fork()+pipe() --> 父子进程间通过管道通信

1.fork()函数：创建新进程

头文件 ：#include <unistd.h>

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

函数原型：pid_t fork(**void**);

返回值：一个是子进程返回0，第二个是父进程的返回值大于0，错误返回-1

功能：创建一个新的进程。（pid_t 是一个宏定义，其实质是int 被定义在**#include**<sys/types.h>中）

来看一下fork之后，发生了什么事情。

由fork创建的新进程被称为子进程（child process）。该函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是0，而父进程的返回值则是新进程（子进程）的进程 id。将子进程id返回给父进程的理由是：因为一个进程的子进程可以多于一个，没有一个函数使一个进程可以获得其所有子进程的进程id。对子进程来说，之所以fork返回0给它，是因为它随时可以调用getpid()来获取自己的pid；也可以调用getppid()来获取父进程的id。（进程id为0的总是由交换进程使用，所以一个子进程的进程id不可能为0）。

fork之后，操作系统会复制一个与父进程完全相同的子进程。虽说是父子关系，但是在操作系统看来，他们更像兄弟关系，这2个进程共享代码空间，但是数据空间是互相独立的，子进程数据空间中的内容是父进程的完整拷贝，指令指针也完全相同，子进程拥有父进程当前运行到的位置（两进程的程序计数器pc值相同。也就是说，子进程是从fork返回处开始执行的），但有一点不同，如果fork成功，子进程中fork的返回值是0，父进程中fork的返回值是子进程的进程号，如果fork不成功，父进程会返回错误。

可以这样想象，2个进程一直同时运行，而且步调一致，在fork之后，他们分别作不同的工作，也就是分岔了。

注意：fork()函数主要是以父进程为蓝本复制一个进程，其ID号和父进程的ID号不同。对于结果fork出来的子进程的父进程ID号是执行fork()函数的进程的ID号；

例如：

父进程， fork返回值是：17025， ID：17024 ，父进程ID：16879

子进程， fork返回值是：0， ID：17025 ，父进程ID：17024

使用管道有一些限制：
两个进程通过一个管道只能实现单向通信，如果有时候也需要子进程写父进程读，就必须另开一个管道。
管道的读写端通过打开的文件描述符来传递，因此要通信的两个进程必须从它们的公共祖先那里继承管道文件描述符。也就是需要通过fork传递文件描述符使两个进程都能访问同一管道，它们才能通信。

使用管道需要注意以下4种特殊情况（假设都是阻塞I/O操作，没有设置O_NONBLOCK标志）：
1. 如果所有指向管道写端的文件描述符都关闭了（管道写端的引用计数等于0），而仍然有进程从管道的读端读数据，那么管道中剩余的数据都被读取后，再次read会返回0，就像读到文件末尾一样。
2. 如果有指向管道写端的文件描述符没关闭（管道写端的引用计数大于0），而持有管道写端的进程也没有向管道中写数据，这时有进程从管道读端读数据，那么管道中剩余的数据都被读取后，再次read会阻塞，直到管道中有数据可读了才读取数据并返回。
3. 如果所有指向管道读端的文件描述符都关闭了（管道读端的引用计数等于0），这时有进程向管道的写端write，那么该进程会收到信号SIGPIPE，通常会导致进程异常终止。
4. 如果有指向管道读端的文件描述符没关闭（管道读端的引用计数大于0），而持有管道读端的进程也没有从管道中读数据，这时有进程向管道写端写数据，那么在管道被写满时再次write会阻塞，直到管道中有空位置了才写入数据并返回。

2.pipe()函数：建立管道

头文件： #include<unistd.h>
函数原型：int pipe(int filedes[2]);
函数说明：pipe()会建立管道，并将文件描述词由参数filedes数组返回。

filedes[0]为管道里的读取端
filedes[1]则为管道的写入端。

返回值： 若成功则返回零，否则返回-1，错误原因存于errno中。

错误代码:

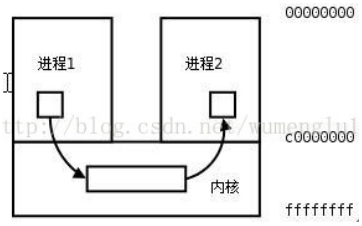
EMFILE 进程已用完文件描述词最大量
ENFILE 系统已无文件描述词可用。
EFAULT 参数 filedes 数组地址不合法。

调用pipe函数时在内核中开辟一块缓冲区（称为管道）用于通信，它有一个读端一个写端，然后通过filedes参数传出给用户程序两个文件描述 符，filedes[0]指向管道的读端，filedes[1]指向管道的写端（很好记，就像0是标准输入1是标准输出一样）。所以管道在用户程序看起来 就像一个打开的文件，通过read(filedes[0]);或者write(filedes[1]);向这个文件读写数据其实是在读写内核缓冲区。

注意：int pipe(int filedes[2]) 中的两个文件描述符被强制规定filedes[0]只能指向管道的读端，如果进行写操作就会出现错误；同理filedes[1]只能指向管道的写端，如果进行读操作就会出现错误。

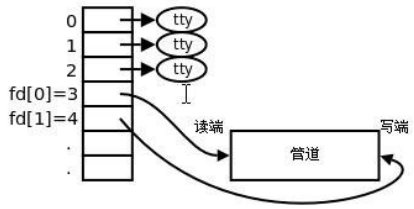
3.父子进程通过管道通信

每个进程各自有不同的用户地址空间，任何一个进程的全局变量在另一个进程中都看不到，所以进程之间要交换数据必须通过内核，在内核中开辟一块缓冲区，进程1把数据从用户空间拷到内核缓冲区，进程2再从内核缓冲区把数据读走，内核提供的这种机制称为进程间通信（IPC，InterProcess Communication）。

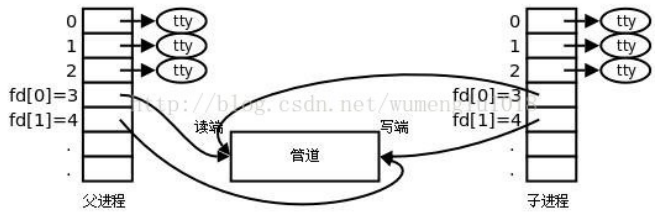


开辟了管道之后如何实现两个进程间的通信呢？比如可以按下面的步骤通信。

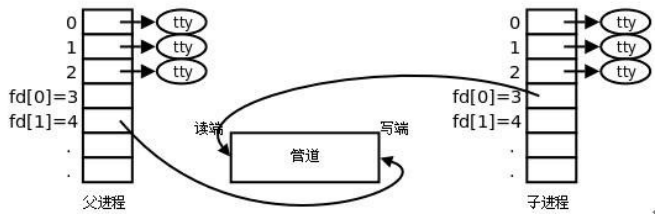
1. 父进程创建管道



2. 父进程fork出子进程



3. 父进程关闭fd[0]，子进程关闭fd[1]



1. 父进程调用pipe开辟管道，得到两个文件描述符指向管道的两端。
2. 父进程调用fork创建子进程，那么子进程也有两个文件描述符指向同一管道。
3. 父进程关闭管道读端，子进程关闭管道写端。父进程可以往管道里写，子进程可以从管道里读，管道是用环形队列实现的，数据从写端流入从读端流出，这样就实现了进程间通信。

之前一直没明白为什么在这里父进程要关闭管道读端，并且子进程要关闭管道写端，想了很久终于想通了...，原因如下：因为上面的这个程序是要模拟父进程和子进程的管道读写操作，其中父进程用于向管道中写入数据，子进程用于向管道中读取数据，因此开始要关闭父进程的读文件描述符filedes[0], 以及关闭子进程的写文件描符filedes[1]，这是为了模拟这个过程。

至于为什么父进程关闭管道的读文件描述符filedes[0]后子进程还能读取管道的数据，是因为系统维护的是一个文件的文件描述符表的计数，父子进程都各自有指向相同文件的文件描述符，当关闭一个文件描述符时，相应计数减一，当这个计数减到0时，文件就被关闭，因此虽然父进程关闭了其文件描述符filedes[0]，但是这个文件的文件描述符计数还没等于0，所以子进程还可以读取。也可以这么理解，父进程和子进程都有各自的文件描述符，因此虽然父进程中关闭了filedes[0]，但是对子进程中的filedes[0]没有影响。

文件表中的每一项都会维护一个引用计数，标识该表项被多少个文件描述符(fd)引用，在引用计数为0的时候，表项才会被删除。所以调用close(fd)关闭子进程的文件描述符，只会减少引用计数，但是不会使文件表项被清除，所以父进程依旧可以访问。

最后需要注意，在linux的pipe管道下，在写端进行写数据时，不需要关闭读端的缓冲文件(即不需要读端的文件描述符计数为0)，但是在读端进行读数据时必须先关闭写端的缓冲文件(即写端的文件描述符计数为0)然后才能读取数据。

简单例子：

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

```
int main(void){
  pid_t pid;
  int i=0;
  int result = -1;
  int fd[2],nbytes;
  char string[100];
  char readbuffer[80];
  int *write_fd = &fd[1];
  int *read_fd = &fd[0];
  printf("Please input data:");
  scanf("%s",string);
  result = pipe(fd);
  if(-1 == result)
   {
     perror("pipe");
     return -1;
   }

  pid=fork();
  if(-1 == pid) //此处为了验证父子进程是否创建成功，如果未创建成功，则返回-1
   {
     perror("fork");
     return -1;
   }
  else if(0 == pid)
   {
     printf("this is child %d\n", getpid());
     close(*read_fd);
     result = write(*write_fd,string,strlen(string));
     return 0;
   }
  else
   {
           printf("this is parent %d\n", getpid());
     close(*write_fd);
     nbytes = read(*read_fd,readbuffer,sizeof(readbuffer)-1);
     printf("receive %d data: %s\n",nbytes,readbuffer);
   }
return 0;
}
```

执行结果：

```
[root@localhost Chapter8]# vi fuzipipe1.c
[root@localhost Chapter8]# gcc -o fuzipipe1 fuzipipe1.c
[root@localhost Chapter8]# ./fuzipipe1
Please input data: Hello
this is child 15451
this is parent 15450
receive 5 data: "Hello"
[root@localhost Chapter8]# ./fuzipipe1
Please input data: AHaHeiHuha7800
this is child 15453
this is parent 15452
receive 14 data: "AHaHeiHuha7800"
[root@localhost Chapter8]#
```