Computer Systems

第十次小班课

助教：罗兆丰

信号

- 操作系统层面的异常，由内核发送
- 通知一个进程发生了某种类型的事
- 内容只有一个信号类型码

序号	名称	默认行为	相应事件
1	SIGHUP	终止	终端线挂断
2	SIGINT	终止	来自键盘的中断
3	SIGQUIT	终止	来自键盘的退出
4	SIGILL	终止	非法指令
5	SIGTRAP	终止并转储内存①	跟踪陷阱
6	SIGABRT	终止并转储内存①	来自 abort 函数的终止信号
7	SIGBUS	终止	总线错误
8	SIGFPE	终止并转储内存①	浮点异常
9	SIGKILL	终止②	杀死程序
10	SIGUSR1	终止	用户定义的信号 1
11	SIGSEGV	终止并转储内存①	无效的内存引用（段故障）
12	SIGUSR2	终止	用户定义的信号 2
13	SIGPIPE	终止	向一个没有读用户的管道做写操作
14	SIGALRM	终止	来自 alarm 函数的定时器信号
15	SIGTERM	终止	软件终止信号
16	SIGSTKFLT	终止	协处理器上的栈故障
17	SIGCHLD	忽略	一个子进程停止或者终止
18	SIGCONT	忽略	继续进程如果该进程停止
19	SIGSTOP	停止直到下一个 SIGCONT②	不是来自终端的停止信号
20	SIGTSTP	停止直到下一个 SIGCONT	来自终端的停止信号
21	SIGTTIN	停止直到下一个 SIGCONT	后台进程从终端读
22	SIGTTOU	停止直到下一个 SIGCONT	后台进程向终端写
23	SIGURG	忽略	套接字上的紧急情况
24	SIGXCPU	终止	CPU 时间限制超出
25	SIGXFSZ	终止	文件大小限制超出
26	SIGVTALRM	终止	虚拟定时器期满
27	SIGPROF	终止	剖析定时器期满
28	SIGWINCH	忽略	窗口大小变化
29	SIGIO	终止	在某个描述符上可执行 I/O 操作
30	SIGPWR	终止	电源故障

信号的发送和接收

- 发送信号：内核更改进程上下文的状态

可以内核检测到事件主动发送，也可以由进程请求内核发送(kill)

- 接收信号：进程对信号进行反应

- 待处理信号：已发送但未被接收，同类信号不排队
- 阻塞信号：可以发送信号，但不会被接受

发送信号的方法

- 父进程和子进程默认属于同一个进程组
- 提供正的pid则针对进程，负的pid则针对进程组
- 程序外：/bin/kill，Ctrl+C/Z（前台进程组）
- 程序内：kill，alarm

接收信号

- 从内核模式切换到用户模式时检查（上下文切换，系统调用返回）
- 检查未阻塞的待处理信号，非空则接收信号（通常为标号最小的）

- 信号的4种默认行为
- signal函数更改信号的行为
- 信号处理程序

- 信号处理程序可以互相打断

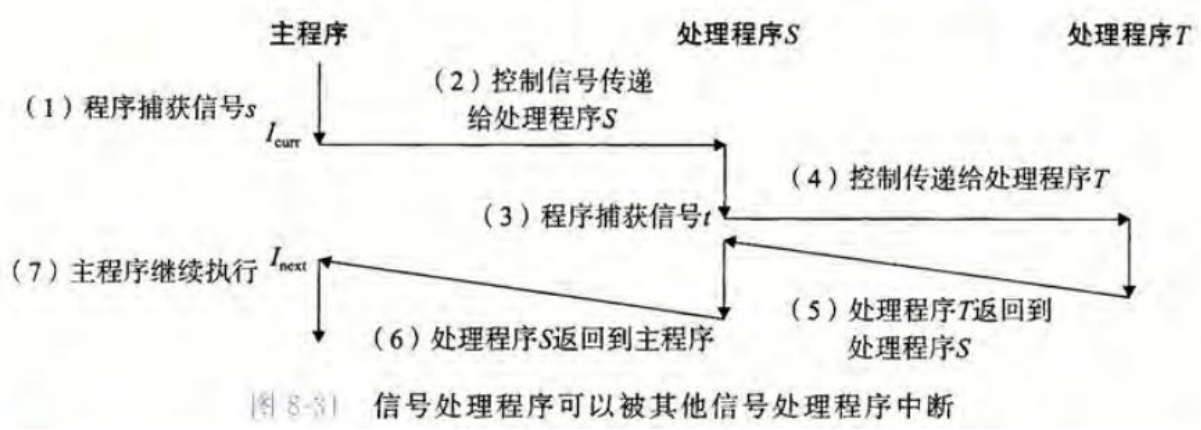


图 8-31 信号处理程序可以被其他信号处理程序中断

阻塞信号

- 内核默认阻塞和当前正在处理的信号同类的信号

- 使用sigprocmask主动阻塞/解除阻塞信号

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

返回：如果成功则为0，若出错则为−1。

```
int sigismember(const sigset_t *set, int signum);
```

返回：若signum是set的成员则为1，如果不是则为0，若出错则为−1。

信号处理程序注意事项

- 和主程序以及其他信号处理程序并发执行
- 并发访问全局数据结构可能导致问题

- 原则：
- 1.处理程序尽可能简单
- 2.只调用异步信号安全函数
- 3.保存恢复errno
- 4.访问全局数据结构时阻塞所有信号
- 5.全局变量使用volatile声明
- 6.标志使用sig_atomic_t声明

异步信号安全函数

- 可重入：不使用共享数据
- 或无法被打断

- Linux保证安全的函数列表
- _exit, write, wait, waitpid, sleep, kill

- printf, sprintf, malloc, exit不为异步信号安全
- 输出只能使用write

printf在信号处理程序中死锁

- 加锁：如果没有别人加过锁就继续，否则等待直到解锁

信号处理注意事项

- 同种信号不排队
- 不能通过信号判断事件发生的次数

并发流同步

- 并发流执行顺序任意
- 竞争(race)：运行结果和具体执行顺序有关

```
while (1) {
    if ((pid = Fork()) == 0) { /* Child process */
        Execve("/bin/date", argv, NULL);
    }
    Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent process */
    addjob(pid);  /* Add the child to the job list */
    Sigprocmask(SIG_SETMASK, &prev_all, NULL);
}
```

```
while (1) {
    Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
    if ((pid = Fork()) == 0) { /* Child process */
        Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
        Execve("/bin/date", argv, NULL);
    }
    Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
    addjob(pid);  /* Add the child to the job list */
    Sigprocmask(SIG_SETMASK, &prev_one, NULL);  /* Unblock SIGCHLD */
}
```

等待信号

- 错误方法：

```
while (!pid)
    pause();
```

- 正确但不好的方法：

```
while (!pid)
    ;
```

```
while (!pid)
    sleep(1);
```

- 正确方法：sigsuspend

sigsuspend函数等价于下述代码的原子的(不可中断的)版本：

```
1       sigprocmask(SIG_SETMASK, &mask, &prev);
2       pause();
3       sigprocmask(SIG_SETMASK, &prev, NULL);
```

非本地跳转

- setjmp：保存当前调用环境（PC，栈指针，通用目的寄存器）

```
int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);
```

- longjmp：恢复调用环境，返回到初始化env的setjmp，返回值retval

```
void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);
```

- 带sig的可在信号处理程序中使用

练习

- 判断下列说法正确性

1. SIGTSTP信号既不能被捕获，也不能被忽略
2. 存在信号的默认处理行为是进程停止直到被SIGCONT信号重启
3. 系统调用不能被中断，因为那是操作系统的工作
4. 在任何时刻，一种类型至多只会有一个待处理信号
5. 信号既可以发送给一个进程，也可以发送给一个进程组
6. SIGTERM和SIGKILL信号既不能被捕获，也不能被忽略
7. 当进程在前台运行时键入Ctrl-C，内核就会发一个SIGINT信号给这个前台进程
8. 子进程能给父进程发送信号，但不能发送给兄弟进程

1. （错）
2. （对）
3. （错）
4. （对）
5. （对）
6. （错）
7. （对）
8. （错）

练习

- 下面关于非局部跳转的描述，正确的是

A. setjmp可以和siglongjmp使用同一个jmp_buf变量

B. setjmp必须放在main()函数中调用

C. 虽然 longjmp 通常不会出错，但仍然需要对其返回值进行出错判断

D. 在同一个函数中既可以出现setjmp，也可以出现longjmp

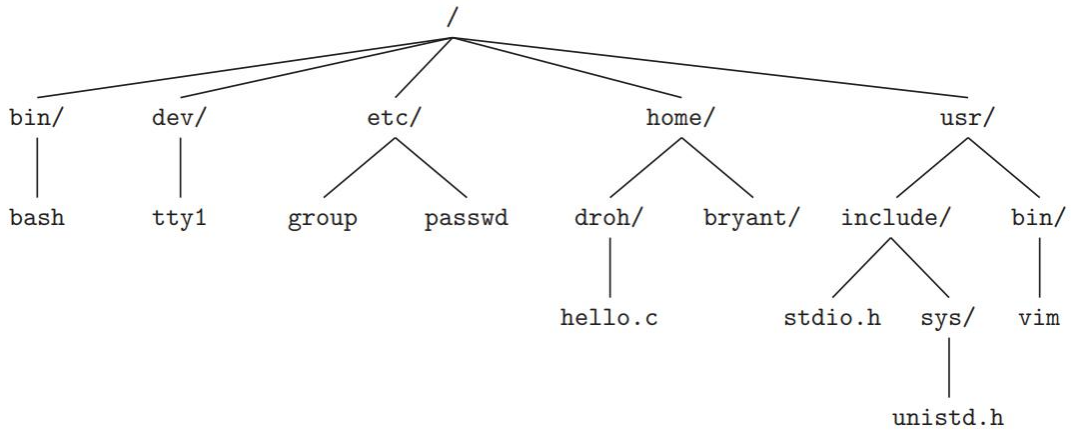
D

系统级I/O

文件

- 分类：
  - 普通文件：包含任意数据
    - 文本文件：只含ASCII或Unicode字符
    - 二进制文件：其他所有文件
  - 目录：包含将文件名映射到文件（普通文件/目录）的链接
  - 套接字等

- 目录层级结构
  - 绝对路径：从根目录开始的路径
  - 例：/home/droh/hello.c
  - 相对路径：从当前工作目录开始的路径
  - 例：./hello.c　../droh/hello.c



打开和关闭文件

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

- flags: 指明进程如何访问权限

-
  - O_RDONLY：只读。
  - O_WRONLY：只写。
  - O_RDWR：可读可写。
  - O_CREAT：如果文件不存在，就创建它的一个截断的(truncated)(空)文件。
  - O_TRUNC：如果文件已经存在，就截断它。
  - O_APPEND：在每次写操作前，设置文件位置到文件的结尾处。

指定权限位为mode & ~umask

掩码	描述
S_IRUSR	使用者（拥有者）能够读这个文件
S_IWUSR	使用者（拥有者）能够写这个文件
S_IXUSR	使用者（拥有者）能够执行这个文件
S_IRGRP	拥有者所在组的成员能够读这个文件
S_IWGRP	拥有者所在组的成员能够写这个文件
S_IXGRP	拥有者所在组的成员能够执行这个文件
S_IROTH	其他人（任何人）能够读这个文件
S_IWOTH	其他人（任何人）能够写这个文件
S_IXOTH	其他人（任何人）能够执行这个文件

打开和关闭文件

- open例

```
#define DEF_MODE   S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
#define DEF_UMASK  S_IWGRP|S_IWOTH
umask(DEF_UMASK);
fd = Open("foo.txt", O_CREAT|O_TRUNC|O_WRONLY, DEF_MODE);
```

- 创 有读的权限。

读和写文件

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t n);
                                返回：若成功则为读的字节数，若 EOF 则为 0，若出错为－1。
ssize_t write(int fd, const void *buf, size_t n);
                                返回：若成功则为写的字节数，若出错则为－1。
```

- 读时遇到EOF
- 从终端读文本行
- 读和写网络套接字

读取文件元数据

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

int stat(const char *filename, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

返回：若成功则为0，若出错则为−1。

- 前者以文件名为输入，后者以文件描述符为输入
- 包含文件信息
  - st_size: 文件的字节数大小，st_mode: 文件访问许可位

S_ISREG(m)。这是一个普通文件吗？
S_ISDIR(m)。这是一个目录文件吗？
S_ISSOCK(m)。这是一个网络套接字吗？

```
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;  /* Block size 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 */
};
```

读取目录内容

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

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

- opendir若成功，则返回处理的指针，失败则为NULL
- readdir以opendir的指针为参数，返回指向流dirp中下一个目录项的指针，若没有更多项，则返回NULL.
- 一个dirent结构体中，存储了该目录的位置和该目录的名字
- closedir关闭流并释放所有资源

RIO包

- 无缓冲的RIO

```
#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);
```

返回：若成功则为传送的字节数，若EOF则为0(只对rio_readn而言)，若出错则为-1。

RIO包

- 带缓冲的RIO

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);
```

返回：无。

```
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：若成功则为读的字节数，若EOF则为0，若出错则为−1。

- 缓冲：减少多次调用系统级I/O带来的陷入内核的开销
- rio_readlineb和rio_readnb可以交替使用，但不能和rio_readn交替使用

标准I/O

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

针。

- 类型为FILE的流是对文件描述符和缓冲区的抽象。
- 尽可能减小Unix I/O系统调用的开销

合理使用I/O函数

- 哪个I/O?
  - G1: 只要有可能就是用标准I/O
  - G2: 不要使用scanf或rio_readlineb来读二进制文件（专用于读取文本文件）
  - G3: 对网络套接字的I/O采用RIO函数

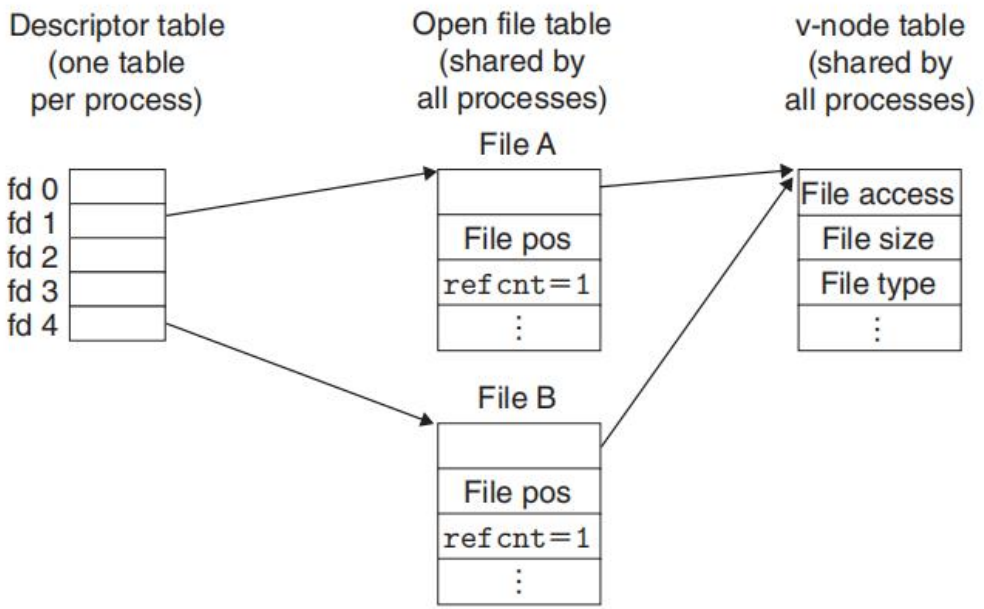
- 注意一些对流的限制和对套接字的限制，有时候会相互冲突：
  - 在输出函数后有输入函数：需要在二者之间插入fflush，fseek，fsetpos或rewind
  - 在输入函数后有输出函数：需要在二者之间插入fseek，fsetpos，rewind（除非输入函数遇到了EOF）

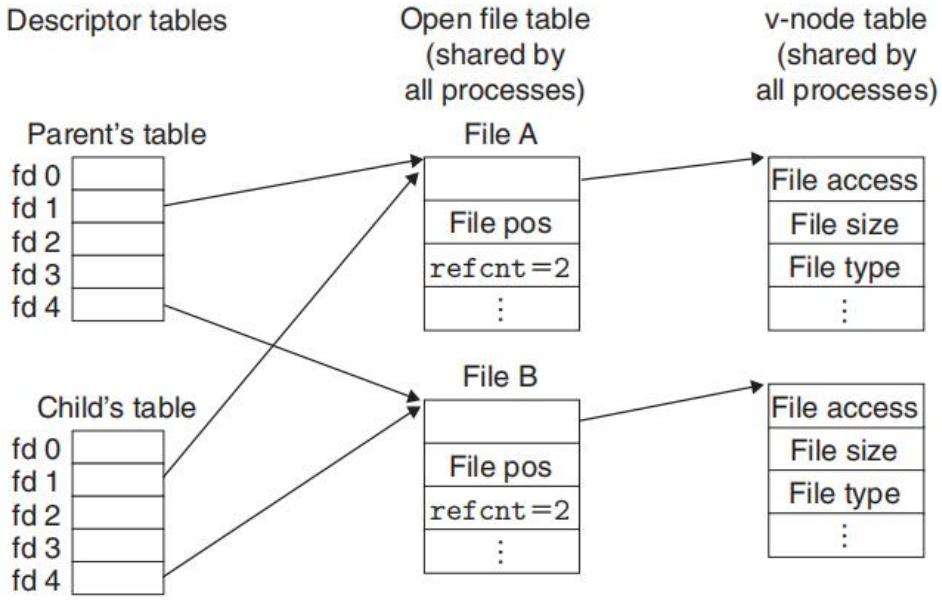
共享文件

- 描述符表：每个进程都有他独立的描述符表，其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。
- 文件表：打开文件的集合，所有的进程共享这一张表，每个文件表的组成包括当前的文件位置、引用计数（当前指向该表项的描述符表项数）、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的引用计数，当为0则从文件表中删除
- v-node表：包含了stat结构中的大多数信息，被所有进程共享

共享文件

- 对同一个filename open两次
- fork 之后，子进程得到父进程描述符表的副本，此时父子进程共享打开文件表





I/O重定向

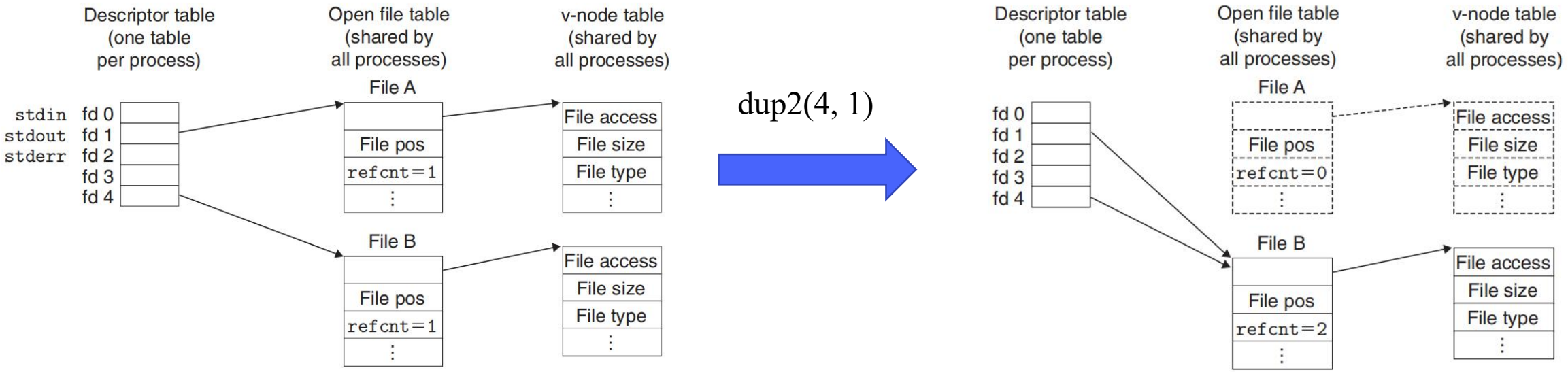
```
#include <unistd.h>

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

- 若成功，返回新的文件描述符；若出错，返回-1
- dup: 创建新的文件描述符，复制oldfd的描述符表项给它
- dup2: 复制oldfd的描述符表项给newfd，如果newfd已创建，复制前关闭原有的

I/O重定向

- dup2例


Descriptor table
(one table
per process)
Open file table
(shared by
all processes)
v-node table
(shared by
all processes)
stdin
stdout
stderr
fd 0
fd 1
fd 2
fd 3
fd 4
File A
File pos
refcnt=1
File access
File size
File type
File B
File pos
refcnt=1
File access
File size
File type
dup2(4, 1)
Descriptor table
(one table
per process)
Open file table
(shared by
all processes)
v-node table
(shared by
all processes)
fd 0
fd 1
fd 2
fd 3
fd 4
File A
File pos
refcnt=0
File access
File size
File type
File B
File pos
refcnt=2
File access
File size
File type

练习

- Do not mix buffered and not buffered read / write with each other!

- Which bytes from the file are read into y?

  A. Bytes 0 to 9

  B. Bytes 10 to 19

  C. None of these?

```
char x[20];
char y[20];
FILE* f = fopen("foo.txt", "rb");
int fd = fileno(f);
fread(x, 10, 1, f);
// read 10 bytes from f
read(fd, y, 10);
```

C

练习

1．假设缓冲区足够大，且 stdout 只有在关闭文件、换行与 fflush 的情况下才会刷新缓冲区。程序运行过程中的所有系统调用均成功。

(1)

```
int main() {
    printf("a");
    fork();
    printf("b");
    fork();
    printf("c");
    return 0;
}
```

(2)

```
int main() {
    write(1, "a", 1);
    fork();
    write(1, "b", 1);
    fork();
    write(1, "c", 1);
    return 0;
}
```

(3)

```
int main() {
    printf("a");
    fork();
    write(1, "b", 1);
    fork();
    write(1, "c", 1);
    return 0;
}
```

对于(1)号程序，写出它的一个可能的输出：abcabcabcabc。这个可能的输出是唯一的吗？是。

abc

abc

abc

abc

abcabcabcabc

练习

1．假设缓冲区足够大，且 stdout 只有在关闭文件、换行与 fflush 的情况下才会刷新缓冲区。程序运行过程中的所有系统调用均成功。

(1)

```
int main() {
    printf("a");
    fork();
    printf("b");
    fork();
    printf("c");
    return 0;
}
```

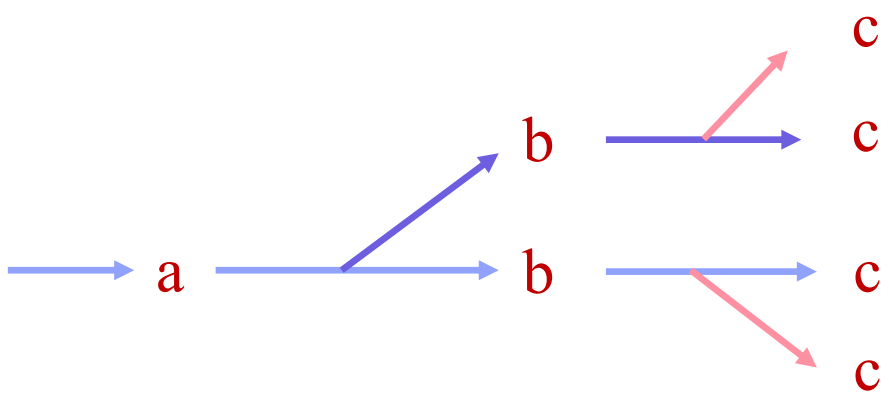
(2)

```
int main() {
    write(1, "a", 1);
    fork();
    write(1, "b", 1);
    fork();
    write(1, "c", 1);
    return 0;
}
```

(3)

```
int main() {
    printf("a");
    fork();
    write(1, "b", 1);
    fork();
    write(1, "c", 1);
    return 0;
}
```

对于(2)号程序，它的输出中包含 1 个 a， 2 个 b， 4 个 c。输出的第一个字符一定是 a 。



练习

1．假设缓冲区足够大，且 stdout 只有在关闭文件、换行与 fflush 的情况下才会刷新缓冲区。程序运行过程中的所有系统调用均成功。

(1)

```
int main() {
    printf("a");
    fork();
    printf("b");
    fork();
    printf("c");
    return 0;
}
```

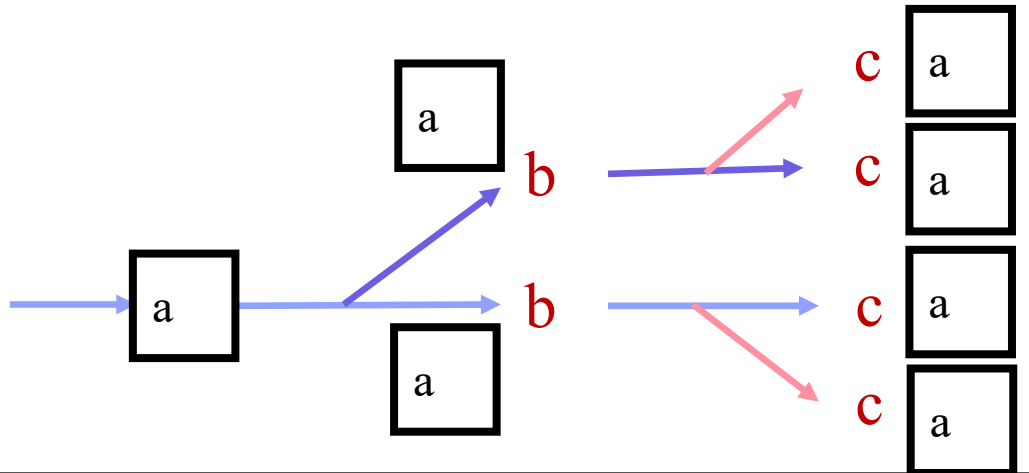
(2)

```
int main() {
    write(1, "a", 1);
    fork();
    write(1, "b", 1);
    fork();
    write(1, "c", 1);
    return 0;
}
```

(3)

```
int main() {
    printf("a");
    fork();
    write(1, "b", 1);
    fork();
    write(1, "c", 1);
    return 0;
}
```

对于(3)号程序，它的输出中包含 4 个 a， 2 个 b， 4 个 c。输出的第一个字符一定是 b 。

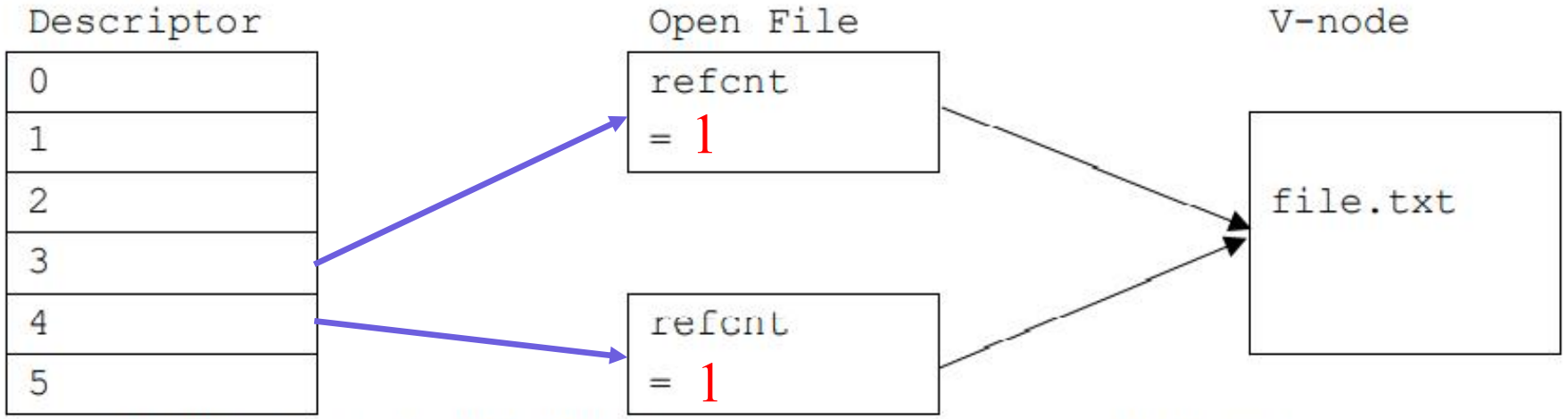


练习

2. 假设磁盘上有空文件 file.txt。程序运行过程中的所有系统调用均成功。

```
int main() {
    int fd1 = open("file.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    int fd2 = open("file.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    printf("%d %d\n", fd1, fd2);
    write(fd1, "123", 3);
    write(fd2, "45", 2);
    close(fd1); close(fd2);
    return 0;
}
```

(1) 程序关闭 fd1 前，画出 LINUX 三级表结构。填写 Open File 表中的 refcnt。



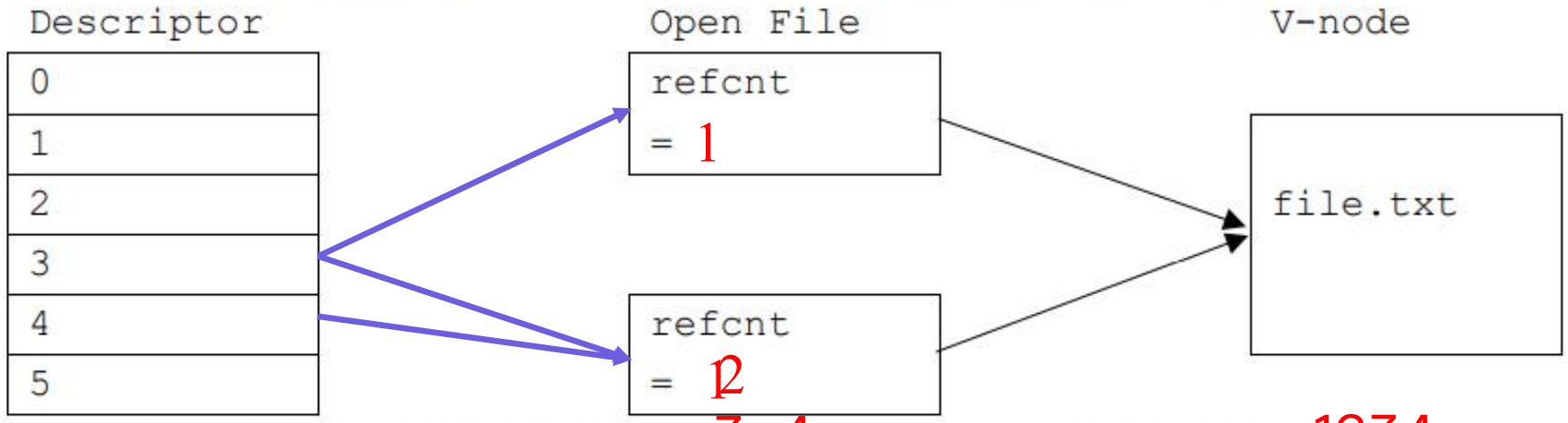
(2) 程序结束时，标准输出上的内容是 3 4 ，file.txt 中的内容是 453 。

练习

3. 假设磁盘上有空文件 file.txt。程序运行过程中的所有系统调用均成功。

```
int main() {
    int fd1 = open("file.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    int fd2 = open("file.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    dup2(fd2, fd1);
    printf("%d %d\n", fd1, fd2);
    write(fd1, "123", 3);
    write(fd2, "45", 2);
    close(fd1); close(fd2); return 0;
}
```

(1) 程序关闭 fd1 前，画出 LINUX 三级表结构。填写 Open File 表中的 refcnt。



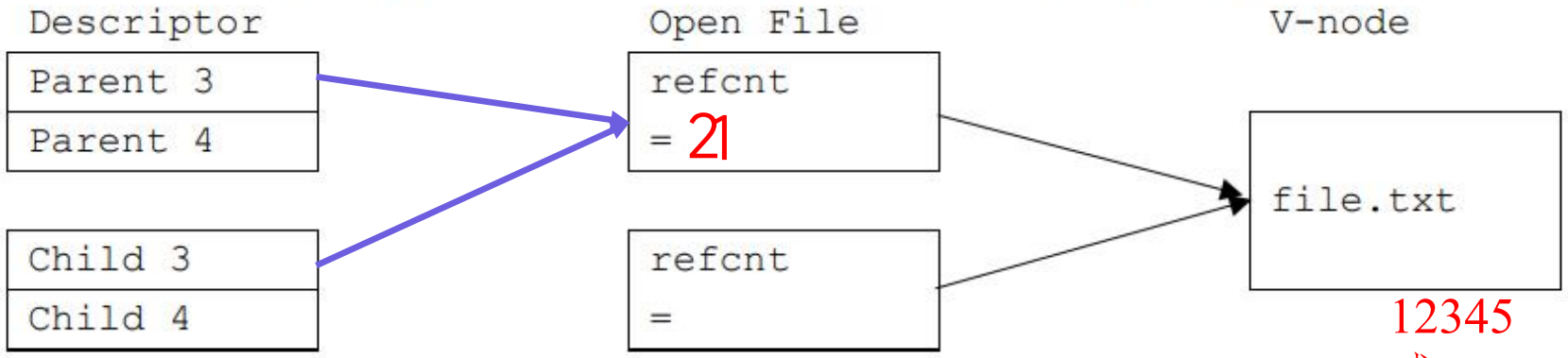
(2) 程序结束时，标准输出上的内容是 3 4 ，file.txt 中的内容是 1234 5 。

练习

4．假设磁盘上有空文件 file.txt。程序运行过程中的所有系统调用均成功。缓冲区足够大，且 stdout 只有在关闭文件、换行与 fflush 的情况下才会刷新缓冲区。

```
int main() {
    pid_t pid; int child_status;
    int fd1 = open("file.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    if ((pid = fork()) > 0) {                    // Parent
        printf("P:%d ", fd1);
        write(fd1, "123", 3);
        waitpid(pid, &child_status, 0);
    } else {                                     // Child
        printf("C:%d ", fd1);
        write(fd1, "45", 2);
    }
    close(fd1); return 0;
}
```

(1) 子进程关闭 fd1 前，画出 LINUX 三级表结构。填写 Open File 表中的 refcnt。



(2) 程序结束时，标准输出上的内容是 C:3 P:3 ，file.txt 中的内容是 12345 或45123

THANK YOU

Come On !