进程概述

进程控制

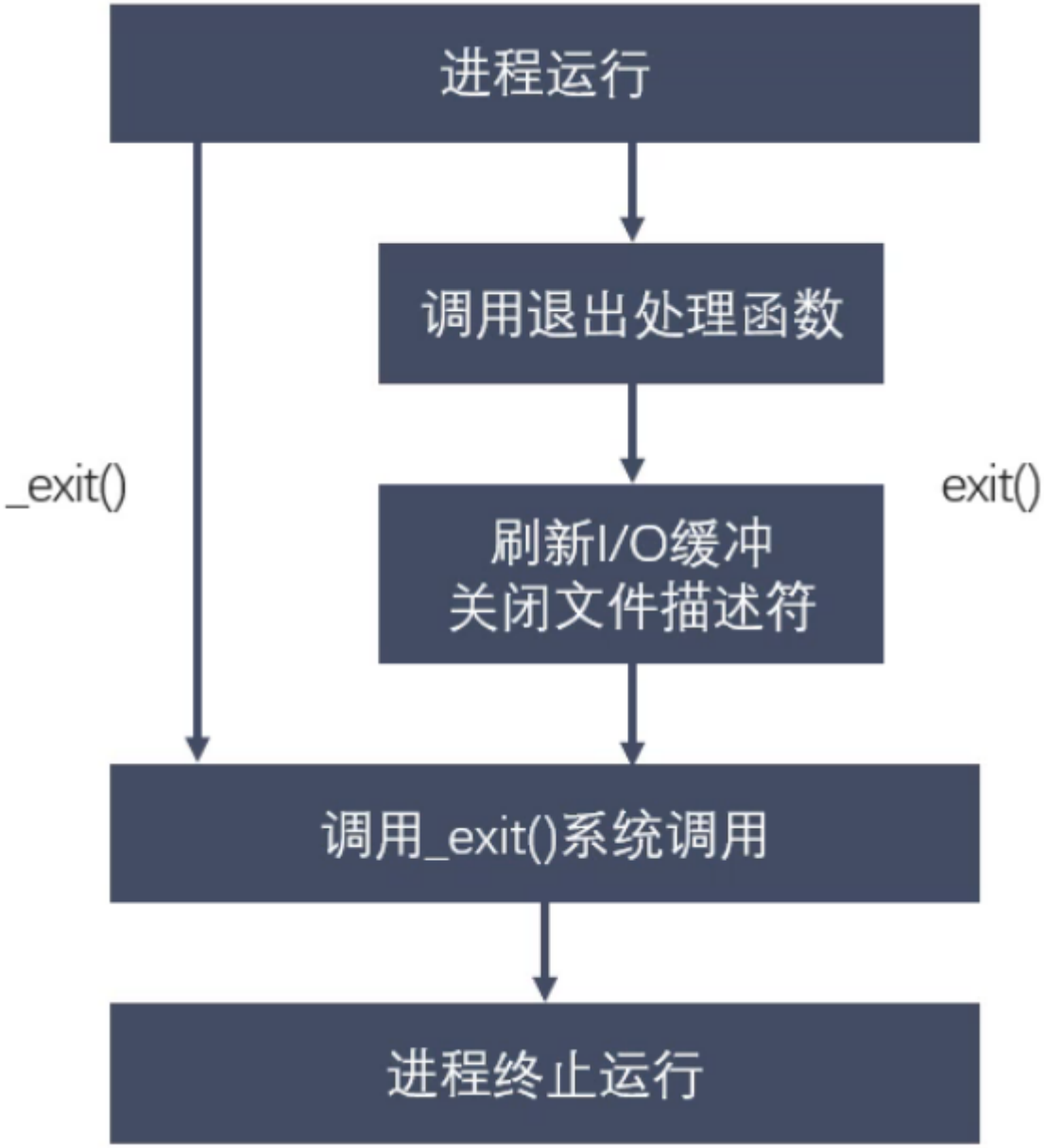
1、fork和exec区别

> fork : 读时共享，写时复制
>
> exec：把进程中的数据替换

进程退出

```
#include <stdlib.h>
// status 是进程退出时的一个状态信息。父进程回收子进程资源的时候可以获取到。
void exit(int status);  //< 标准c库， 多做了2件事情， 看下图

#include <unistd.h>     //< linux系统的
void _exit(int status);
```



这里的进程退出指的是父进程将子进程退出，父进程有义务将子进程的数据资源清理。

孤儿进程

> 父进程运行结束，但子进程还在运行（未运行结束），这样的子进程就被称为孤儿进程（Orphan Process）

1、每当出现一个孤儿进程的时候，内核就会把孤儿进程的父进程设置为init，而init进程会循环地wait()它的已经退出的子进程。这样，当一个孤儿进程凄凉地结束了其生命周期的时候，init进程就会代表党和政府出面处理它的一切善后工作。

2、孤儿进程并不会有什么危害。

例子：

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

int main()
{
    pid_t pid = fork();

    if(pid > 0)
    {
        printf("i am parent process, pid : %d, ppid : %d\n", getpid(), getppid());
    }
    else if(pid == 0)
    {
        sleep(1);  // 等待1s是为了让父进程退出
        printf("i am child process, pid : %d, ppid : %d\n", getpid(), getppid());

    }

    for(int i =0; i < 3; i++)
    {
        printf("i : %d, pid : %d\n", i, getpid());
    }
    return 0;
}

=======运行后
root@ubuntu:~/home/2# ./orphan
i am parent process, pid : 20164, ppid : 20001
i : 0, pid : 20164
i : 1, pid : 20164
```

```
i : 2, pid : 20164
root@ubuntu:~/home/2# i am child process, pid : 20165, ppid : 1   // 可以看到这里子进程的父进程换为1了
i : 0, pid : 20165
i : 1, pid : 20165
i : 2, pid : 20165

```

僵尸进程

> 每个进程结束之后，都**会释放自己地址空间中的用户区数据**，内核区的PCB没有办法自己释放掉，需要父进程去释放。进程终止时，父进程尚未回收，子进程残留资源（PCB）存放于内核中， 变成僵尸进程（Zombie）

1、僵尸进程不能被kill -9 杀死

2、会导致一个问题，如果父进程不调用wait（）或waitpid()的话，那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵尸进程，将因为没有可用的进程号而导致系统不能产生新的进程，此即为僵尸进程的危害，应当避免。

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    pid_t pid = fork();

    if(pid > 0)
    {
        while(1)
        {
            printf("i am parent process, pid : %d, ppid : %d\n", getpid(), getppid());
            sleep(1);
        }
```

```
16
17      }
18      else if(pid == 0)
19      {
20          // sleep(1);
21          printf("i am child process, pid : %d, ppid : %d\n", getpid(), getppid());
22
23      }
24
25      for(int i =0; i < 3; i++)
26      {
27          printf("i : %d, pid : %d\n", i, getpid());
28      }
29      return 0;
30  }
31  ============================
32  root@ubuntu:~/home/2# ./zombie
33  i am parent process, pid : 20235, ppid : 20001
34  i am child process, pid : 20236, ppid : 20235
35  i : 0, pid : 20236
36  i : 1, pid : 20236
37  i : 2, pid : 20236
38  i am parent process, pid : 20235, ppid : 20001
39  i am parent process, pid : 20235, ppid : 20001
40  i am parent process, pid : 20235, ppid : 20001
41  i am parent process, pid : 20235, ppid : 20001
42  ...
43
44  root@ubuntu:~/home/2# ps aux
45  USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
46  root     20247  0.0  0.0   4516   724 pts/0    S+   09:17   0:00 ./zombie
47  root     20248  0.0  0.0      0     0 pts/0    Z+   09:17   0:00 [zombie] <defunct>
48  这个僵尸进程是不能用kill -9 杀死的， 演示是用的Ctrl + C断开的
49
```

(进程回收)wait函数

1、在每个进程退出的时候，内核释放该进程所有资源、包括打开的文件、占用的内存等。但是仍然为其保留一定的信息，这些信息主要指进程控制块PCB的信息（包括进程号、退出状态、运行时间等）

2、父进程可以通过调用wait或waitpid 得到它的退出状态同时彻底清楚掉这个进程

3、wait（）和waitpid（）函数功能一样，区别在于，wait()函数会阻塞，waitpid()函数可以设置不阻塞，waitpid还可以指定等待哪个子进程结束。

【注意】:一次wait 或waitpid 调用只能清理一个子进程，清理多个子进程应使用循环。

```
WIFEXITED(status) 非0，进程正常退出
WEXITSTATUS(status) 如果上宏为真，获取进程退出的状态（exit的参数）
WIFSIGNALED(status) 非0，进程异常终止
WTERMSIG(status) 如果上宏为真，获取使进程终止的信号编号
WIFSTOPPED(status) 非0，进程处于暂停状态
WSTOPSIG(status) 如果上宏为真，获取使进程暂停的信号的编号
WIFCONTINUED(status) 非0，进程暂停后已经继续运行
```

```
/*
   #include <sys/types.h>
   #include <sys/wait.h>
   pid_t wait(int *wstatus);
      功能：等待任意一个子进程结束，如果任意一个子进程结束了，次函数会回收子进程的资源。
      参数：int *wstatus
         进程退出时的状态信息，传入的是一个int类型的地址，传出参数。
      返回值：
         - 成功：返回被回收的子进程的id
         - 失败：-1 (所有的子进程都结束，调用函数失败)

   调用wait函数的进程会被挂起（阻塞），直到它的一个子进程退出或者收到一个不能被忽略的信号时才被唤醒（相当于继续往下执行）
   如果没有子进程了，函数立刻返回，返回-1；如果子进程都已经结束了，也会立即返回，返回-1.

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

```c
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {

    // 有一个父进程，创建5个子进程（兄弟）
    pid_t pid;

    // 创建5个子进程
    for(int i = 0; i < 5; i++) {
        pid = fork();
        if(pid == 0) {    // 为了防止孙子进程出现
            break;
        }
    }

    if(pid > 0) {
        // 父进程
        while(1) {
            printf("parent, pid = %d\n", getpid());

            // int ret = wait(NULL);
            int st;
            int ret = wait(&st);   // 阻塞

            if(ret == -1) {
                break;
            }

            if(WIFEXITED(st)) {
                // 是不是正常退出
                printf("退出的状态码：%d\n", WEXITSTATUS(st));
            }
            if(WIFSIGNALED(st)) {
                // 是不是异常终止

                printf("被哪个信号干掉了：%d\n", WTERMSIG(st));
```

```
        }

        printf("child die, pid = %d\n", ret);

        sleep(1);
    }

} else if (pid == 0){
    // 子进程
     while(1) {
        printf("child, pid = %d\n",getpid());
        sleep(1);
     }

    exit(0);
}

return 0; // exit(0)
}
```

waitpid函数

> 上面解释

```
/*
  #include <sys/types.h>
  #include <sys/wait.h>
  pid_t waitpid(pid_t pid, int *wstatus, int options);
    功能：回收指定进程号的子进程，可以设置是否阻塞。
    参数：
      - pid:
        pid > 0 : 某个子进程的pid
        pid = 0 : 回收当前进程组的所有子进程
        pid = -1 : 回收所有的子进程，相当于 wait()  （最常用）
        pid < -1 : 某个进程组的组id的绝对值，回收指定进程组中的子进程
      - options：设置阻塞或者非阻塞

        0 : 阻塞
```

```
            WNOHANG : 非阻塞
        - 返回值:
            > 0 : 返回子进程的id
            = 0 : options=WNOHANG, 表示还有子进程或者
            = -1 ： 错误，或者没有子进程了
*/
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {

    // 有一个父进程，创建5个子进程（兄弟）
    pid_t pid;

    // 创建5个子进程
    for(int i = 0; i < 5; i++) {
        pid = fork();
        if(pid == 0) {
            break;
        }
    }

    if(pid > 0) {
        // 父进程
        while(1) {
            printf("parent, pid = %d\n", getpid());
            sleep(1);

            int st;
            // int ret = waitpid(-1, &st, 0);
            int ret = waitpid(-1, &st, WNOHANG);

            if(ret == -1) {   // 没有子进程了
                break;

            } else if(ret == 0) {
```

```
            // 说明还有子进程存在
            continue;
        } else if(ret > 0) {

            if(WIFEXITED(st)) {
                // 是不是正常退出
                printf("退出的状态码：%d\n", WEXITSTATUS(st));
            }
            if(WIFSIGNALED(st)) {
                // 是不是异常终止
                printf("被哪个信号干掉了：%d\n", WTERMSIG(st));
            }
            printf("child die, pid = %d\n", ret);
        }
    }
  } else if (pid == 0){
    // 子进程
     while(1) {
        printf("child, pid = %d\n",getpid());
        sleep(1);
     }
    exit(0);
  }

  return 0;
}
```

进程间通信

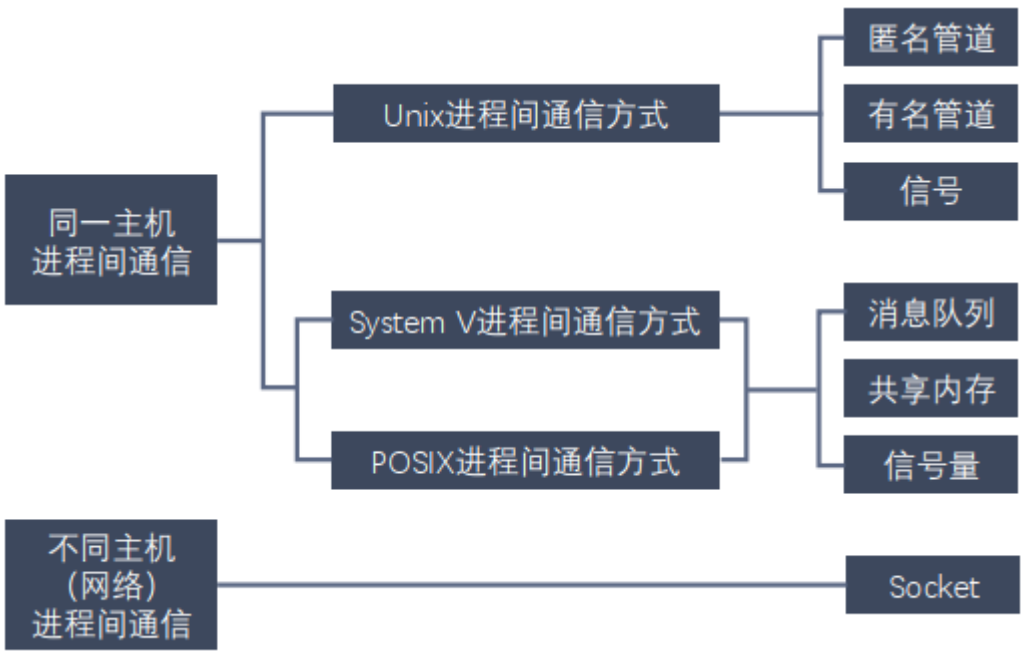
概念

1、进程是一个独立的资源分配单元，不同进程（这里所说的进程通常指的是用户进程）之间的资源是独立的，没有关联，不能在一个进程中直接访问另一个进程的资源。

2、进程不是孤立的、不同的进程需要进行信息的交互和状态的传递等，因此需要进程间通信（IPC: Inter Processes Communication）

3、进程间通信的目的：

- 数据传输：一个进程需要将它的数据发送给另一个进程（边下载边播放）
- 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）（下载完了需要通知其他线程）
- 资源共享：多个进程之间共享同样的资源。为了做到这一点，需要内核提供互斥和同步机制（几个播放同一个文件）
- 进程控制：有些进程希望完全控制另一个进程的执行（如 Debug 进程，GDB调试），此时控制 进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变



同步，异步，阻塞，非阻塞

> 同步/异步关注的是消息通信机制 (synchronous communication/ asynchronous communication) 。
>
> 所谓同步，就是在发出一个调用时，在没有得到结果之前， 该调用就不返回。
>
> 异步则是相反，调用在发出之后，这个调用就直接返回了，所以没有返回结果
>
> 阻塞/非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态.

> 阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。
>
> 非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

匿名管道

> 概念

1、管道也叫无名（匿名）管道，它是是 UNIX 系统 IPC（进程间通信）的最古老形式， 所有的 UNIX 系统都支持这种通信机制

2、统计一个目录中文件的数目命令： ls | wc –l，为了执行该命令（"|" 这个就叫管道）， shell 创建了两 个进程来分别执行 ls 和 wc

> 特点

1、管道其实是一个在内核内存中维护的缓冲器，这个缓冲器的存储能力是有限的，不同的 操作系统大小不一定相同

2、管道拥有文件的特质：读操作、写操作，匿名管道没有文件实体，有名管道有文件实体， 但不存储数据。可以按照操作文件的方式对管道进行操作

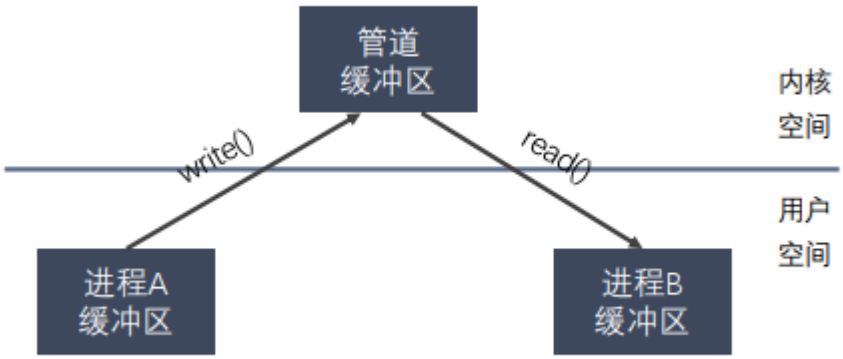
3、一个管道是一个字节流，使用管道时不存在消息或者消息边界的概念，从管道读取数据 的进程可以读取任意大小的数据块，而不管写入进程写入管道的数据块的大小是多少

4、通过管道传递的数据是顺序的，从管道中读取出来的字节的顺序和它们被写入管道的顺 序是完全一样的

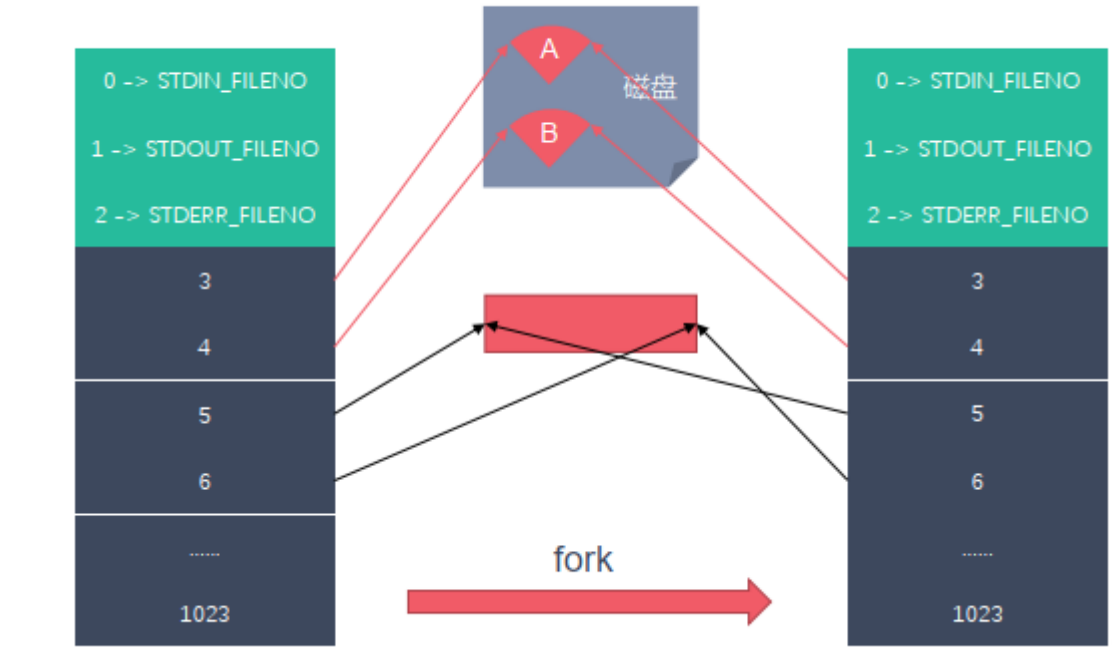
5、 在管道中的数据的传递方向是单向的，一端用于写入，一端用于读取，管道是半双工的

6、从管道读数据是一次性操作，数据一旦被读走，它就从管道中被抛弃，释放空间以便写 更多的数据，在管道中无法使用 lseek() 来随机的访问数据

7、匿名管道只能在具有公共祖先的进程（父进程与子进程，或者两个兄弟进程，具有亲缘 关系）之间使用

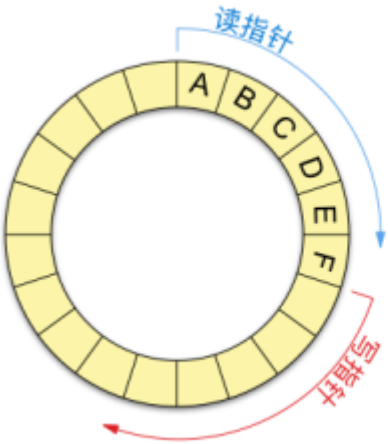


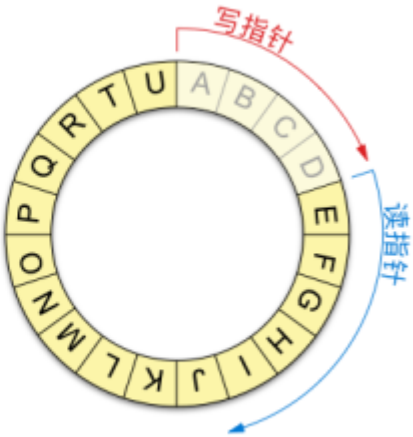
> 为什么管道可以进行进程间通信？或者说管道只能在具有公共祖先的进程间使用?



子进程fork出来后，是与父进程共享文件描述符的。

> 管道的数据结构





匿名管道的使用

```
创建匿名管道
#include <unistd.h>
int pipe(int pipefd[2]);

查看管道缓冲大小命令
ulimit -a

查看管道缓冲大小函数
#include <unistd.h>
long fpathconf(int fd, int name);

```

```
/*
   #include <unistd.h>
   int pipe(int pipefd[2]);
      功能：创建一个匿名管道，用来进程间通信。
      参数：int pipefd[2] 这个数组是一个传出参数。
         pipefd[0] 对应的是管道的读端
         pipefd[1] 对应的是管道的写端
      返回值：
         成功 0
         失败 -1
```

```
11
12    管道默认是阻塞的：如果管道中没有数据，read阻塞，如果管道满了，write阻塞
13
14    注意：匿名管道只能用于具有关系的进程之间的通信（父子进程，兄弟进程）
15 */
16
17 // 子进程发送数据给父进程，父进程读取到数据输出
18 #include <unistd.h>
19 #include <sys/types.h>
20 #include <stdio.h>
21 #include <stdlib.h>
22 #include <string.h>
23
24 int main() {
25
26     // 在fork之前创建管道
27     int pipefd[2];
28     int ret = pipe(pipefd);
29     if(ret == -1) {
30         perror("pipe");
31         exit(0);
32     }
33
34     // 创建子进程
35     pid_t pid = fork();
36     if(pid > 0) {
37         // 父进程
38         printf("i am parent process, pid : %d\n", getpid());
39
40         // 关闭写端
41         close(pipefd[1]);
42
43         // 从管道的读取端读取数据
44         char buf[1024] = {0};
45         while(1) {
46             int len = read(pipefd[0], buf, sizeof(buf));
47             printf("parent recv : %s, pid : %d\n", buf, getpid());
48
```

```
49          // 向管道中写入数据
50          //char * str = "hello,i am parent";
51          //write(pipefd[1], str, strlen(str));
52          //sleep(1);
53      }
54
55  } else if(pid == 0){
56      // 子进程
57
58      // 关闭读端
59      close(pipefd[0]);
60      char buf[1024] = {0};
61      while(1) {
62          // 向管道中写入数据
63          char * str = "hello,i am child";
64          write(pipefd[1], str, strlen(str));
65          //sleep(1);
66
67          // int len = read(pipefd[0], buf, sizeof(buf));
68          // printf("child recv : %s, pid : %d\n", buf, getpid());
69          // bzero(buf, 1024);
70      }
71
72  }
73  return 0;
74 }
```