



第5章 进程管理



主要内容

- ➡ 5.1 进程基本概念
- ➡ 5.2 进程的创建和命令执行
- ➡ 5.3 进程的退出
 - ➡ 守护进程
 - ➡ 僵尸进程
 - ➡ 进程退出状态
- ➡ 5.4 进程开发实例
- ➡ 5.5 重定向和管道

进程基本概念

Linux进程的主要类型：

- **交互进程**：由Shell启动的进程。可在前台或者后台运行。前台可通过Shell与用户交互
- **批处理进程**：该类进程和终端没有联系，由多个进程按照指定的方式执行
- **守护进程**：在后台运行的与任何终端无关的进程。

如何编写shell ?

- shell是一个管理进程和运行程序的程序
 - (1)运行程序
 - (2)管理输入/输出
 - (3)可编程

shell如何运行程序的？

- ➡ (1)用户键入a.out
- ➡ (2)shell建立一个新进程来运行这个程序
- ➡ (3)shell将程序从磁盘载入
- ➡ (4)程序在它的进程中运行直到结束



shell的主循环

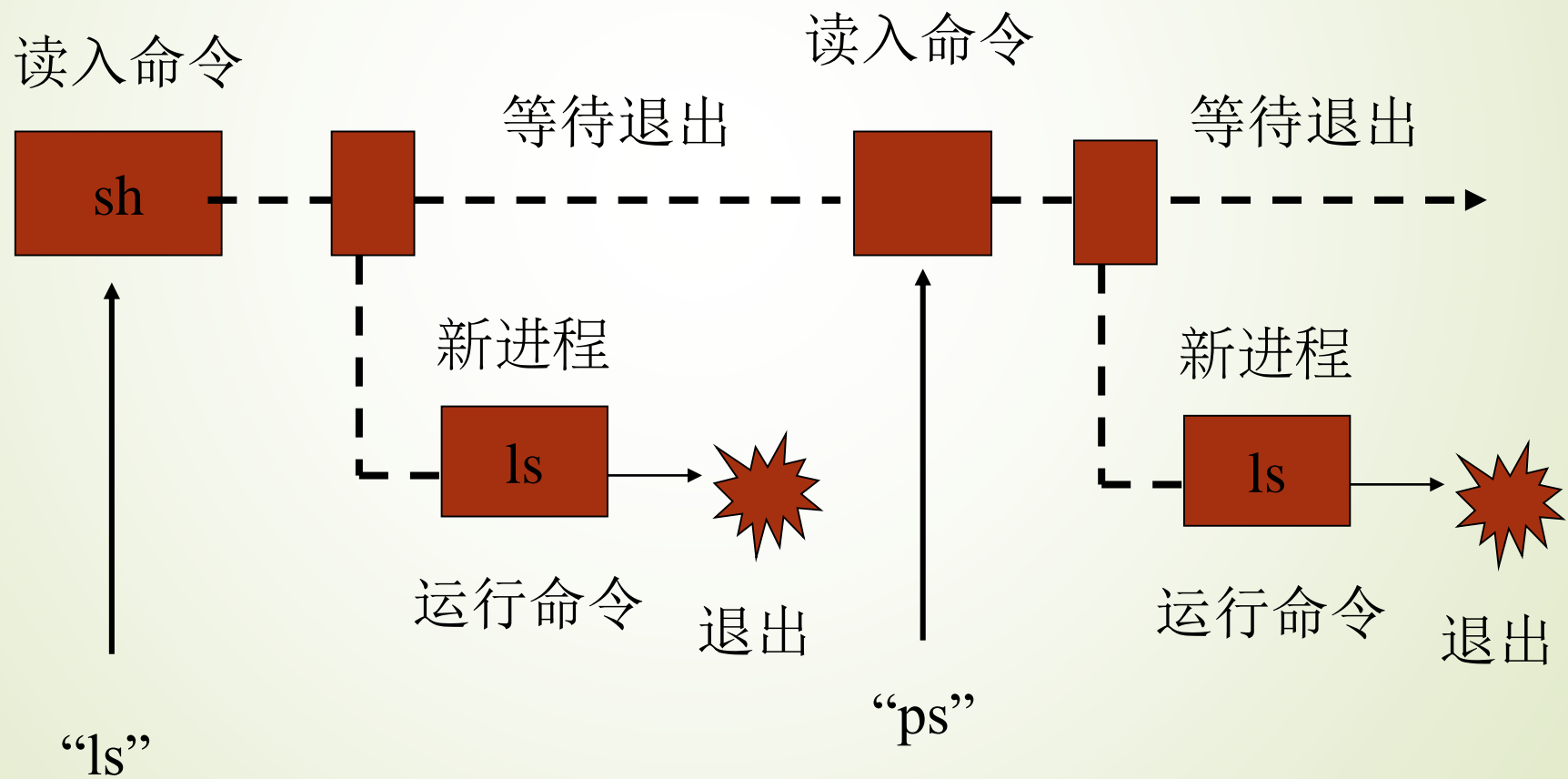
```
while (! end_of_input)
    get command
    execute command
    wait command to finish
```

例如:

ls

ps命令

时间



主要内容

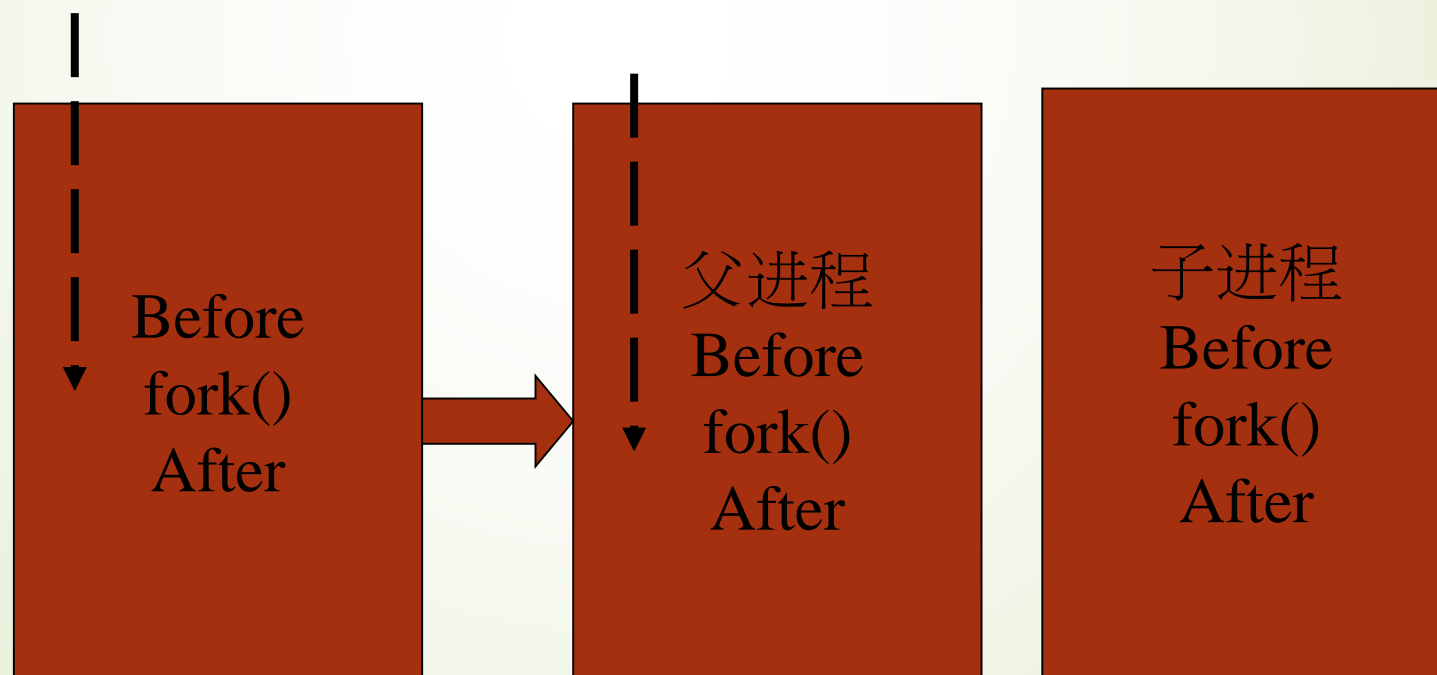
- ➡ 5.1 进程基本概念
- ➡ 5.2 进程的创建和命令执行
- ➡ 5.3 进程的退出
 - ➡ 守护进程
 - ➡ 僵尸进程
 - ➡ 进程退出状态
- ➡ 5.4 进程开发实例
- ➡ 5.5 重定向和管道

进程的创建 fork 系统调用

目标	创建进程
头文件	<code>#include <unistd.h></code>
函数原型	<code>pid_t result=fork(void);</code>
参数	无
返回值	-1 如果出错 0 返回到子进程 pid 将子进程的进程ID返回给父进程

如何建立新的进程？

➡ 进程调用fork复制自己




fork的执行过程

- 由内核执行如下任务：
 - (1) 分配新的内存块和内核数据结构
 - (2) 复制原来的进程到新的进程
 - (3) 向运行进程集添加新的进程
 - (4) 将控制返回给两个进程

forkdemo1.c例程

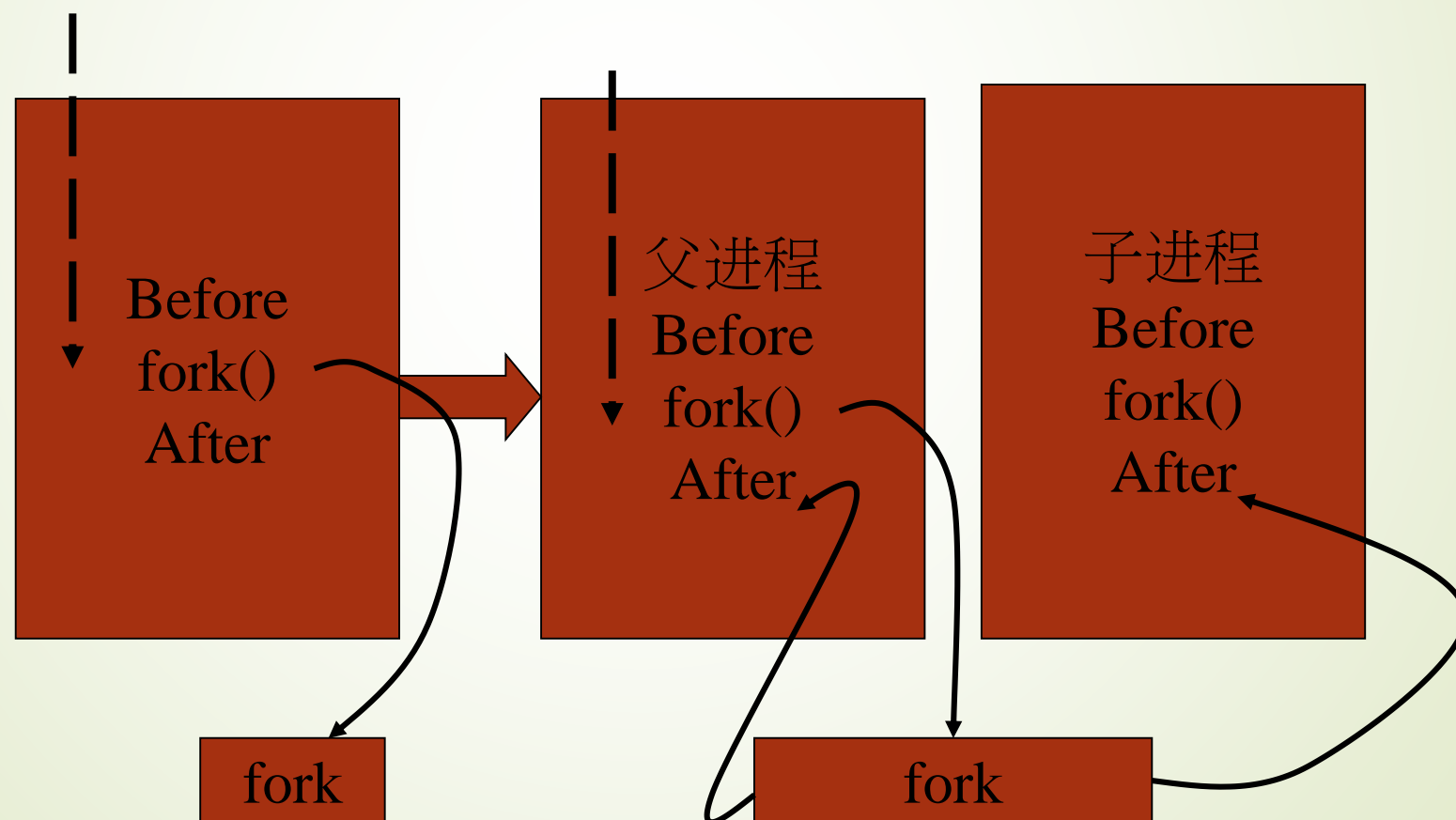
```
#include <stdio.h>
main()
{
    int ret_from_fork, mypid;
    mypid=getpid();
    printf( "Before : my pid is %d\n" ,mypid);
    ret_from_fork=fork();
    sleep(1);
    printf( "After:my pid is %d,fork() said
            %d\n" ,getpid(),ret_from_fork);
}
```



运行结果

- Before:my pid is 4170
- After:my pid is 4170, fork() said 4171
- After:my pid is 4171, fork() said 0

结果分析




fork功能的示意性说明

- 内核通过复制父进程4170来创建子进程4171，它将父进程的代码和当前运行的位置都复制给子进程
- 子进程从fork返回的地方开始运行，而不是从头开始运行
- 因此最终显示三条而不是四条打印信息
- fork调用一次，返回两次


fork例程2 子进程创建进程

```
//forkdemo2.c
main()
{
    printf( "my pid is %d\n" ,getpid());
    fork();
    fork();
    fork();
    printf( "my pid is %d\n" ,getpid());
}
```

程序运行结果：

- ➡ 该程序创建几个进程？
- ➡ 打印几条信息？



分辨父子进程


- 从forkdemo1可以看出，父进程调用fork创建子进程，此时父子进程同时运行到同一行而且有相同的数据和进程属性
- fork调用在父进程中返回子进程的pid
- fork在子进程中返回进程号为0
- 因此可根据fork的值判断父子进程

forkdemo3.c程序

```
main()
{
    int fork_rv;
    printf( "Before:my pid is %d\n" ,getpid());
    fork_rv=fork();
    if ( fork_rv== -1 ) exit(0);
    else if (fork_rv==0)
        printf( "I am the child,my pid is %d" ,getpid());
    else
        printf( "I am the parent,my child is %d" ,getpid());
}
```

父子进程数据共享

- 子进程被创建后，子进程拷贝了代码段和数据段,但它们之间相互独立，不受影响
- 文件描述符在父子进程之间共享
- 例5-2



```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int glob = 10;
int main(void)
{
    int local;
    pid_t pid;
    local = 8;
    if ((pid = fork()) == 0)
    {
        sleep(2);
    }
    else if (pid>0)
    {
        glob++;
        local--;
        sleep(10);
    }
    printf("glob = %d, local = %d mypid=%d\n", glob, local, getpid());
    exit (0);
}
```

execvp系统调用

目标	在指定路径中查找并执行一个文件
头文件	#include <unistd.h>
函数原型	result=execvp(const char *file,const char *argv[]);
参数	file 要执行的文件名 argv 字符串数组
返回值	-1 如果出错 成功，execvp没有返回值



主要内容

- ➡ 5.1 进程基本概念
- ➡ 5.2 进程的创建和命令执行
- ➡ 5.3 进程的退出
 - ➡ 守护进程
 - ➡ 僵尸进程
 - ➡ 进程退出状态
- ➡ 5.4 进程开发实例
- ➡ 5.5 重写向和管道



进程退出

- 退出时正常执行结束
- 执行exit函数
- 或者_exit

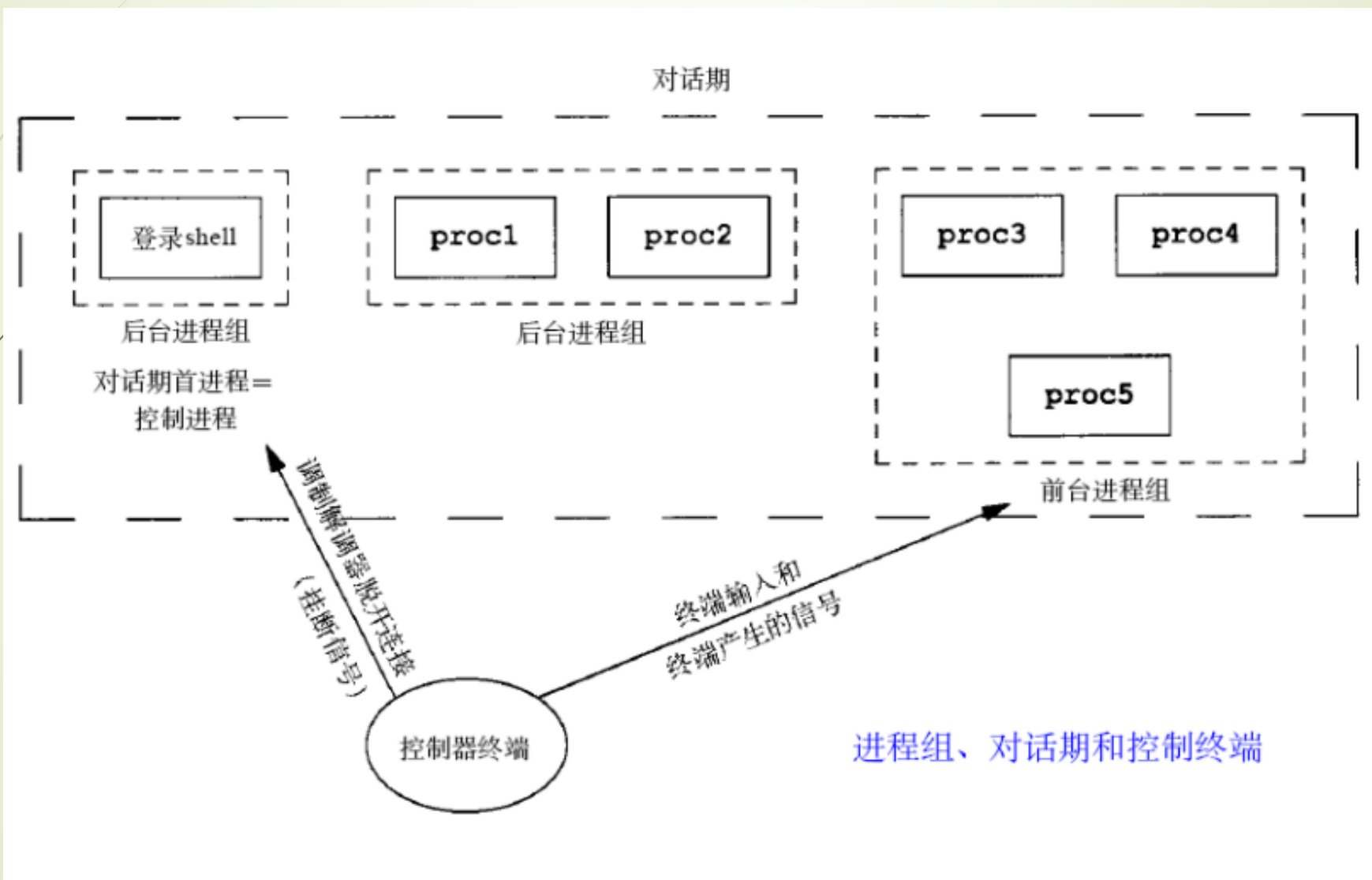
进程退出一守护进程

- 守护进程脱离终端控制
- 一般进程的执行都是在终端的提示符下输入命令得以执行
- 当该终端被关闭时，其所执行的进程也会被终止
- 守护进程正好相反

进程组与会话期

- **进程组**是一组进程的集合，由进程组PID表示
- 每个进程必须有一个进程组PID，即必须属于某个进程组
- 一个**终端的**控制进程是它所发起的一系列进程的进程组长
- **会话期**是由一个或者多个进程组组成的集合
- 开始于用户登录，结束于用户退出
- 在此期间，用户所运行的所有进程属于该会话期

进程组、会话期和控制终端关系




创建守护进程的过程

- ✧ 创建子进程，父进程退出
- ✧ 在子进程中创建新会话--setsid
- ✧ 改变当前目录为根目录
- ✧ 重设文件权限掩码(取消文件掩码)
- ✧ 关闭文件描述符(不再和标准输入/输出/错误输出进行交互，因此关闭)
- ✧ 例5-5.

```
#define MAXFILE 65535
int main(void)
{
    pid_t pc;
    int i, fd, len;
    char *buf = "Hello, everybody!\n";
    len = strlen(buf);
    pc = fork();
    if (pc < 0) {
        printf("fork error \n");
        exit(1);
    }
    else if (pc > 0)
        exit(0);
    setsid();
    chdir("/");
    umask(0);
    for (i = 0; i < MAXFILE; i++)
        while(1)
        {
            if ((fd = open("/tmp/daemon.log", O_CREAT | O_WRONLY | O_APPEND, 0600)) < 0)
            {
                perror("open");
                exit(1);
            }
            write(fd, buf, len+1);
            close(fd);
            sleep(10);
        }
} ? end main ?
```

僵尸进程

- 父进程还没有结束而子进程结束运行，同时父进程未调用wait系统调用等待子进程时，子进程将成为僵尸进程
- 它没有任何代码，数据或者堆栈，占用不了多少资源
- 但存在于系统的任务列表中，占据进程表的一个位置
- 例5-6



```
#include <stdio.h>
#include <unistd.h>
void parent_code(int delay) {
    sleep(delay);
}
main() {
    pid_t pid;
    int status;
    pid=fork();
    if (pid==0) ;
    if (pid>0)
        parent_code(100000);
}
```

进程的退出状态

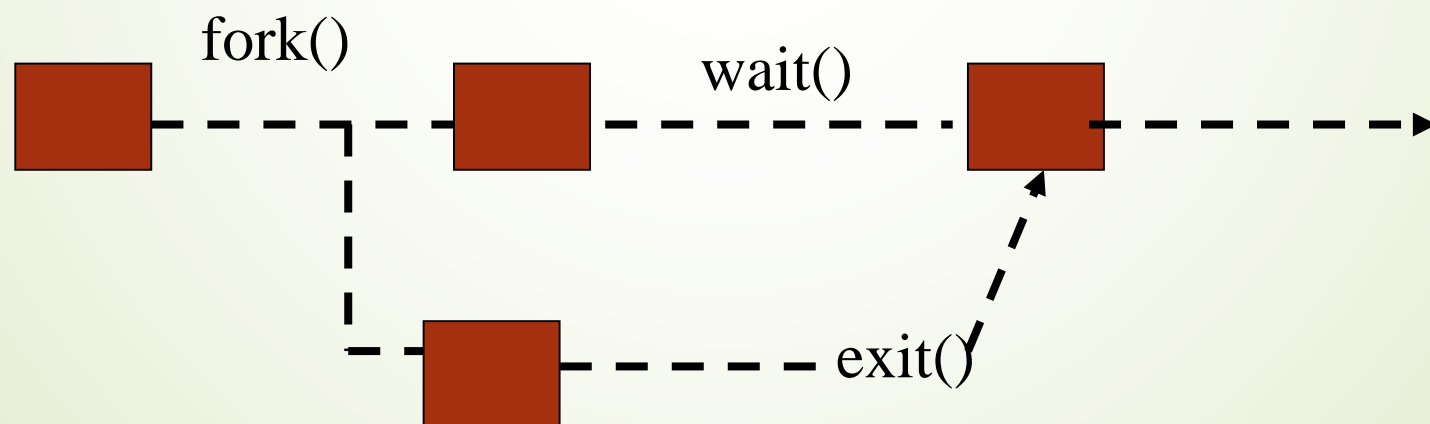
- 为了防止子进程变为僵尸进程，一般在父进程调用 `wait()` 系统调用等待子进程的结束并获取子进程的返回状态。
- 首先暂停调用它的进程直到子进程结束，然后取得子进程结束时传给 `exit` 的值

wait系统调用

目标	等待子进程的结束
头文件	<code>#include <sys/wait.h></code> <code>#include <sys/types.h></code>
函数原型	<code>pid_t pid=wait(int *status);</code>
参数	<code>status</code> 指向一个保存子进程返回状态的整形变量
返回值	如果不存在子进程，返回-1 若有任何一个子进程结束，则返回该子进程的pid并保存期返回状态在 <code>status</code> 中，同时， <code>wait</code> 调用也结束

父进程如何等待子进程的退出

- 进程调用wait等待子进程的退出
- `pid=wait(&status);`
- wait做两件事：首先暂停调用它的进程直到子进程结束，然后取得子进程结束时传给exit的值



waitdemo1.c程序


```
main(){  
    printf( "before :my pid is%d\n" ,getpid());  
    newpid=fork();  
    if (newpid==0)  
        child_code(Delay);  
    if (newpid>0)  
        parent_code(newpid);  
}
```

parent

```
main()
{
    fork();
}
child_code()
{
    ...
    exit(n);
}
parent_code(){
    int status;
    wait(&status);
}
```

child

```
main()
{
    fork();
}
child_code()
{
    ...
    exit(n);
}
parent_code(){
    int status;
    wait(&status);
}
```



wait的两个重要特征：

- ➡ (1)wait阻塞调用它的程序直到子进程结束
- ➡ (2)wait返回结束进程的PID

waitdemo2.c通信

- 进程退出有三种方式：
 - 成功、失败或死亡
- 成功调用`exit(0)`函数或者`main`函数中的`return 0`结束即为正常退出
- 进程执行失败，例如内存耗尽,通过`exit`传递一个非0值
- 进程可能被其它进程通过信号杀死
- 父进程如何知道子进程退出的原因呢？

wait(&status)

- 内核在进程退出时，将状态保存在status中，status为一整型变量
- 该整型变量由3部分组成：前8个位记录退出值，后7位记录信号序号，另一个bit用来指明发生错误并产生了core dump

exit value	core dump flag	signal number
------------	----------------	---------------

waitpid调用

- wait只能得到任何一个子进程结束的状态
- wait()调用属于阻塞调用，父进程执行该指令后，其等待子进程结束之后才能执行它后面的代码
- waitpid()可提供非阻塞调用的方式。
- waitpid()调用可以等待指定的子进程。
- 具有比wait多的功能

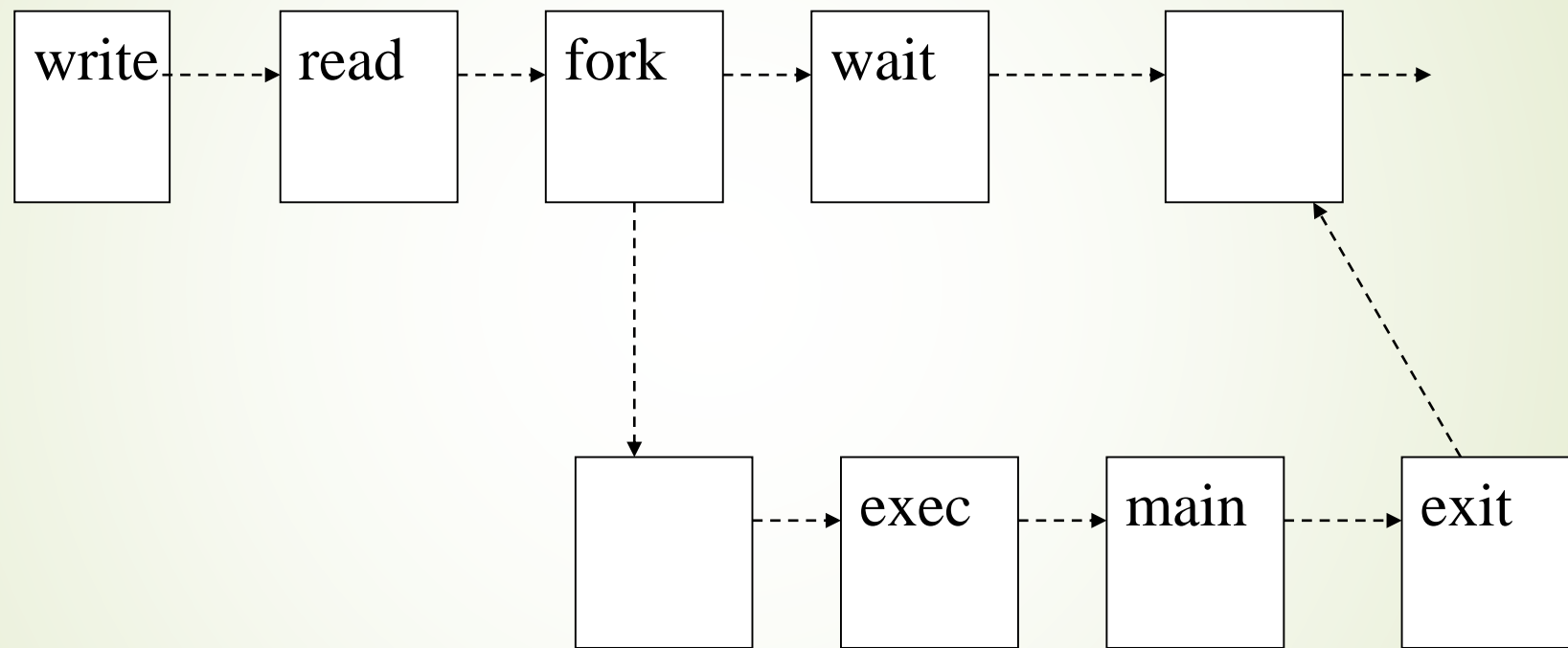
目标	等待某个子进程的结束
头文件	<pre>#include <sys/wait.h> #include <sys/types.h></pre>
函数原型	<pre>pid_t pid=waitpid(pid_t pid,int *status,int options);</pre>
参数	<p>pid=-1 等待任一个子进程。与wait等效。</p> <p>pid>0 等待其进程ID与pid相等的子进程。</p> <p>pid=0等待其组ID等于调用进程组ID的任一个子进程。</p> <p>pid<-1等待其组ID等于pid绝对值的任一子进程。</p> <p>Options选项:</p> <p>WNOHANG 表示如果没有任何已经结束的子进程则马上返回, 不予以等待。</p> <p>WUNTRACED 如果子进程进入暂停执行情况则马上返回,但结束状态不予以理会。</p> <p>0 作用和wait一样, 阻塞父进程, 等待子进程结束</p>
返回值	<p>如果有错误发生, 返回-1</p> <p>若有指定子进程结束, 则返回该子进程的pid并保存其返回状态在status变量中, 同时, waitpid调用结束。</p>




主要内容

- ➡ 5.1 进程基本概念
- ➡ 5.2 进程的创建和命令执行
- ➡ 5.3 进程的退出
 - ➡ 守护进程
 - ➡ 僵尸进程
 - ➡ 进程退出状态
- ➡ 5.4 进程开发实例
- ➡ 5.5 重定向和管道

shell如何运行程序？



例5-8



```
main()
{
    int pid ;
    int fd;
    printf("About to run who into a file\n");
    if( (pid = fork() ) == -1 ){
        perror("fork"); exit(1);
    }
    if ( pid == 0 ){
        close(1);                /* close, */
        fd = creat( "userlist", 0644 );    /* then open */
        execlp( "who", "who", NULL );    /* and run */
        perror("execlp");
        exit(1);
    }
    if ( pid != 0 ){
        wait(NULL);
        printf("Done running who. results in userlist\n");
    }
}
```

主要内容

- ➡ 5.1 进程基本概念
- ➡ 5.2 进程的创建和命令执行
- ➡ 5.3 进程的退出
 - ➡ 守护进程
 - ➡ 僵尸进程
 - ➡ 进程退出状态
- ➡ 5.4 进程开发实例
- ➡ 5.5 重定向和管道



I/O重定向和管道

- 标准输入/输出和标准错误的定义
- 重定向标准I/O到文件
- 使用fork为其他程序重定向
- 管道PIPE
- 创建管道后调用fork
- dup dup2 pipe系统调用

3个标准文件描述符

- 0：标准输入
- 1：标准输出
- 2：标准错误
- Unix中这三个文件描述符默认情况下为每个进程都打开，进程可以直接对其读写操作



tty


- 在shell中运行程序时，程序的stdin、stdout和stderr连接在终端上
- 程序将结果写到文件描述符1，错误消息写到文件描述符2
- 若需要写入文件中，需要进行重定向

重定向I/O的是shell而不是程序

- 通过重定向符号>告诉shell将文件描述符定位到文件
- 程序不断将数据写到文件描述符1中，不会意识到数据的目的地已经改变

listargs.c程序

```
main(int ac,char *av[])
{
    int i;
    printf("Number of args:%d,Args are:\n",ac);
    for(i=0;i<ac;i++)
        printf("args[%d] %s\n",i,av[i]);
    fprintf(stderr,"This message is sent to stderr.\n");
}
```



执行程序

```
./listargs testing one two
```

```
args[0] ./listargs
```

```
args[1] testing
```

```
args[2] one
```

```
args[3] two
```

```
This message is sent to stderr
```



`./listargs testing one two >xyz`

This message is sent to stderr

`cat xyz`

`args[0] ./listargs`

`args[1] testing`

`args[2] one`

`args[3] two`



`./listargs testing >xyz one two 2>oops`

`cat xyz`

`args[0] ./listargs`

`args[1] testing`

`args[2] one`

`args[3] two`

`cat oops`

This message is sent to stderr

结果分析

- shell并不将重定向符号及文件名传递给程序
- 重定向可以出现在命令行中的任何地方，重定向符号并不能终止命令和参数
- shell能够重定向其他文件描述符，例如2>filename

最低可用文件描述符

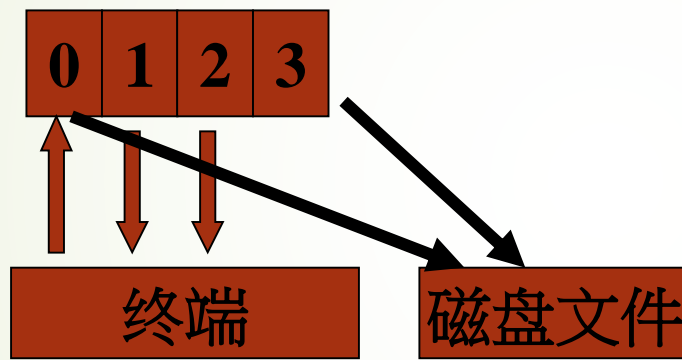
- 文件描述符就是一个数组的索引号
- 每个进程都有其打开的一组文件，它们保存在一个数组中
- 文件描述符即为某文件在此数组中的索引
- 当打开文件时，为此文件安排的描述符总是此数组中最低可用位置的索引

如何将stdin定向到文件

- 方法一： close 然后 open
 - 初始时，0、1、2分别连接到终端上
 - 使用close(0)关闭标准输入
 - 此时，文件描述符数组中第一个元素处于空闲状态
 - 然后open(filename,O_RDONLY)，此时最低可用文件描述符为0，此时所打开的文件连接到标准输入0

方法2: open close dup close

- 第一步打开要重定向的文件: `fd=open(file)`, 此时如果是第一个打开的文件, 那么返回的描述符应该为3
- 第二步关闭标准输入: `close(0)`
- 第三步 复制第一步所得到的文件描述符 `dup(fd)`, 此次复制使用最低可用文件描述符号, 因此, 获得文件描述符0
- 第四步 `close(fd)` 关闭文件的原始连接



open("f",O_RDONLY)

close(0)

dup(fd)

close(fd)

系统调用dup小结

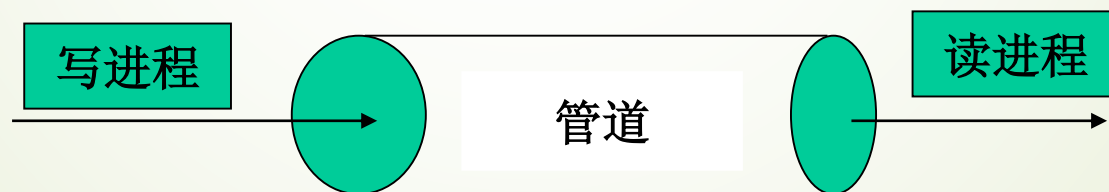
目标	复制一个文件描述符
头文件	<code>#include <unistd.h></code>
函数原型	<code>newfd=dup(oldfd);</code> <code>newfd=dup2(oldfd,newfd)</code>
参数	<code>oldfd</code> 需要复制的文件描述符 <code>newfd</code> 复制 <code>oldfd</code> 后得到的文件描述符
返回值	-1 如果出错 <code>newfd</code> 新的文件描述符

方法3: open...dup2..close

- ➡ `dup2(fd,0)` 相当于 `close(0)`、`dup(fd)` 的效果
- ➡ `dup2(old, new)` 将文件描述符 `old` 复制到 `new`，之前，它先关闭已经存在的连接 `new`

管道通信

- 管道是Linux的一种最简单的通信机制。它在进程之间建立一种逻辑上的管道，一端为流入端，一端为流出端
- 一个进程在流入端写入数据，另外进程在流出段按照流入顺序读出数据，从而实现进程间的通信。



管道的特点

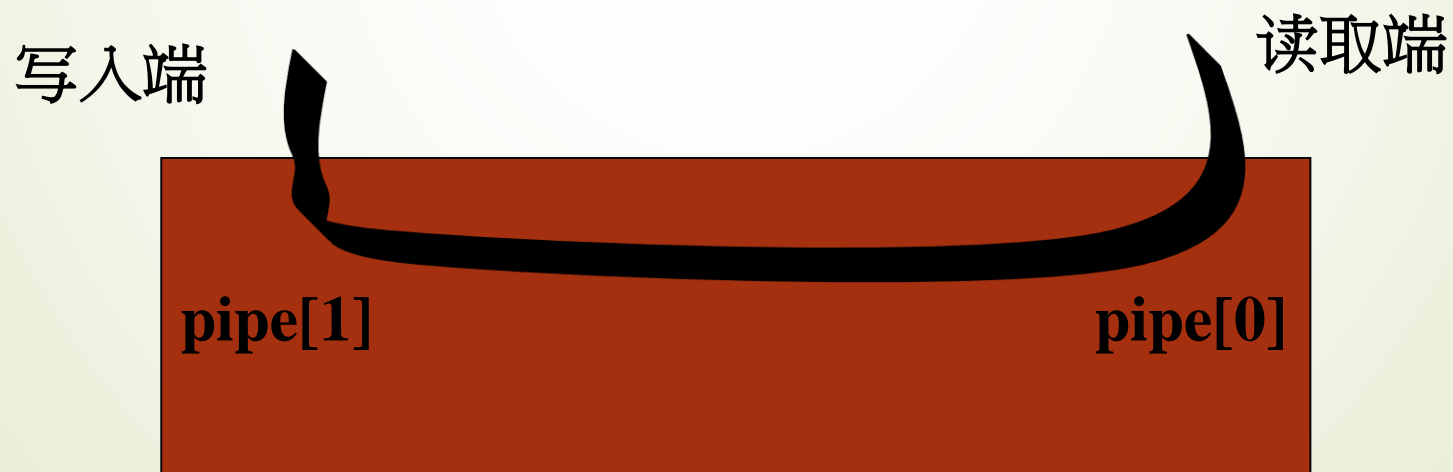
- 1. **单工且单向通信**
 - 要向通过管道实现进程之间的双向通信，需要在进程之间创建两个管道。
- 2. 数据在管道中以**字节流**的形式传送的，即以字节为单位的按照流入顺序传递数据（FIFO方式）。
- **命名管道**和**无名管道**的区别为：
 - 1. 无名管道只能在父子进程之间通信，有名管道可以在任意进程间通信
 - 2. 无名管道没有名字标识，有名管道有名称。

无名管道

头文件	<code>#include <unistd.h ></code>
函数原型	<code>int pipe(int pipe[2])</code>
函数作用	创建无名管道
参数	参数是长度为2的int型数组，创建成功后，该数组里面保存了两个文件描述符， <code>pipe[0]</code> 是读端的文件描述符， <code>pipe[1]</code> 是写端的文件描述符
返回值	成功时，返回0；失败时，返回-1

创建管道

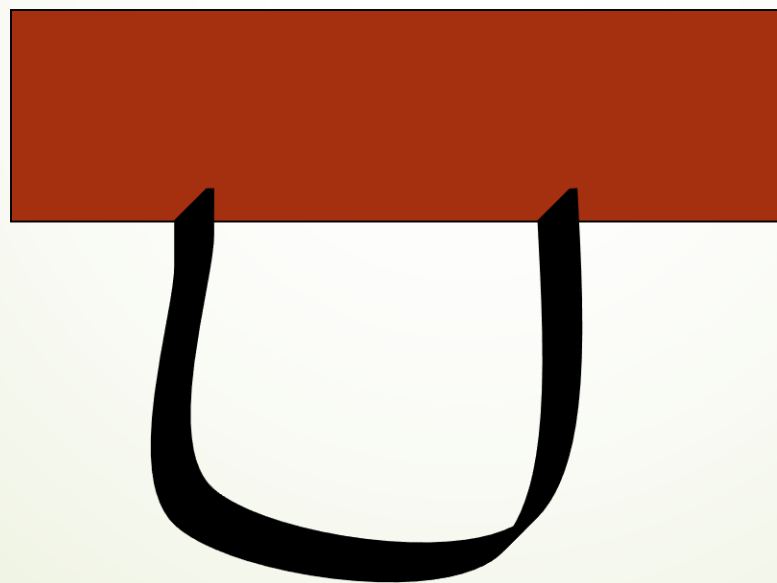
- ➡ pipe创建管道并将其两端连接到两个文件描述符
- ➡ array[0] 为读数据的文件描述符
- ➡ array[1]为写数据端的文件描述符



进程创建管道后

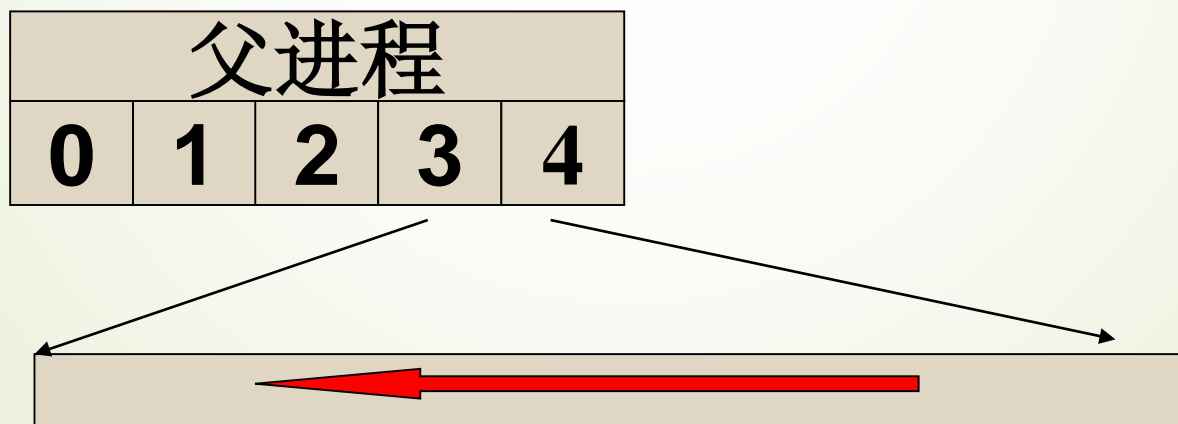
进程创建管道后，管道的读端与写端都与该进程相连

pipe调用也使用最低可用文件描述符



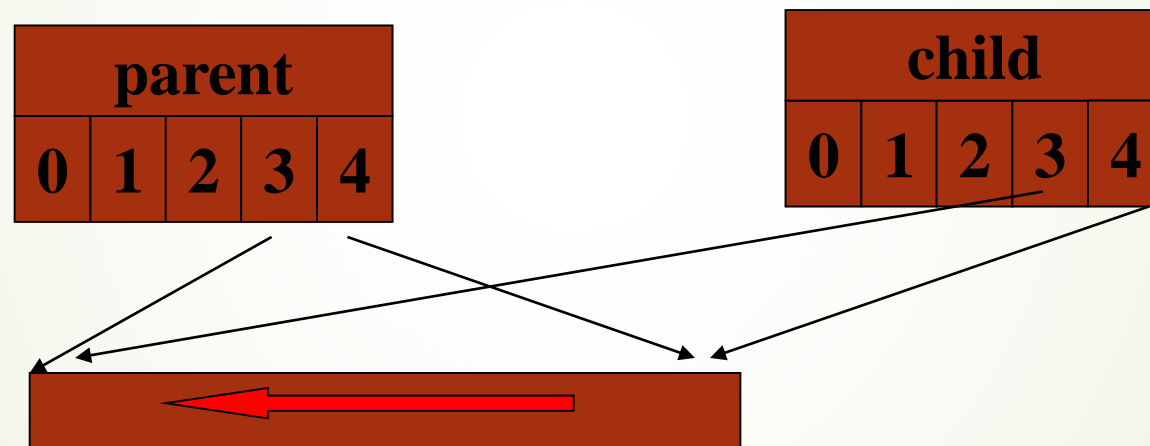
管道通信实例

- 实例7.1 编写一个程序，实现如下功能：创建父子进程，父进程向子进程通过管道发送一个字符串，子进程读取该字符串显示并倒序后发送给父进程，父进程读取该倒序后的字符串并打印出来。



使用fork共享管道

- 进程创建管道后，然后创建子进程，那么子进程与父进程共享该管道



执行过程

父进程				
0	1	2	3	4

子进程				
0	1	2	3	4



父进程				
0	1	2	3	4

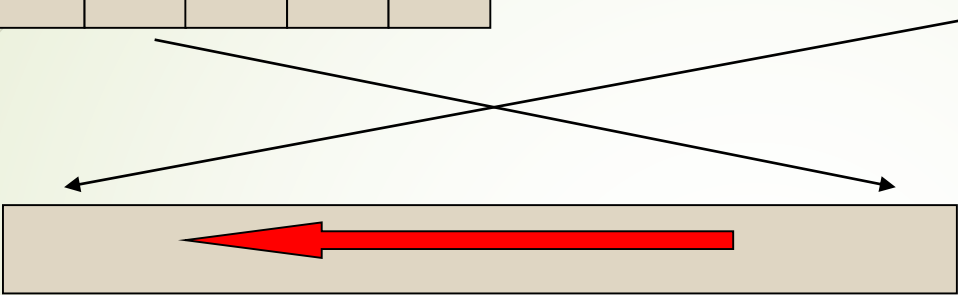
子进程				
0	1	2	3	4





父进程				
0	1	2	3	4

子进程				
0	1	2	3	4





```
main() //pipedemo.c
```

```
{
```

```
    int  len, i, apipe[2];  /* two file descriptors */
```

```
    char  buf[BUFSIZ];      /* for reading end    */
```

```
    if ( pipe ( apipe ) == -1 ){
```

```
        perror("could not make pipe"); exit(1); }
```

```
    printf("Got a pipe! It is file descriptors: { %d %d }\n", apipe[0], apipe[1]);
```

```
    while ( fgets(buf, BUFSIZ, stdin) ){
```

```
        len = strlen( buf );
```

```
        if ( write( apipe[1], buf, len) != len ){ /* send */
```

```
            perror("writing to pipe");          /* down */
```

```
            break;                               /* pipe */
```

```
        }
```

```
        for ( i = 0 ; i<len ; i++ )              /* wipe */
```

```
            buf[i] = 'X' ;
```

```
        len = read( apipe[0], buf, BUFSIZ ) ;    /* read */
```

```
        if ( len == -1 ){                        /* from */
```

```
            perror("reading from pipe"); break;
```

```
        }
```

```
        if ( write( 1 , buf, len ) != len ){    /* send */
```

```
            perror("writing to stdout"); break;
```

```
        } }
```

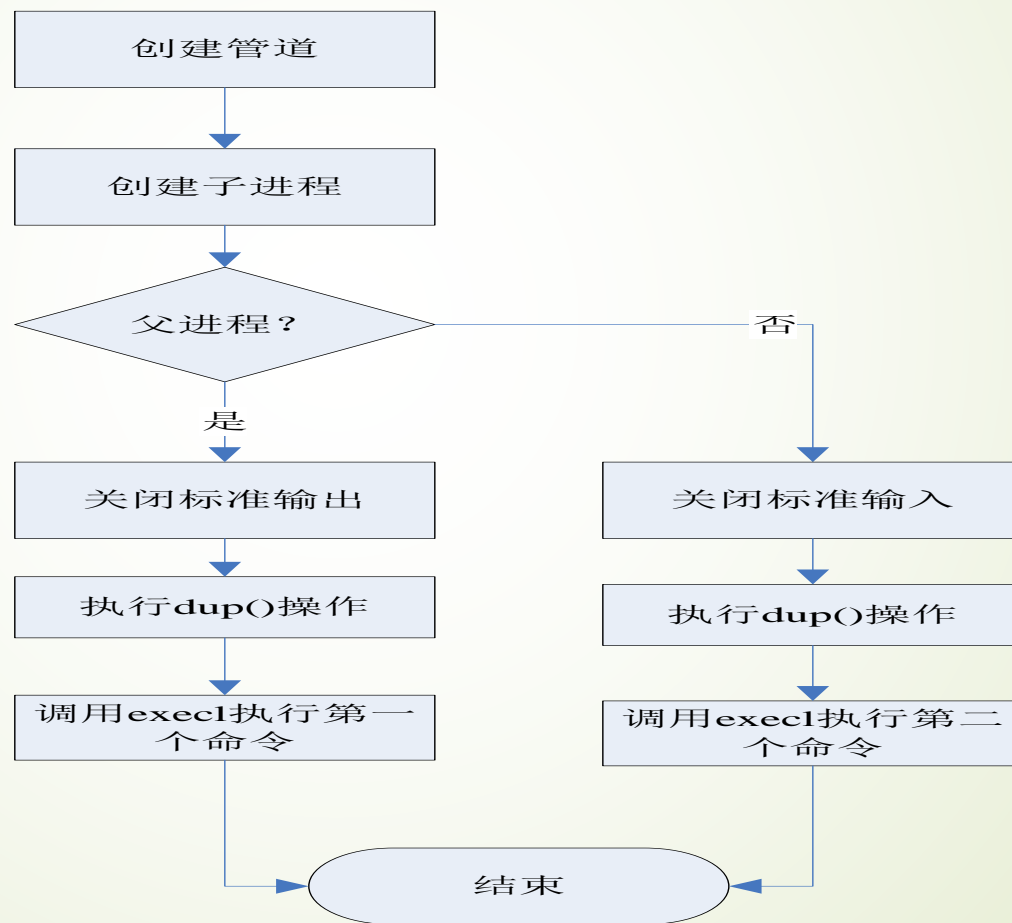
```
}
```

管道与重定向

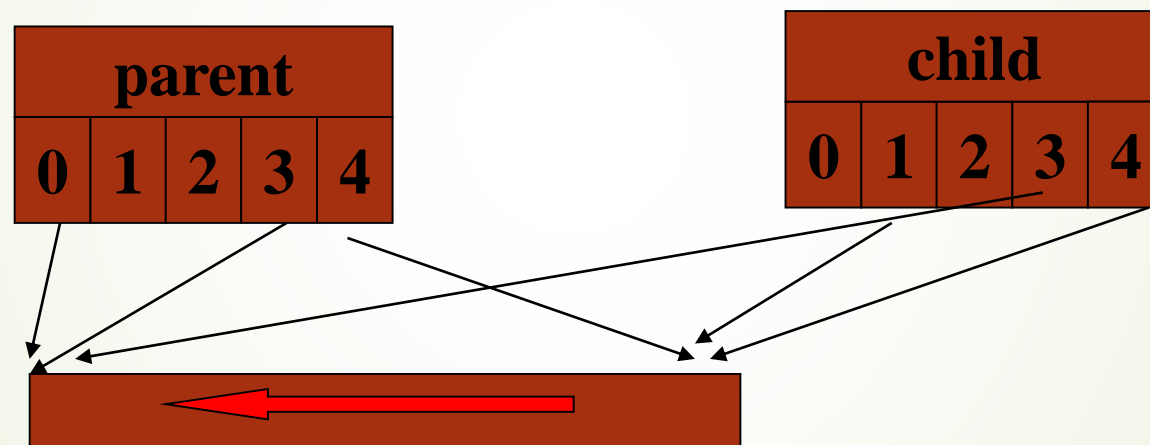
➡ dup系统调用 例7-2和7-3

目标	复制一个文件描述符
头文件	<code>#include <unistd.h></code>
函数原型	<code>newfd=dup(oldfd);</code> <code>newfd=dup2(oldfd,newfd)</code>
参数	<code>oldfd</code> 需要复制的文件描述符 <code>newfd</code> 复制 <code>oldfd</code> 后得到的文件描述符
返回值	-1 如果出错 <code>newfd</code> 新的文件描述符

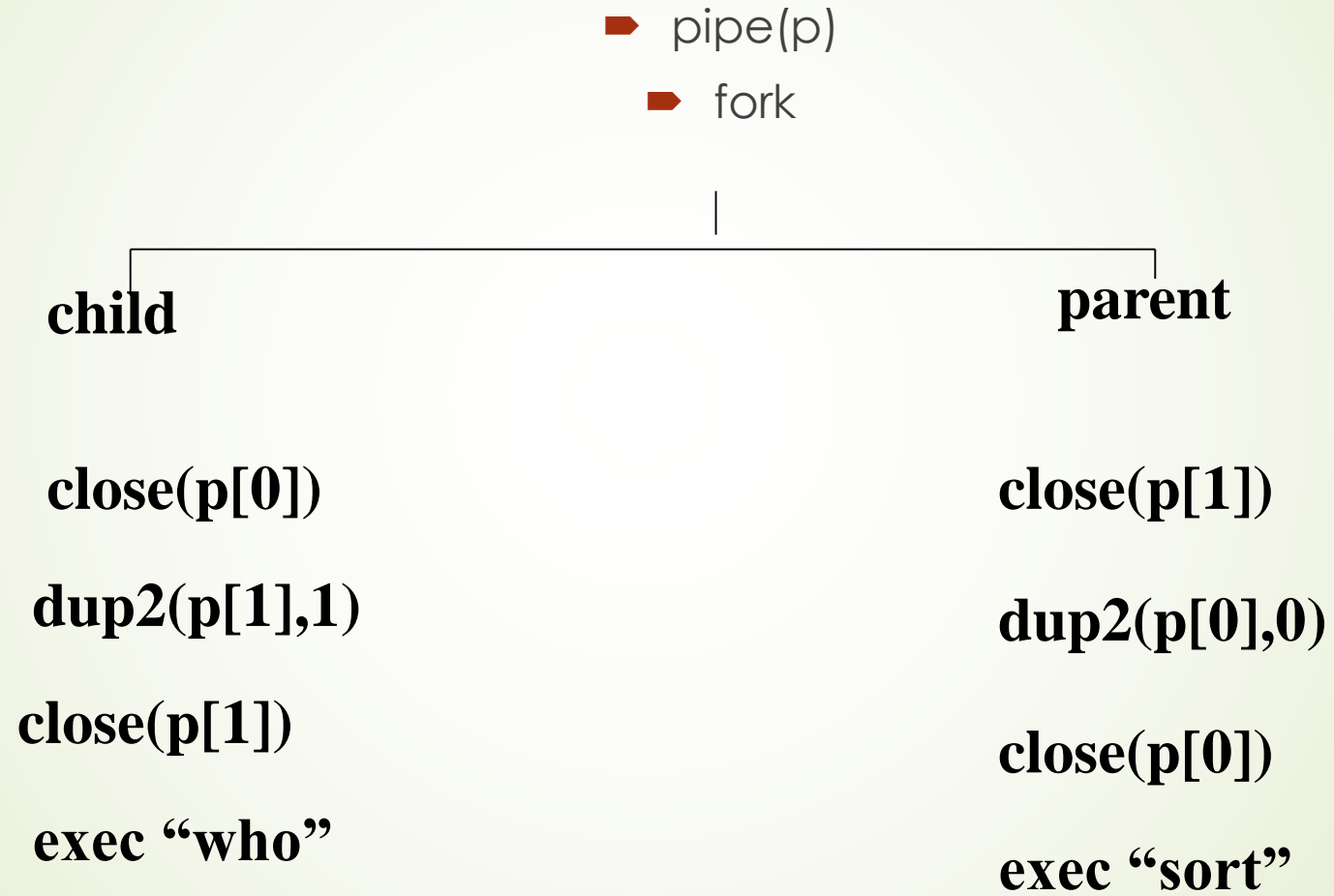
- 请编写程序，实现类似于“ps -aux | grep init”的功能



- p[0] 的文件描述符为3，为读端
- p[1]的文件描述符为4 为写端



pipe的逻辑





通用的程序pipe

- 它使用两个程序的名字做参数
- pipe who sort
- pipe ls head
- pipe在两个参数所代表的程序之间建立管道

命名管道

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

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

```
int mkfifo(const char *filename, mode_t mode);
```

```
int mknod(const char *filename, mode_t mode | S_IFIFO,  
          (dev_t) 0 );
```

```
hadoop@ubuntu:~$ mkfifo mypipe
```

```
hadoop@ubuntu:~$ ls -l mypipe
```


```
prw-rw-r-- 1 hadoop hadoop 0 Dec  9 00:48 mypipe
```

命名管道的注意事项

- 可通过**open**读写操作
- 一般只读或者只写，不建议读写同时进行
- 若需在程序之间双向传递数据，最好使用一对**FIFO**，一个方向使用一个
- 对于该文件，可使用**rm -f**命令删除

命名管道读写规则

- 读方式打开时，**O_NONBLOCK**选项
 - 关闭，则进程阻塞直到有相应进程为写操作而打开该管道文件。
 - 打开，则打开时立刻返回成功，即使没有数据或者没有进程以写的方式打开该管道文件。
- 写方式打开管道文件时，**O_NONBLOCK** 选项
 - 关闭，则进程阻塞直到有相应进程以读打开该文件
 - 打开，则进程立刻返回失败，错误码为**ENXIO**。
 - 例7-4,7-5



思考题

➡ 课后习题1--4

