

目黒区
市山科区

日本スタンダードサ

稲沢市治郎丸大明町
東京都中野区新井一ノ三六一
浜市青葉区樺が丘五ノ五 味処

黄
直上
神空

2024/11/27

System Level I/O

13 元培数科 常欣海

Here we go! →



Knowledge Review

This part is cited from Arthals

Unix I/O

Unix I/O

- 所有的 I/O 设备都被模型化为 **文件**
- 输入输出 **等价于文件的读写**

文件的读写

File Read/Write

- 每次打开文件，都会有唯一非负整数 **文件描述符** 与之对应
- Shell 创建时的默认三个文件描述符：
 - 0 标准输入 STDIN_FILENO
 - 1 标准输出 STDOUT_FILENO
 - 2 标准错误 STDERR_FILENO
- 文件的读：复制文件的内容到内存，可能会遇到 EOF (End of File)
- 文件的写：复制内存的内容到文件

文件类型

File Type

- 普通文件 `-`：文本文件和二进制文件
- 目录文件 `d`：文件夹 / 目录文件，一个目录至少含有两个条目，一个指向自己本身 (`.`)，一个指向其父亲 (`..`)
- 套接字文件 `s`：和另外一个进程进行跨网络通信的 **文件**

```
cd /var/run
eza -l --grid # eza 是一个 ls 的替代品，功能更强大
cd ~
mkdir test && cd test
eza -laa # -l 列出文件的详细信息，-a 列出所有文件，包括隐藏文件，两个 -a 启用对 . / .. 的显示
```

不要认为套接字这个拗口的词汇和文件有什么本质的不同，它就是一层抽象，使得网络通信和文件一样

会略有差异，但是概念是相通的。在学未来的网络编程时，尤其需要明确这一点。

文件路径

File Path

- 绝对路径：从根目录 `/` 开始的路径
- 相对路径：从当前目录 `.` 开始的路径

尝试运行：

```
ls / # 列出根目录下的所有文件  
ls . # 列出当前目录下的所有文件
```

其他路径：

- `..` 父目录
- `~` 当前用户的主目录，在 macOS / Linux 中是 `/Users/username`

目录层次结构

Directory Hierarchy

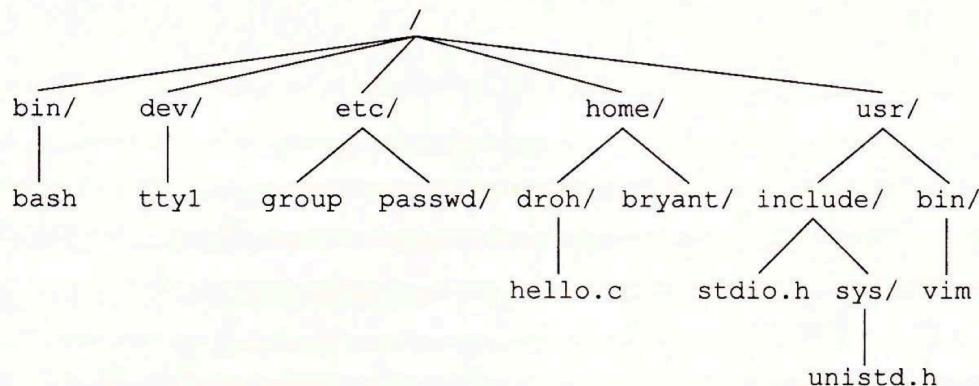


图 10-1 Linux 目录层次的一部分。尾部有斜杠表示是目录

你可以使用 `tree` 命令来查看当前目录的层次结构：

```
tree  
# 如果太多，可以使用 -L 2 来限制层级  
tree -L 2
```

打开文件

Open File

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

参数:

- pathname 文件路径，绝对路径或当前路径下文件名（可以使用相对路径）
- flags 打开文件的方式
- mode 新文件的权限

返回值: int

- 成功: 返回文件描述符
- 失败: 返回 -1，并设置 errno

打开文件 - Flags

Open File - Flags

常量	含义
O_RDONLY	只读打开, Read Only
O_WRONLY	只写打开, Write Only
O_RDWR	读写打开, Read Write
O_CREAT	如果文件不存在, 则创建它, 需要 mode , Create
O_TRUNC	如果文件存在, 且以写模式 (O_WRONLY 或 O_RDWR) 打开, 则将其长度截断为 0, Truncate
O_APPEND	在每次写操作前, 设置文件位置到文件结尾处 (是写操作前而不是就在打开文件时) , Append

前三个必选其一, 后三个可选。类似上节课讲的, 我们可以用 | 来组合多个标志。

如 O_RDWR | O_CREAT | O_TRUNC 表示以读写模式打开文件, 如果文件不存在则创建它, 若存在则截断文件长度为 0。

This slide is cited from Arthals.

打开文件 - Mode

Open File - Mode

新文件的访问权限位，每个进程还额外有 `umask` 掩码，用于限制新文件的权限。

当进程创建新文件时，新文件的权限位是 `mode & ~umask`。

```
eza -l ~
```

假设得到结果 `drwxrwxr-x`，其代表：

```
d | rwx | rwx | r-x
```

- 文件类型：目录
- 文件所有者权限：读、写、执行
- 文件所有者所在组权限：读、写、执行
- 其他用户权限：读、执行

打开文件 - Mode

Open File - Mode

umask 值是一个三位八进制数，每一位分别对应文件权限中的用户（owner）、组（group）和其他人（others）。它表示要从默认权限中屏蔽的位。

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

mode_t umask(mode_t mask);
```

打开文件 - Mode

Open File - Mode

`mode` 值是一个四位八进制数，一般使用常量来指定，在创建新文件时生效，其组成包括：

- 特殊权限位
- 文件所有者权限
- 文件所有者所在组权限
- 其他用户权限

常量：

- `S_IRUSR`、`S_IWUSR`、`S_IXUSR`：用户（`USR`, `user`）对文件的读/写/执行权限。
- `S_IRGRP`、`S_IWGRP`、`S_IXGRP`：用户组（`GRP`, `group`）对文件的读/写/执行权限。
- `S_IROTH`、`S_IWOTH`、`S_IXOTH`：其他人（`OTH`, `other`）对文件的读/写/执行权限。

关闭文件

Close File

```
int close(int fd);
```

返回值: int

- 成功: 返回 0
- 失败: 返回 -1, 并设置 errno

关闭一个已经关闭的描述符会出错。

无论进程因为什么而终止, 内核都会关闭所有打开的文件并释放内存资源。

读和写

Read and Write

读 read

```
ssize_t read(int fd, void *buf, size_t n);
```

- `fd` 文件描述符
- `buf` 缓冲区，用于存储读取到的数据
- `n` 读取的字节数

返回值: `ssize_t` (有符号整数, `signed size_t`)

- 成功: **返回读取的字节数**
- 读到 EOF: 返回 `0` 也不算失败, 但也不算完全正常?
- 失败: 返回 `-1`, 并设置 `errno`

写 write

```
ssize_t write(int fd, const void *buf, size_t n);
```

- `fd` 文件描述符
- `buf` 缓冲区, 用于存储写入的数据
- `n` 写入的字节数

返回值: `ssize_t` (有符号整数, `signed size_t`)

- 成功: **返回写入的字节数**
- 失败: 返回 `-1`, 并设置 `errno`

当返回非负, 但是小于预期的 `n` 时, 表示写入操作只成功了一部分, 我们称之为不足值

不足值

Short Count

不足值: $-1 < \text{ret} < n$

不足值的出现原因:

- 读到 EOF: (假设 $n = 50$) 此次返回到 EOF 时已读取的数量 < 50 , 比如 20, 下次再读取时返回 0
- 从终端传输文本行: 一般设置读取的字节数较大, 但是每次读取的文本行可能较短, 所以返回的字节数会小于请求的字节数
- 读和写网络套接字 (Socket) : 如果打开的文件对应于网络套接字, 那么内部缓冲约束和较长的网络延迟会引起 `read` 和 `write` 返回不足值 (回忆一下信号中断)

由于不足值的存在, 为了能够保证输入输出能够正常的、按照预期的完成 (主要是情况 3), 我们可能需要反复、多次调用 `read` 和 `write`。

RIO 包

Robust I/O

让你的 I/O 健壮一点~

RIO - 不带缓冲的读

rio_readn

运行到这的唯二两种可能：

- `nleft == 0`，完全读入成功，返回 `n`
- EOF 导致退出，返回 `n - nleft`，即实际读入的字节数。

```
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) /* 被信号处理程序中断返回 */
                nread = 0;      /* 再次调用 read() */
            else
                return -1;     /* read() 设置了 errno */
        }
        else if (nread == 0)
            break;           /* EOF */
        nleft -= nread;
        bufp += nread;
    }
    return (n - nleft);          /* 返回值 >= 0 */
}
```

RIO - 不带缓冲的写

rio_written

rio_written 函数用于从文件描述符 fd 中写入 n 个字节到缓冲区 usrbuf 中。

它是 **无缓冲** 的，与 write 相比，多了对不足值的处理。

和 readn 行为类似，但是没有了对于 EOF 的处理，所以永远不会返回不足值（要么返回错误 -1，要么返回 n）。

```
ssize_t rio_written(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nwritten;
    char *bufp = usrbuf;

    while (nleft > 0) {
        if ((nwritten = write(fd, bufp, nleft)) <= 0) {
            if (errno == EINTR) /* 被信号处理程序中断返回 */
                nwritten = 0; /* 再次调用 write() */
            else
                return -1; /* write() 设置了 errno */
        }
        nleft -= nwritten;
        bufp += nwritten;
    }
    return n; /* 返回原始请求的字节数 */
}
```

RIO - 带缓冲的读

`rio_read` / `rio_readnb` / `rio_readlineb`

`rio_readlineb`

`rio_readlineb` 函数从文件 `rp` 读出下一个文本行（包括结尾的换行符），将它复制到内存位置 `usrbuf`，并且用 `NULL` (零) 字符来结束这个文本行。

`rio_readlineb` 函数最多读 `maxlen-1` 个字节，余下的一个字符留给结尾的 `NULL` 字符。

超过 `maxlen-1` 字节的文本行会被截断。

由于核心的 57 行实现是基于 `rio_read` 函数的，所以我们可以肆无忌惮地使用它，每次只读 1 个字节，而不用顾虑系统调用次数。

```

50
51 // rio_readlineb - 健壮地读入一行, 带缓冲区
52 ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen) {
53     int n, rc;
54     char c, *bufp = usrbuf;
55
56     for (n = 1; n < maxlen; n++) {
57         if ((rc = rio_read(rp, &c, 1)) == 1) {
58             *bufp++ = c;
59             if (c == '\n')
60                 break;
61         } else if (rc == 0) {
62             if (n == 1)
63                 return 0; /* EOF, 没有读入任何数据 */
64             else
65                 break; /* EOF, 读入了部分数据 */
66         } else
67             return -1; /* 错误 */
68     }
69     *bufp = 0;
70     return n;
71 }
```

RIO 总结

Robust I/O Summary

缓冲区的实现：基于 `rio_t` 结构体，在此基础上封装出了 `rio_read` / `rio_readinitb` / `rio_readnb` / `rio_readlineb` 函数。

缓冲区的思想：通过一次读入很多字节，实现一个缓存机制，后续再读直接从缓存中取，从而减少系统调用次数。

不带缓冲区的实现：实现了对不足值的处理，包括 EOF / 系统调用中断等，在此基础上封装出了 `rio_readn` / `rio_writen` 函数。

带缓冲的版本和不带缓冲的版本，在实现上有很多类似的地方，**但一定不能混用**。

例：`rio_readlineb` 和 `rio_readnb` 可以混用，但不能和 `rio_readn` 混用。

共享文件

Shared Files

数据结构	描述	共享性	关键内容
描述符表	每个进程都有独立的描述符表	不共享	描述符表项由文件描述符索引，指向文件表项
文件表	表示打开文件的集合	所有进程共享	文件位置、引用计数、v-node 指针
v-node 表	包含文件的 <code>stat</code> 结构信息	所有进程共享	<code>st_mode</code> 和 <code>st_size</code>

这么类比着记：

- 描述表，每个进程单独一个，符合直觉
- v-node 是类的声明，一个真实文件只有一个，所有进程共享
- 打开文件表是类的实例，所有进程共享，每个实例必然有与之对应的声明（v-node），但是多个实例可以指向同一个声明（v-node）
- 多个进程可以共用同一个文件表，类比他们共用相同的实例。

共享文件

Shared Files

打开文件表和 v-node 表的最核心差异：

- 打开文件表的每一项 **对应一次打开**
- v-node 表的每一项 **对应一个文件**

由于一个文件可以打开多次，所以可以有多个打开文件表项指向同一个 v-node 表项。

由于多个文件可以共享一次打开，所以有了打开文件表中的 `refcnt`，多个进程的描述表中的描述符可以对应同一个打开文件表项（如父子进程）。

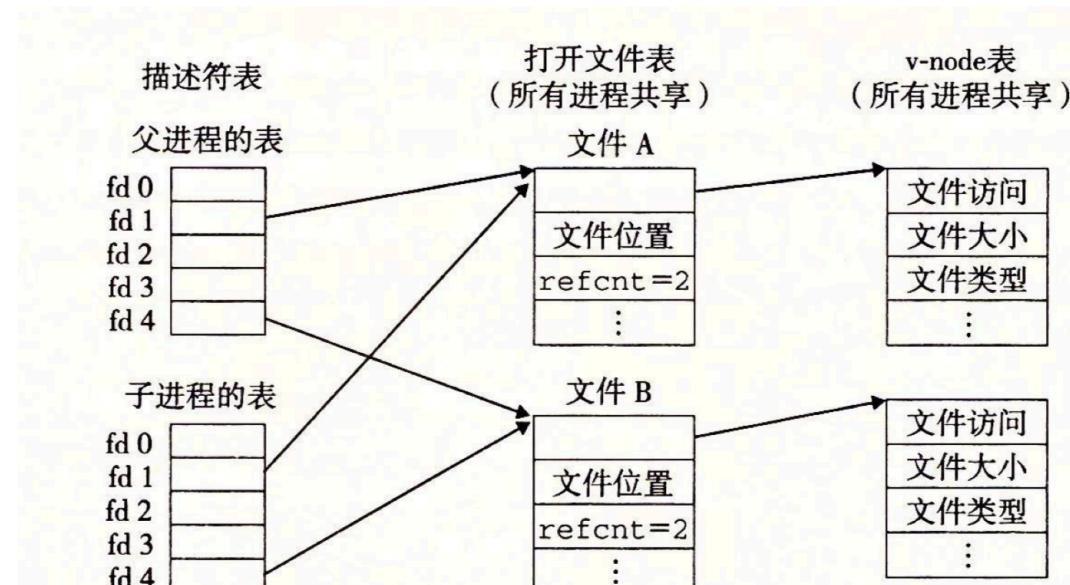


图 10-14 子进程如何继承父进程的打开文件。初始状态如图 10-12 所示

共享文件 - 总结

Shared Files - Summary

描述符表

- 每个进程都有独立的描述符表
- 描述符表的表项由进程打开的文件描述符来索引
- 每个打开的描述符表项指向文件表中的一个表项
- **对应一个进程所需要的描述符信息**

文件表

- 打开的文件集合由文件表表示，所有进程共享该表。
- 文件表的表项包括：
 - 文件位置：当前文件位置
 - 引用计数 `refcnt`：当前指向该表项的描述符表项数量
 - v-node 指针：指向 v-node 表中对应表项的指针
- 关闭一个描述符会减少相应文件表表项的引用计数
- 内核不会删除文件表表项，直到引用计数为零
- **对应一次打开**

v-node 表

- v-node 表同样由所有进程共享
- 每个表项包含 `stat` 结构中的大多数信息，包括 `st_mode` 和 `st_size` 成员
- **对应一个文件**

I/O 重定向

I/O Redirection

dup

```
int dup(int oldfd);
```

`dup` 用于复制一个文件描述符 `oldfd`，并返回一个新的文件描述符。

新的文件描述符与原来的文件描述符共享同一个文件表项。

返回值

- 成功时，返回新的文件描述符。
- 失败时，返回 `-1`，并设置 `errno` 以指示错误。

dup2

```
int dup2(int oldfd, int newfd);
```

`dup2` 复制文件描述符 `oldfd` 到 `newfd`。

注意顺序：是用前面的覆盖后面的，也就是把后面的指向前面

如果 `newfd` 已经打开，它会首先被关闭。

返回值

- 成功时，返回 `newfd`。
- 失败时，返回 `-1`，并设置 `errno` 以指示错误。

I/O 重定向

I/O Redirection

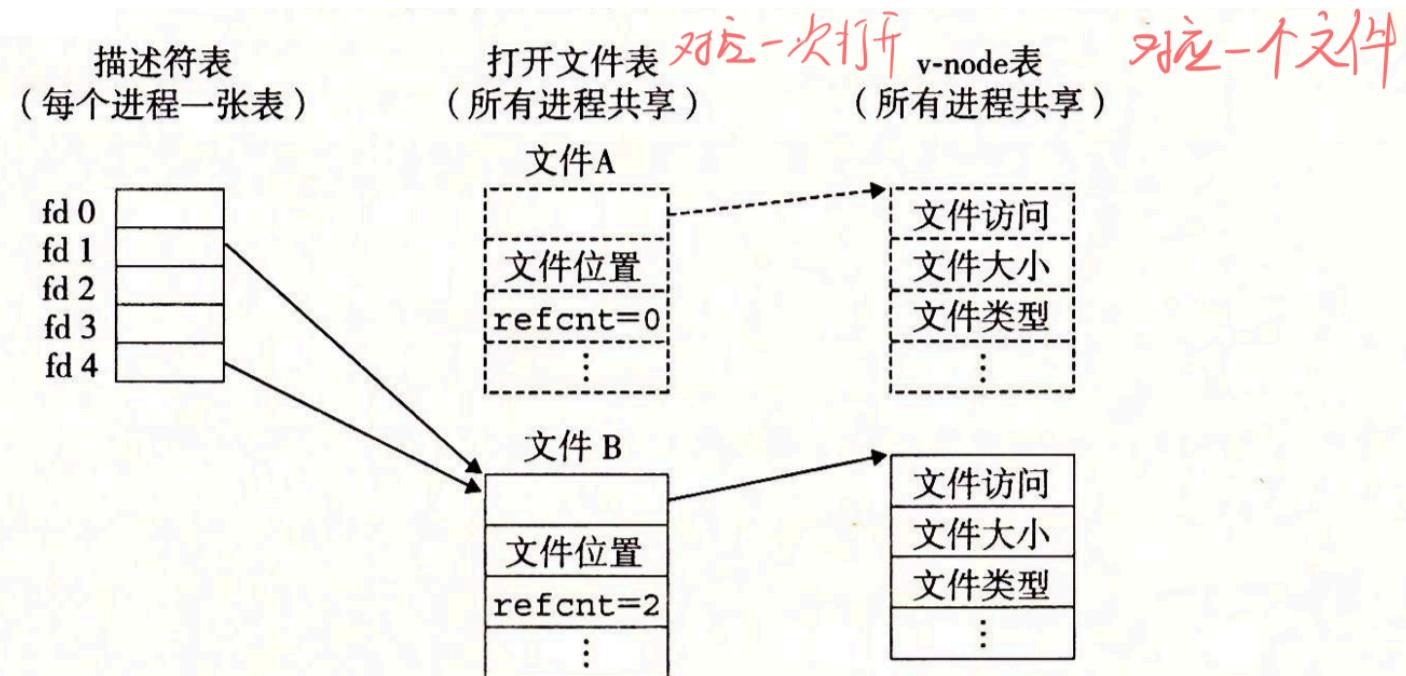


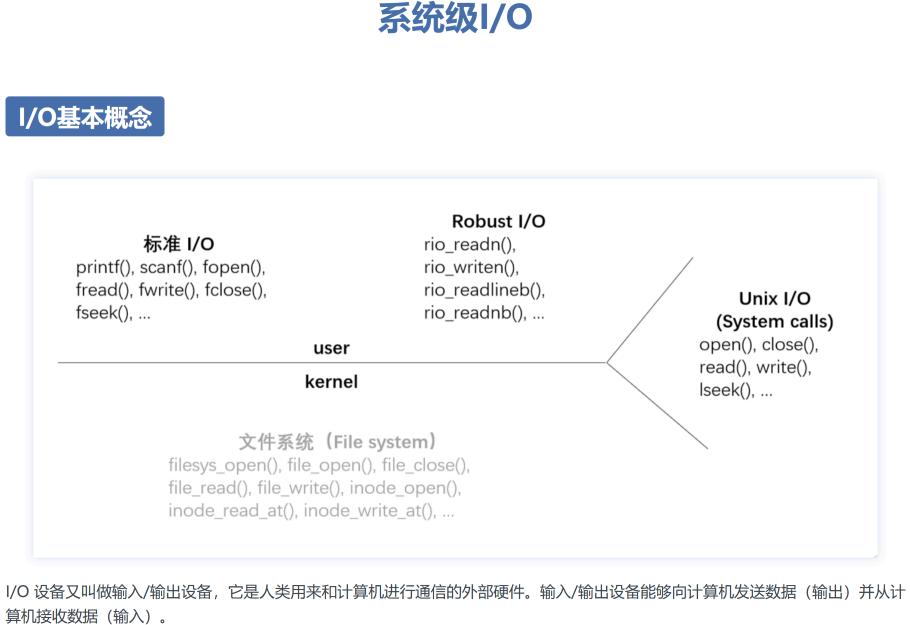
图 10-15 通过调用 `dup2(4, 1)` 重定向标准输出之后的内核数据结构。初始状态如图 10-12 所示

Emphasis

Notes

■ Note Link

- 系统级I/O
 - I/O基本概念
 - 文件操作
 - 打开文件
 - 关闭文件
 - 改变位置
 - 读写文件
 - 读取文件元数据
 - 读取目录内容
 - 文件类型
 - 共享文件
 - 打开文件
 - 文件共享
 - 父子文件共享
 - I/O重定向
 - I/O包
 - Unix I/O
 - RIO
 - 无缓冲I/O
 - 带缓冲I/O
 - 标准I/O
 - 综合使用
 - 基本原则
 - 缓冲
 - 缓冲区注意
 - 缓冲与缓存
 - 缓冲的问题
 - 标准I/O的限制



Homework Review

HW8

学号	HW-分数	HW-问题
2300012932	10	
2300012935	10	
2300012950	10	
2300012951	10	
2300013083	10	
2300013115	10	
2300013158	10	
2300013201	10	
2300013222	10	
2300013230	10	
2300013272	10	
2300017788	10	
2300094610	10	

Exercises

E1

13. 下列关于系统 I/O 的说法中，正确的是：

- A. Linux shell 创建的每个进程开始时都有三个打开的文件：标准输入（文件描述符为 0）、标准输出（文件描述符为 1）、标准错误（文件描述符为 2），这使得程序始终不能使用保留的描述符 0、1、2 读写其他文件。
- B. Unix I/O 的 read/write 函数是异步信号安全的，故可以在信号处理函数中使用。
- C. RIO 函数包的健壮性保证了对于同一个文件描述符，任意顺序调用 RIO 包中的任意函数不会造成问题。
- D. 使用 `int fd1 = open("ICS.txt", O_RDWR);` 打开 ICS.txt 文件后，再用 `int fd2 = open("ICS.txt", O_RDWR);` 再次打开文件，会使得 `fd1` 对应的打开文件表中的引用计数 `refcnt` 加一。

E1

13. 下列关于系统 I/O 的说法中，正确的是：

- A. Linux shell 创建的每个进程开始时都有三个打开的文件：标准输入（文件描述符为 0）、标准输出（文件描述符为 1）、标准错误（文件描述符为 2），这使得程序始终不能使用保留的描述符 0、1、2 读写其他文件。
- B. Unix I/O 的 read/write 函数是异步信号安全的，故可以在信号处理函数中使用。
- C. RIO 函数包的健壮性保证了对于同一个文件描述符，任意顺序调用 RIO 包中的任意函数不会造成问题。
- D. 使用 `int fd1 = open("ICS.txt", O_RDWR);` 打开 ICS.txt 文件后，再用 `int fd2 = open("ICS.txt", O_RDWR);` 再次打开文件，会使得 `fd1` 对应的打开文件表中的引用计数 `refcnt` 加一。

答案：B（简单） ↶

- A. 描述符 0, 1, 2 可以重定向到其他文件。 ↶
- B. 正确，教材 P534。 ↶
- C. 有缓冲区和无缓冲区的不可以交叉使用，教材 P629。 ↶
- D. 多次打开文件应该对应着多个打开文件表，教材 P635。 ↶

E2

14. 考虑以下代码，假设 ICS.txt 中的初始内容为"ICS!!!ics!!!"。

```
int fd = open("ICS.txt", O_RDWR | O_CREAT | O_TRUNC,  
S_IRUSR | S_IWUSR);  
for (int i = 0; i < 2; ++i){  
    int fd1 = open("ICS.txt", O_RDWR | O_APPEND);  
    int fd2 = open("ICS.txt", O_RDWR);  
    write(fd2, "!!!!!!", 6);  
    write(fd1, "ICS", 3);  
    write(fd, "ics", 3);  
}
```

假设所有系统调用均成功，则这段代码执行结束后，ICS.txt 的内容为：

- A. ICSics
- B. !!!icsICS
- C. !!!icsics!!!ICSIICS
- D. !!!icsICSIICS

E2

14. 考虑以下代码，假设 ICS.txt 中的初始内容为"ICS!!!ics!!!"。

```
int fd = open("ICS.txt", O_RDWR | O_CREAT | O_TRUNC,
S_IRUSR | S_IWUSR);
for (int i = 0; i < 2; ++i){
    int fd1 = open("ICS.txt", O_RDWR | O_APPEND);
    int fd2 = open("ICS.txt", O_RDWR);
    write(fd2, "!!!!!!", 6);
    write(fd1, "ICS", 3);
    write(fd, "ics", 3);
}
```

假设所有系统调用均成功，则这段代码执行结束后，ICS.txt 的内容为：

- A. ICSics
- B. !!!icsICS
- C. !!!icsics!!!ICSIICS
- D. !!!icsICSIICS

答案：D（中等）

E3

6. 假设某进程有且仅有五个已打开的文件描述符: $0 \sim 4$, 分别引用了五个不同的文件, 尝试运行以下代码:

```
dup2(3, 2); dup2(0, 3); dup2(1, 10); dup2(10, 4); dup2(4, 0);
```

关于得到的结果, 说法正确的是:

- A. 运行正常完成, 现在有四个描述符引用同一个文件
- B. 运行正常完成, 现在进程共引用四个不同的文件
- C. 由于试图从一个未打开的描述符进行复制, 发生错误
- D. 由于试图向一个未打开的描述符进行复制, 发生错误

E3

6. 假设某进程有且仅有五个已打开的文件描述符: 0~4, 分别引用了五个不同的文件, 尝试运行以下代码:

```
dup2(3, 2); dup2(0, 3); dup2(1, 10); dup2(10, 4); dup2(4, 0);
```

关于得到的结果, 说法正确的是:

- A. 运行正常完成, 现在有四个描述符引用同一个文件
- B. 运行正常完成, 现在进程共引用四个不同的文件
- C. 由于试图从一个未打开的描述符进行复制, 发生错误
- D. 由于试图向一个未打开的描述符进行复制, 发生错误

答案: A

说明: 一开始打开文件描述符 (0, 1, 2, 3, 4) 对应文件 (A, B, C, D, E), 结束后打开描述符 (0, 1, 2, 3, 4, 10) 对应 (B, B, D, A, B, B), A 正确, B 应该为引用三个不同文件。教材 637 页说明过可以向未打开的描述符进行复制, 因此 D 错误; 虽然教材未提及从未打开的描述符进行复制的后果, 但执行过程中并没有发生这种情况, 因此 C 错误。

E4

第四题 (10 分)

分析以下C程序，其中f1.txt和f2.txt为已有用户有读写的文件，初始文件内容为空。

```

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <sys/types.h>
4. #include <sys/stat.h>
5. #include <fcntl.h>
6. #include <unistd.h>
7. #include <sys/types.h>
8. #include <sys/wait.h>
9.
10. int main()
11. {
12.     int fd1,fd2,fd3,fd4;
13.     int pid;
14.     int c=1;
15.     fd1=open("./f1.txt",O_WRONLY,0);
16.     fd2=open("./f1.txt",O_WRONLY,0);
17.
18.     printf("fd1=%d,fd2=%d:\n",fd1,fd2);
19.
20.     write(fd1,"EECS PKU",7);
21.     write(fd2,"2019",4);
22.
23.     close(fd2);
24.
25.     fd3=open("./f2.txt",O_WRONLY,0);
26.     fd4=dup(fd3);
27.
28.     printf("fd3=%d,fd4=%d:\n",fd3,fd4);
29.
30.     pid=fork();
31.     if((pid==0)) {
32.         c--;
33.         write(fd3,"PKU",3);
34.         write(fd4,"ICS",3);
35.         printf("c=%d\n",c);
36.     }
37.     else {
38.         waitpid(-1,NULL,0);
39.         c++;
40.         write(fd3,"2019",4);

```

```

41.         close(fd3);
42.         close(fd4);
43.         printf("c= %d\n",c);
44.     }
45.
46.     if(c)
47.         write(fd1,"CS",2);
48.     c++;
49.     close(fd1);
50.     printf("c=%d\n",c);
51. }
```

当程序正确运行后，填写输出结果：

- (1) 程序第 18 行: fd1= ①, fd2= ②;
- (2) 程序第 28 行: fd3= ①, fd4= ②;
- (3) 程序第 35、43、50 行输出 c 的值依次分别为: _____;
- (4) 文件 f1.txt 中的内容为: _____;
- (5) 文件 f2.txt 中的内容为: _____。

E4

解答题三

1. 3, 4。注意 0, 1, 2 是标准 I/O 流占用的。
2. 4, 5。因为 fd2 已经被关闭，其描述符可复用；另外 fd3, fd4 虽然指针相同，但是描述符的号码是不同的。
3. 0, 1, 2, 3。父进程等待子进程结束，所以子进程先输出，另外父子进程的 count 当然是独立的。
4. 2019PKUCS。对 f1.txt 的写有三处，其中 20、21 行是独立的，得到 2019PKU，最后只有子进程再次写文件，此时 fd1 的指针在最后，即 2019PKUCS。
5. PKUICS2019。鉴于 dup 产生 fd4，而父子进程的文件描述符指向同样的打开文件表条目，所以对 f2.txt 的写本质上只有一个指针，因为父进程等待子进程结束，所以先输出 PKUICS 再输出 2019。

Notices

THANKS

Made by WalkerCH

changxinhai@stu.pku.edu.cn

Reference: [Weicheng Lin]'s presentation.

Reference: [Arthals]'s templates and content.

