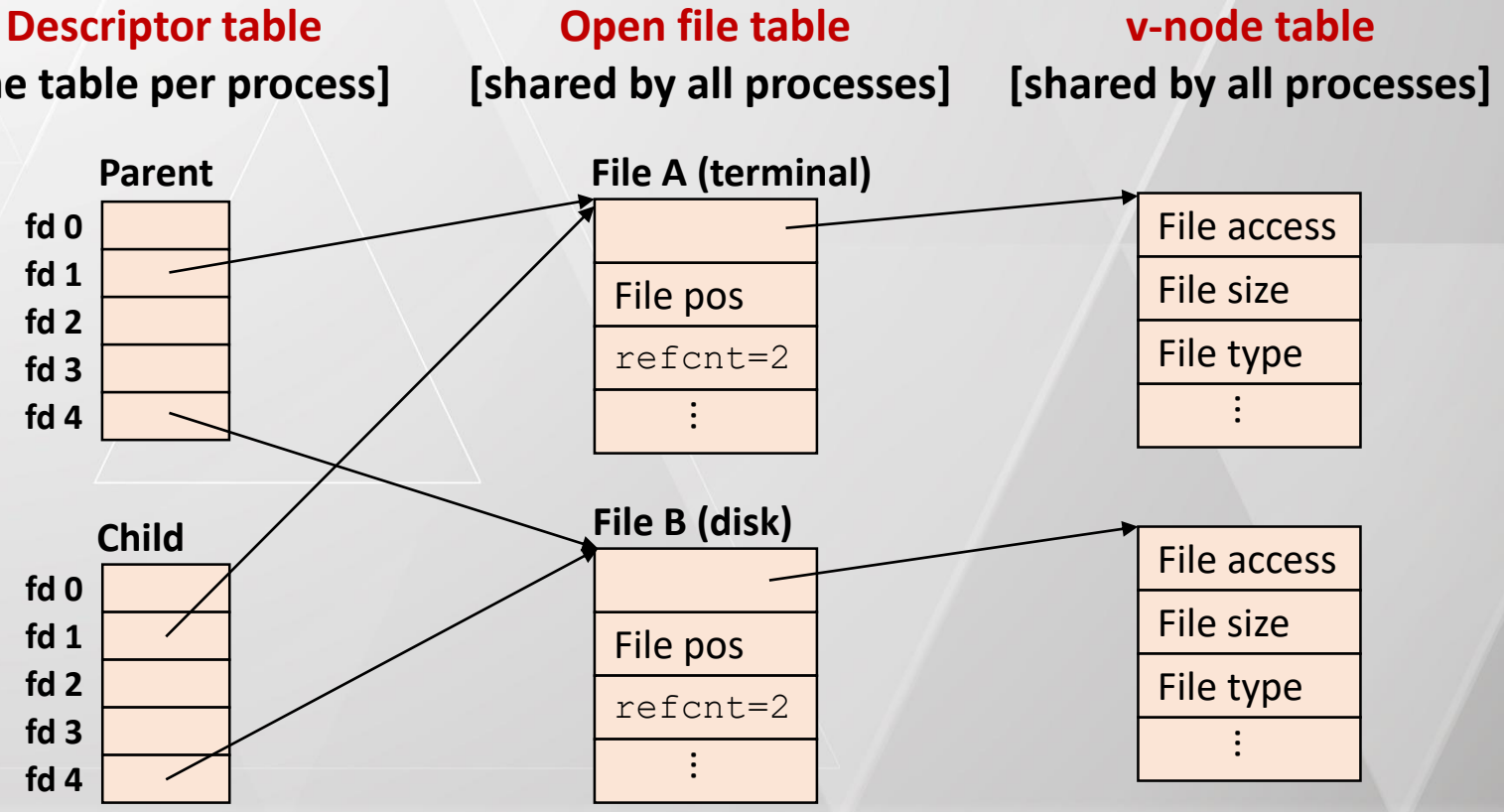
进程间如何共享文件: Fork()

- 子进程继承其父进程的打开文件
  - 在fork() 调用之前:



进程间如何共享文件: Fork()

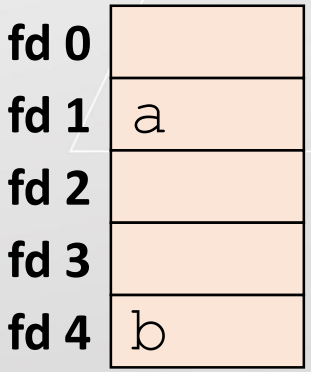
- 子进程继承其父进程的打开文件
  - fork() 调用后
    - 子进程的文件描述符表与父进程相同，引用计数加一 *(refcnt+1)*



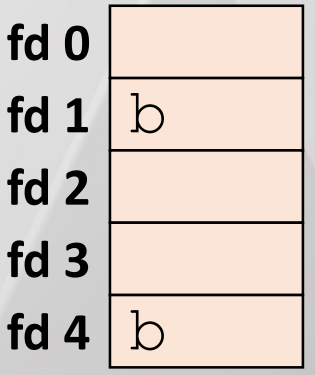
I/O 重定向

- 问题: shell如何实现IO重定向?
  - unix> ls > foo.txt
- 回答: 通过调用 dup2(oldfd, newfd)函数
  - 每个进程复制描述符表 oldfd 到 newfd

Descriptor table
before dup2(4,1)

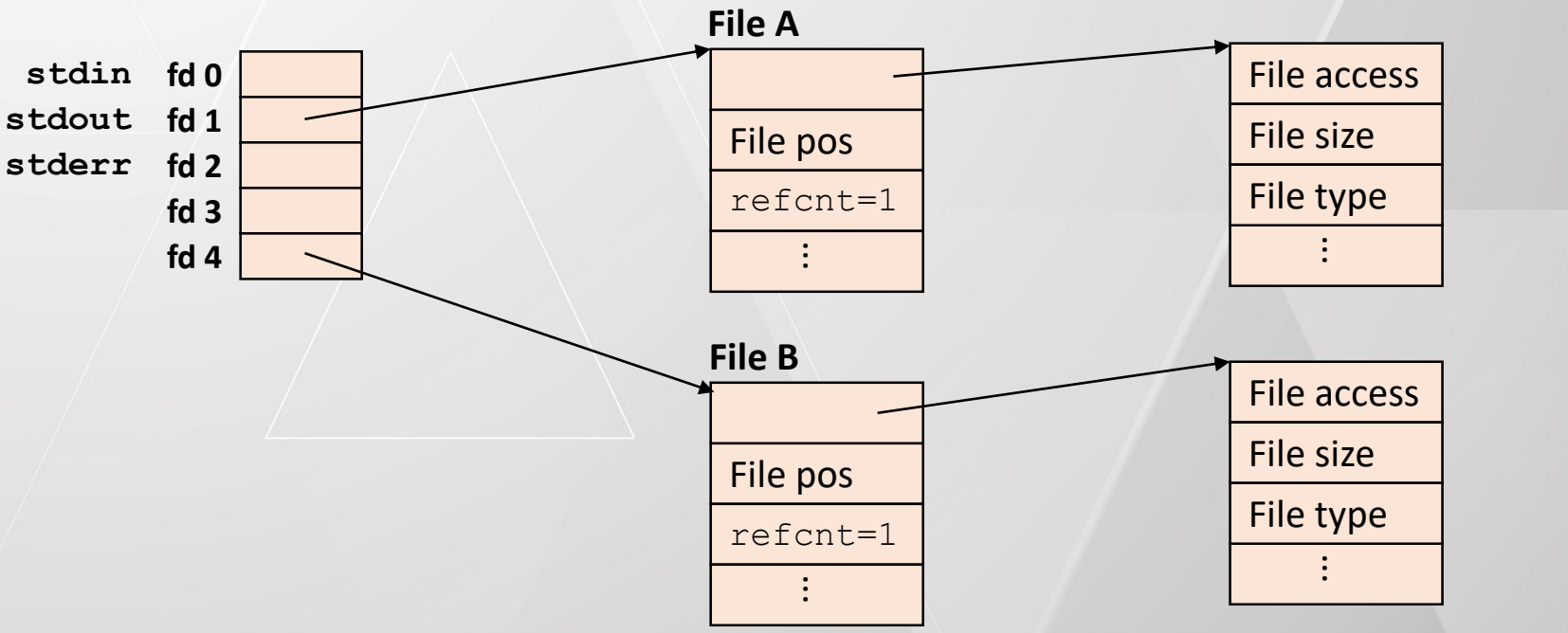


Descriptor table
after dup2(4,1)



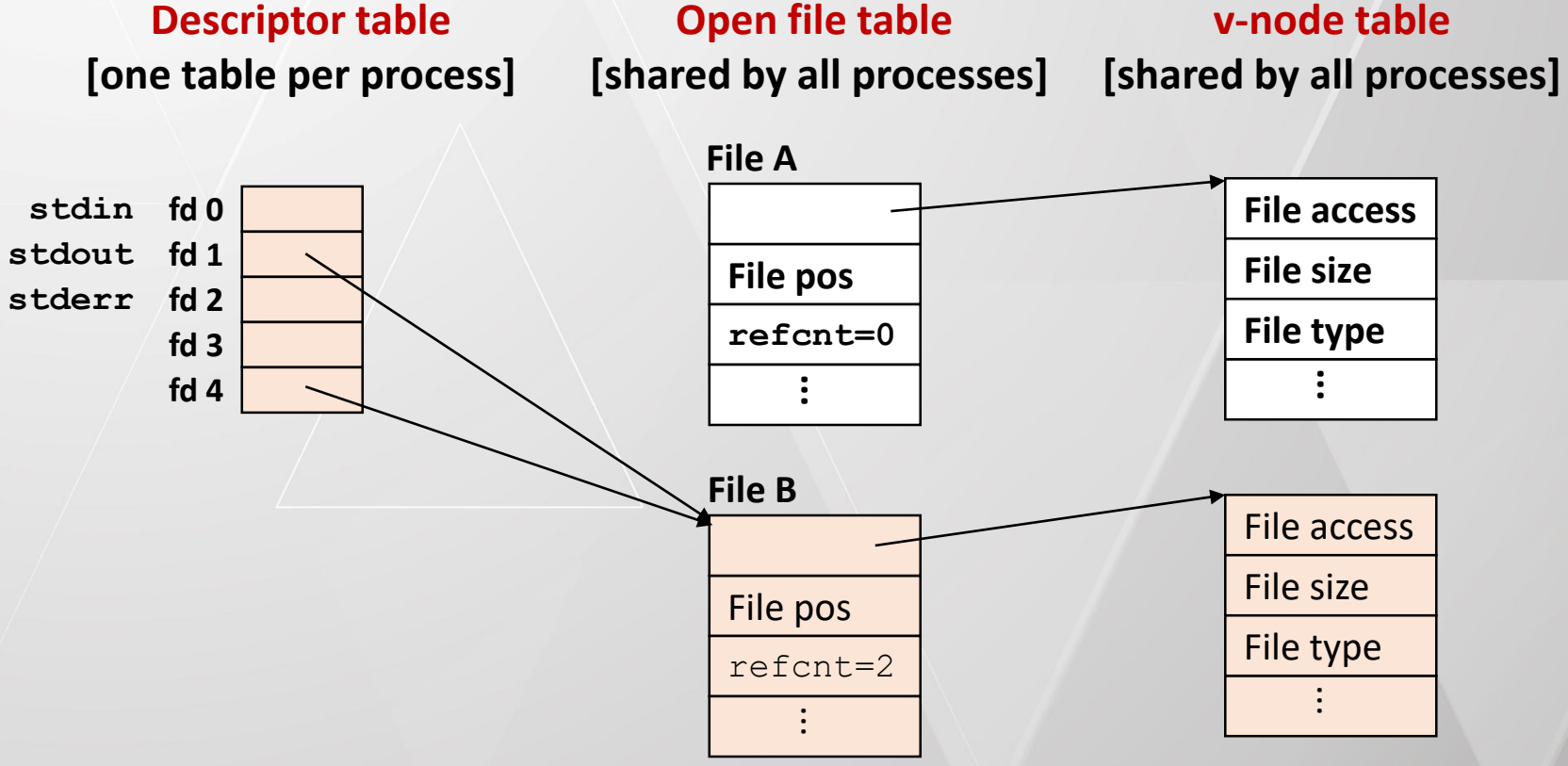
I/O 重定向

- 步骤 #1: 打开要将stdout定向到的文件



I/O 重定向

- 步骤 #2: 调用 dup2(4,1)
  - 导致 fd=1 (stdout) 指向由fd=4 指向的磁盘文件



文件示例-1

```
#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

- 该程序运行结果？文件包含“abcde”

文件示例-2

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1;
    int s = 1; //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

- 该程序运行结果？文件包含“abcde”

文件示例-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?

标准IO函数

- C标准库(libc.so)包含一组高级的标准IO函数

- 标准IO库的例子:
  - Opening and closing files (fopen and fclose)
  - Reading and writing bytes (fread and fwrite)
  - Reading and writing text lines (fgets and fputs)
  - Formatted reading and writing (fscanf and fprintf)

标准文件IO

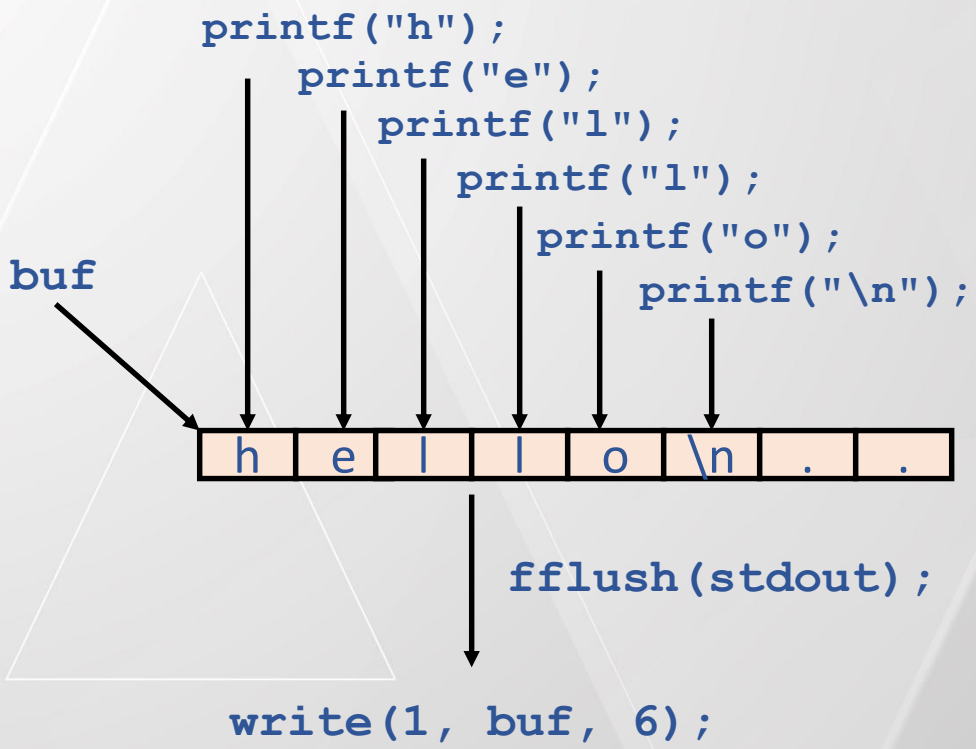
- 标准IO将打开的文件模型化为流
  - 维护对文件描述符及其对应的内存缓冲区的抽象
  - C 程序以三个打开操作开始 (defined in stdio.h)
    - stdin (standard input)
    - stdout (standard output)
    - stderr (standard error)

```
#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");
}
```

标准文件IO

- 标准IO函数使用带缓冲的IO



- 缓冲区碰到*换行符*或者fflush调用后被刷到输出fd中

标准文件IO的实际运行

- 可以自己看到缓冲机制的实际运行, 使用Unix strace 程序:

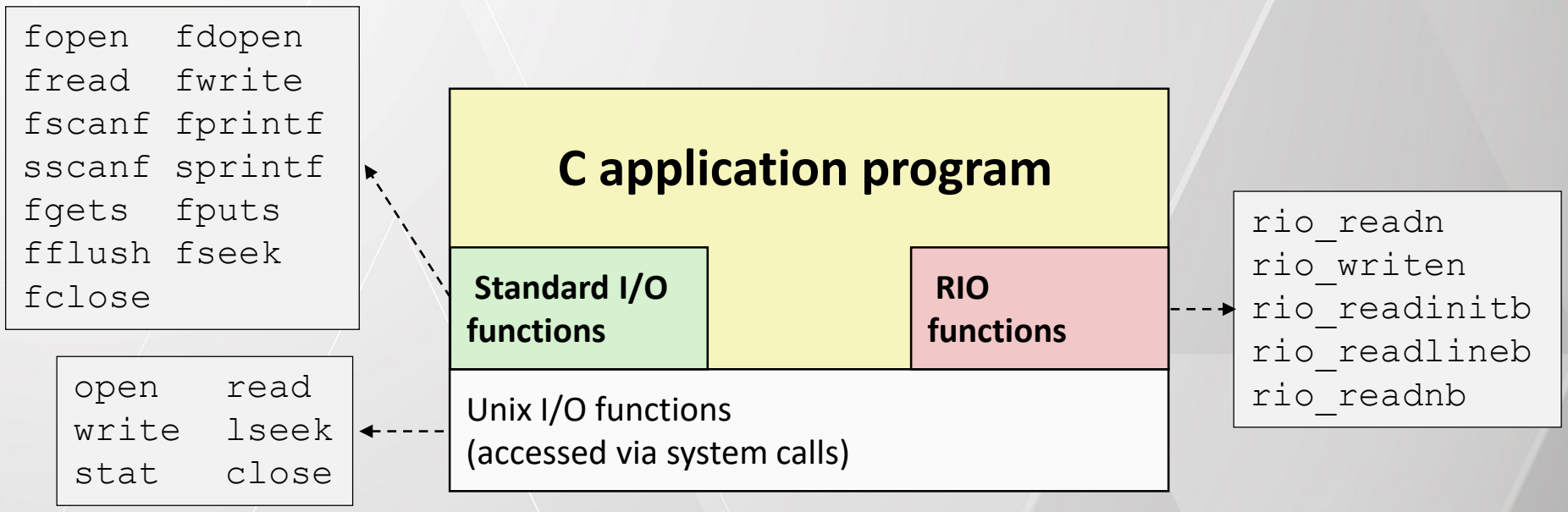
```
#include <stdio.h>

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)                           = ?
```

Unix I/O vs. 标准I/O vs. RIO

- 标准 I/O 和 RIO 都是使用底层UNIX IO实现的



- 程序中选用哪个?

优劣：Unix I/O

- 优点
  - Unix IO是最通用而且额外开销最小的IO形式
  - 所有其他IO包都是使用UNIX IO来实现的
  - Unix IO提供了访问文件元数据的功能
- 缺点
  - 对不足值的处理不够，容易出错
  - 有效读取文本行需要某种形式的缓冲，容易出错
  - 这些问题在标准IO和RIO包中都得到解决.

优劣：标准I/O

- 优点
  - 通过减小read和write调用的数目，缓冲增加了效率
  - 不足值可以自动处理
- 缺点
  - 没有提供访问文件元数据的功能
  - 标准IO程序并非同步信号安全的，也不适合在信号处理程序中采用.
  - 标准IO不适合于对网络socket的输入输出进行处理

选择IO函数

- 通用规则: 使用能用的最高级别的IO函数
  - 许多C程序员可以使用标准IO函数来做所有工作
- 什么时候使用标准IO
  - 当处理磁盘和终端文件的时候
- 什么时候使用裸UNIX IO
  - 信号处理程序，因为只有它是同步信号安全的
  - 在很少的情况下，当你需要更高性能的时候
- 什么时候使用RIO
  - 读写网络套接字的时候.
  - 避免对socket使用标准IO

二进制文件

- 二进制文件
  - Object code, Images (JPEG, GIF),
- 不能用在二进制文件上的函数
  - 基于行处理的IO：fgets, scanf, printf, rio_readlineb
    - 不同系统解释换行符的方式不同0x0A（‘\n’）:
      - Linux 和 Mac OS X：LF(0x0a) [‘\n’]
      - HTTP 服务器 & Windows：CR+LF(0x0d 0x0a) [‘\r\n’]
    - 使用 rio_readn 或 rio_readnb 替代
  - 串处理函数
    - strlen, strcpy
    - 解释字节0为串结束