@toc

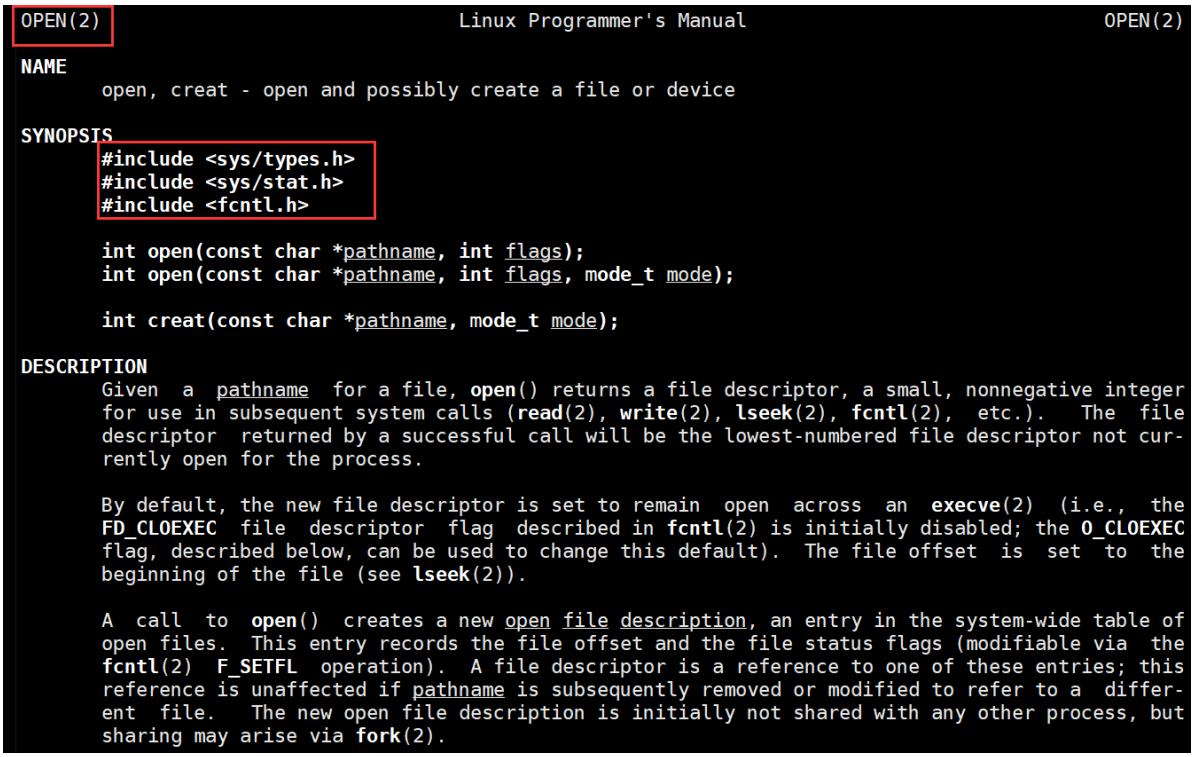
文件描述符fd

- 我们之前学过输出重定向，在linux中它的本质是指把stdout的内容重定向到文件中，而不会去重定向stderr

操作系统是文件的管理者，所有语言上的对"文件"的操作，都必须贯穿OS，又因为操作系统不相信任何人，访问操作系统需要通过系统调用接口，所以几乎所有的语言fopen,fclose,fwrite,fgets,fputs,fgetc,fputc等底层一定需要使用OS提供的系统调用，为了能够更深入的了解文件操作，就需要去学习文件的系统调用接口。

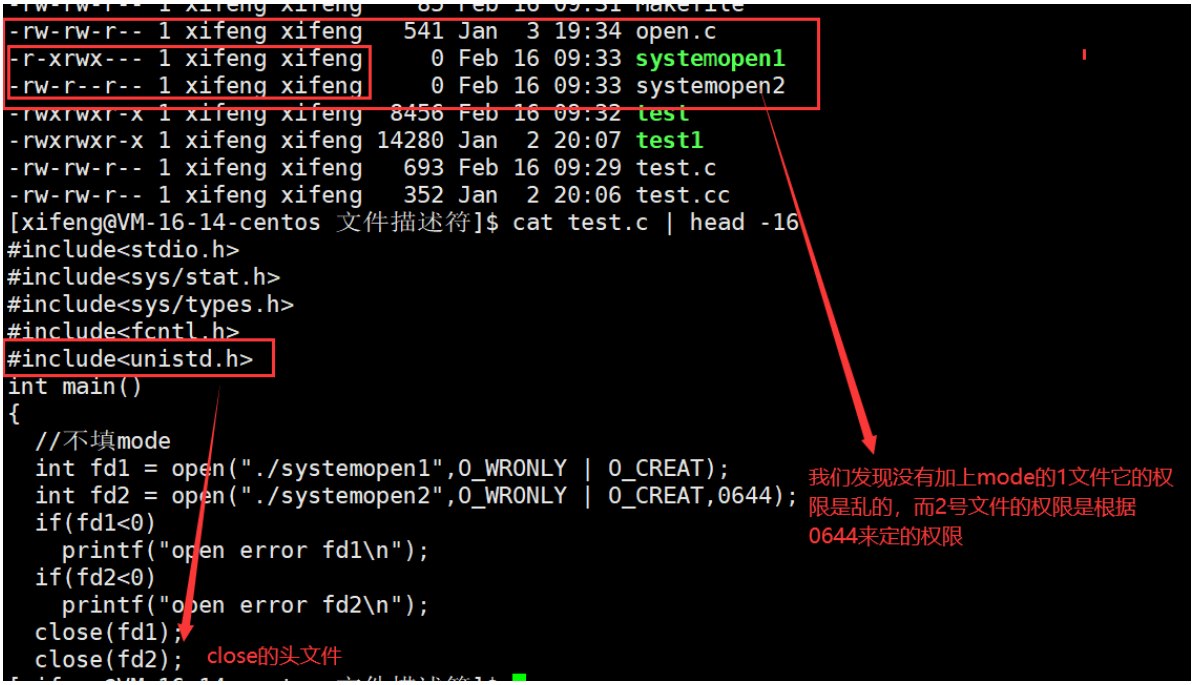
系统文件IO接口介绍

open



```
OPEN(2)                          Linux Programmer's Manual                          OPEN(2)

NAME
       open, creat - open and possibly create a file or device

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

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

       int creat(const char *pathname, mode_t mode);

DESCRIPTION
       Given  a  pathname  for a file, open() returns a file descriptor, a small, nonnegative integer
       for use in subsequent system calls (read(2), write(2), lseek(2), fcntl(2),  etc.).   The  file
       descriptor  returned by a successful call will be the lowest-numbered file descriptor not cur-
       rently open for the process.

       By default, the new file descriptor is set to remain  open  across  an  execve(2)  (i.e.,  the
       FD_CLOEXEC  file  descriptor  flag  described in fcntl(2) is initially disabled; the O_CLOEXEC
       flag, described below, can be used to change this default).  The file offset  is  set  to  the
       beginning of the file (see lseek(2)).

       A  call  to  open()  creates a new open file description, an entry in the system-wide table of
       open files.  This entry records the file offset and the file status flags (modifiable via  the
       fcntl(2)  F_SETFL  operation).  A file descriptor is a reference to one of these entries; this
       reference is unaffected if pathname is subsequently removed or modified to refer to a  differ-
       ent  file.   The new open file description is initially not shared with any other process, but
       sharing may arise via fork(2).
```

```
int open(char* pathname,int flags,mode_t mode);
//flags-->打开方式，通过传bit位的方式，传多组标记
//O_RDONLY: 只读打开
//O_WRONLY: 只写打开
//O_RDWR : 读，写打开
//这三个常量，必须指定一个且只能指定一个
//O_CREAT : 若文件不存在，则创建它。需要使用mode选项，来指明新文件的访问权限
//O_APPEND: 追加写
//mode-->最常用的是权限
//返回的是一个文件描述符
```



- flags的运行原理可以理解为flags是一个设定了的值，通过给定的O_WRONLY,O_RDONLY.....来与flags按位与

```
if(O_WRONLY & flags)
{
    //....
}
//O_WRONLY可以理解为
#define O_WRONLY 0x1 0000 0001
//O_RDONLY可以理解为
#define O_RDONLY 0x2 0000 0010
//O_CREAT可以理解为
#define O_CREAT 0x4  0000 0100
//所以如果需要传多组标志就可以运用或，也就是上图的O_WRONLY | O_CREAT
```

close

```
CLOSE(2)                        Linux Programmer's Manual                        CLOSE(2)

NAME
       close - close a file descriptor

SYNOPSIS
       #include <unistd.h>

       int close(int fd);

DESCRIPTION
       close()  closes  a file descriptor, so that it no longer refers to any file and may be reused.
       Any record locks (see fcntl(2)) held on the file it was associated  with,  and  owned  by  the
       process, are removed (regardless of the file descriptor that was used to obtain the lock).

       If  fd  is  the  last  file  descriptor referring to the underlying open file description (see
       open(2)), the resources associated with the open file description are freed; if the descriptor
       was the last reference to a file which has been removed using unlink(2) the file is deleted.

RETURN VALUE
       close() returns zero on success.  On error, -1 is returned, and errno is set appropriately.

ERRORS
       EBADF  fd isn't a valid open file descriptor.

       EINTR  The close() call was interrupted by a signal; see signal(7).

       EIO    An I/O error occurred.

CONFORMING TO
       SVr4, 4.3BSD, POSIX.1-2001.
```

用法其实就是关闭文件，里面的参数是open的返回值，也就文件描述符fd

write

```
WRITE(2)                        Linux Programmer's Manual                        WRITE(2)

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

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

DESCRIPTION
       write()  writes  up  to count bytes from the buffer pointed buf to the file
       referred to by the file descriptor fd.

       The number of bytes written may be less than count if, for  example,  there
       is  insufficient  space  on  the  underlying  physical  medium,  or  the
       RLIMIT_FSIZE resource limit is encountered (see setrlimit(2)), or the  call
       was  interrupted  by  a signal handler after having written less than count
       bytes.  (See also pipe(7).)

       For a seekable file (i.e., one to which lseek(2) may be applied, for  exam-
       ple,  a  regular  file) writing takes place at the current file offset, and
       the file offset is incremented by the number of bytes actually written.  If
       the  file  was open(2)ed with O_APPEND, the file offset is first set to the
       end of the file before writing.  The adjustment of the file offset and  the
       write operation are performed as an atomic step.

       POSIX  requires that a read(2) which can be proved to occur after a write()
       has returned returns the new data.  Note that  not  all  file  systems  are
       POSIX conforming.
```

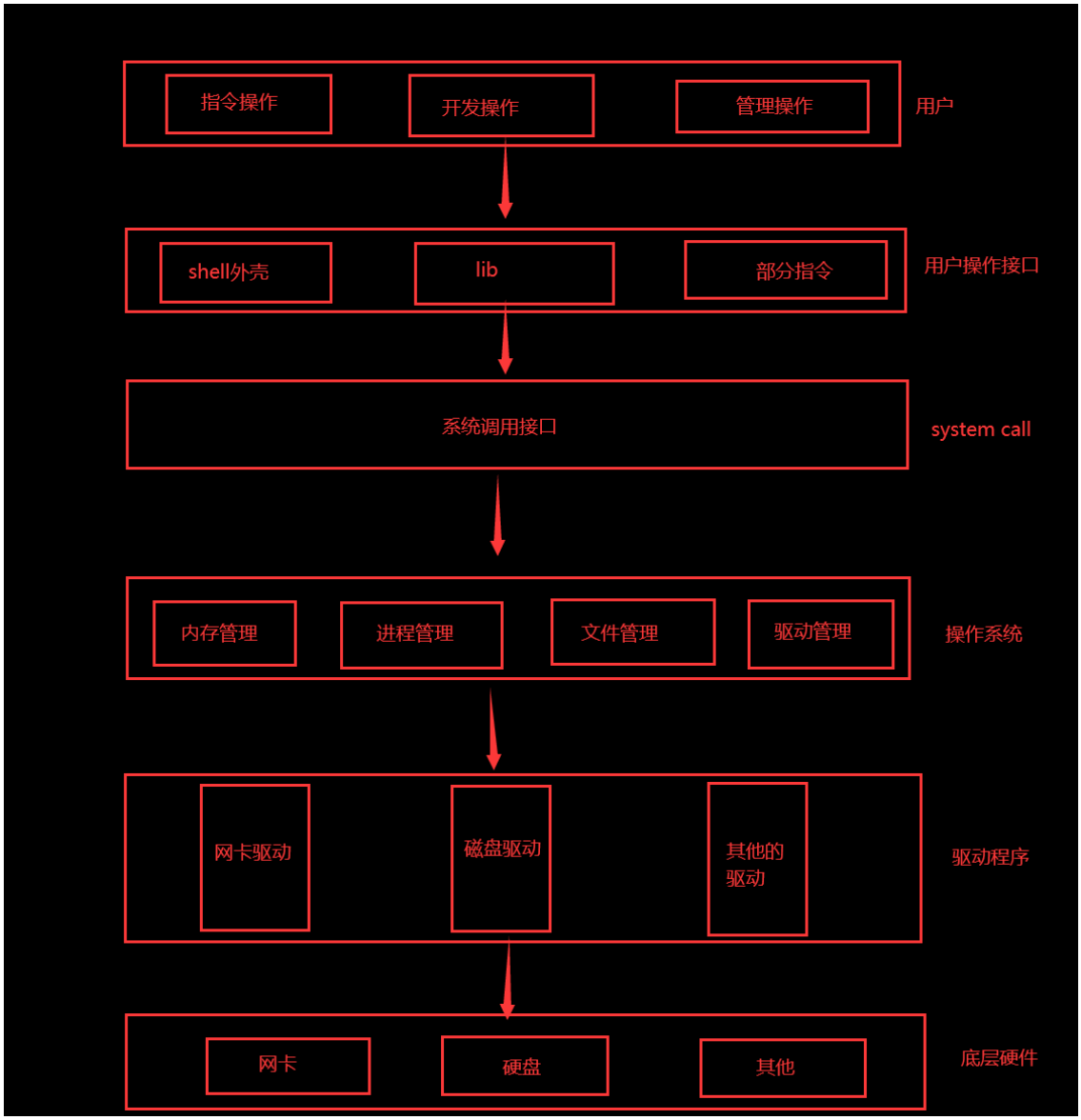
```
total 48
-rw-r--r-- 1 xifeng xifeng     9 Feb 16 09:51 file
-rw-rw-r-- 1 xifeng xifeng    85 Feb 16 09:31 Makefile
-rw-rw-r-- 1 xifeng xifeng   541 Jan  3 19:34 open.c
-rwxrwxr-x 1 xifeng xifeng  8512 Feb 16 09:51 test
-rwxrwxr-x 1 xifeng xifeng 14280 Jan  2 20:07 test1
-rw-rw-r-- 1 xifeng xifeng   656 Feb 16 09:51 test.c
-rw-rw-r-- 1 xifeng xifeng   352 Jan  2 20:06 test.cc
[xifeng@VM-16-14-centos 文件描述符]$ cat file
hello fd
[xifeng@VM-16-14-centos 文件描述符]$ cat test.c | head -19
#include<stdio.h>
#include<string.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
int main()
{
  //不填mode
  int fd = open("./file",O_WRONLY | O_CREAT,0644);
  //write的头文件也是unistd
  if(fd>=0)
  {
   const char*str ="hello fd\n";
   write(fd,str,strlen(str));
  }
  close(fd);
}

[xifeng@VM-16-14-centos 文件描述符]$
```

三个参数从左到右依次
fd：文件描述符也就是open的返回值
buf：要写入的数据
size_t：写入多少

open的函数返回值

在认识返回值之前，我们先来了解一下**系统调用和库函数**

- 像fopen,fclose,fread,fwrite都是C标准库中的函数，也就是库函数(libc)
- 上面的open，close，write都属于系统接口

- 

  从这个图中看系统调用接口和库函数的关系就清晰了，可以认为f#系列的库函数，都是对系统调用的封装，方便二次开发

文件描述符fd通过对open的学习，可知文件描述符就是一个小整数

0 & 1 & 2

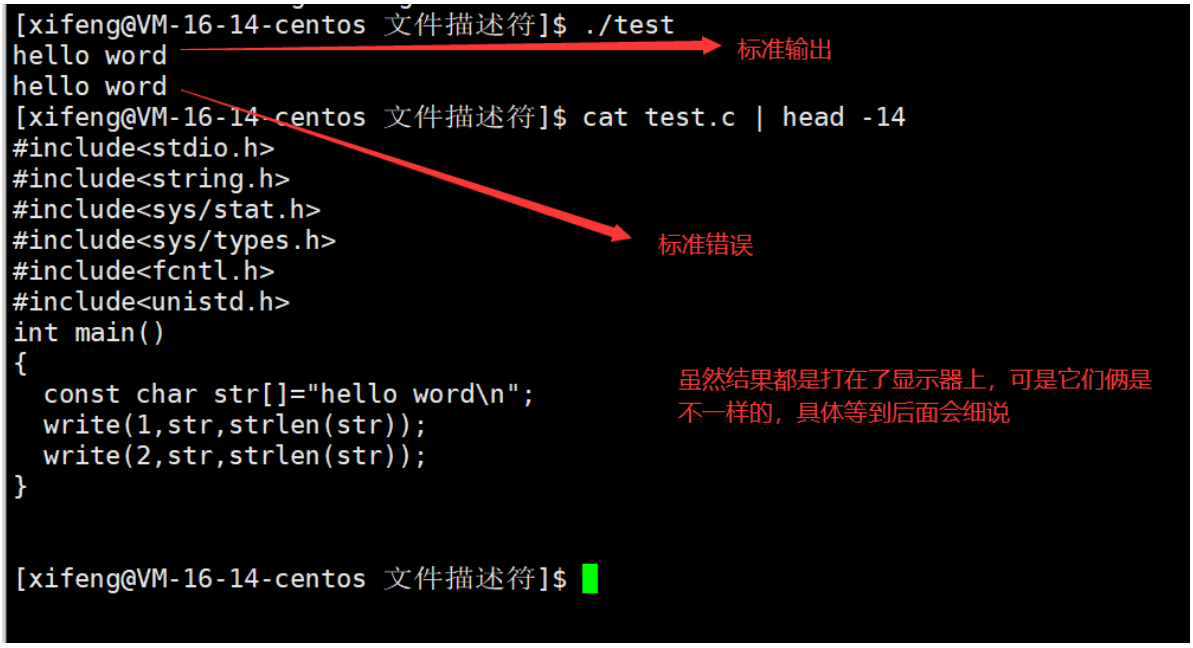
Linux进程默认情况下，OS会帮助我们进程打开三个标准输入输出！

0：标准输入，键盘

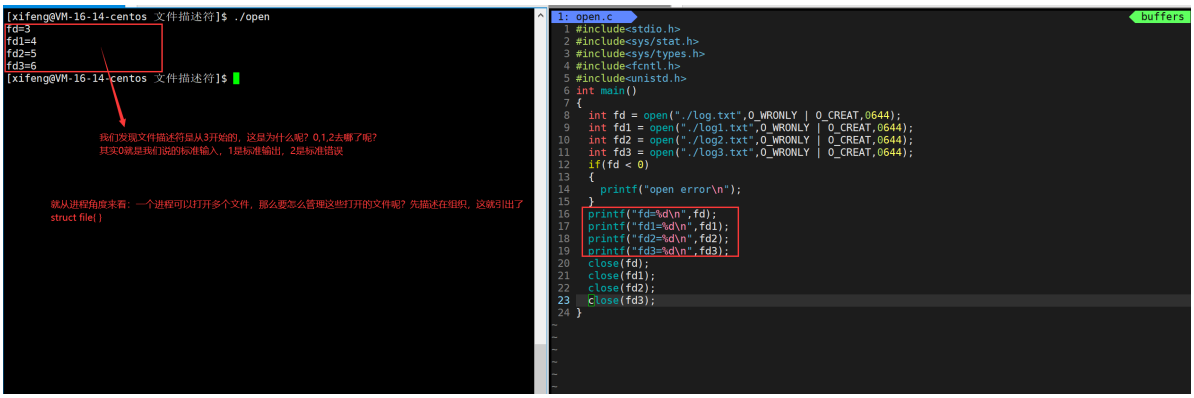
1：标准输入，显示器

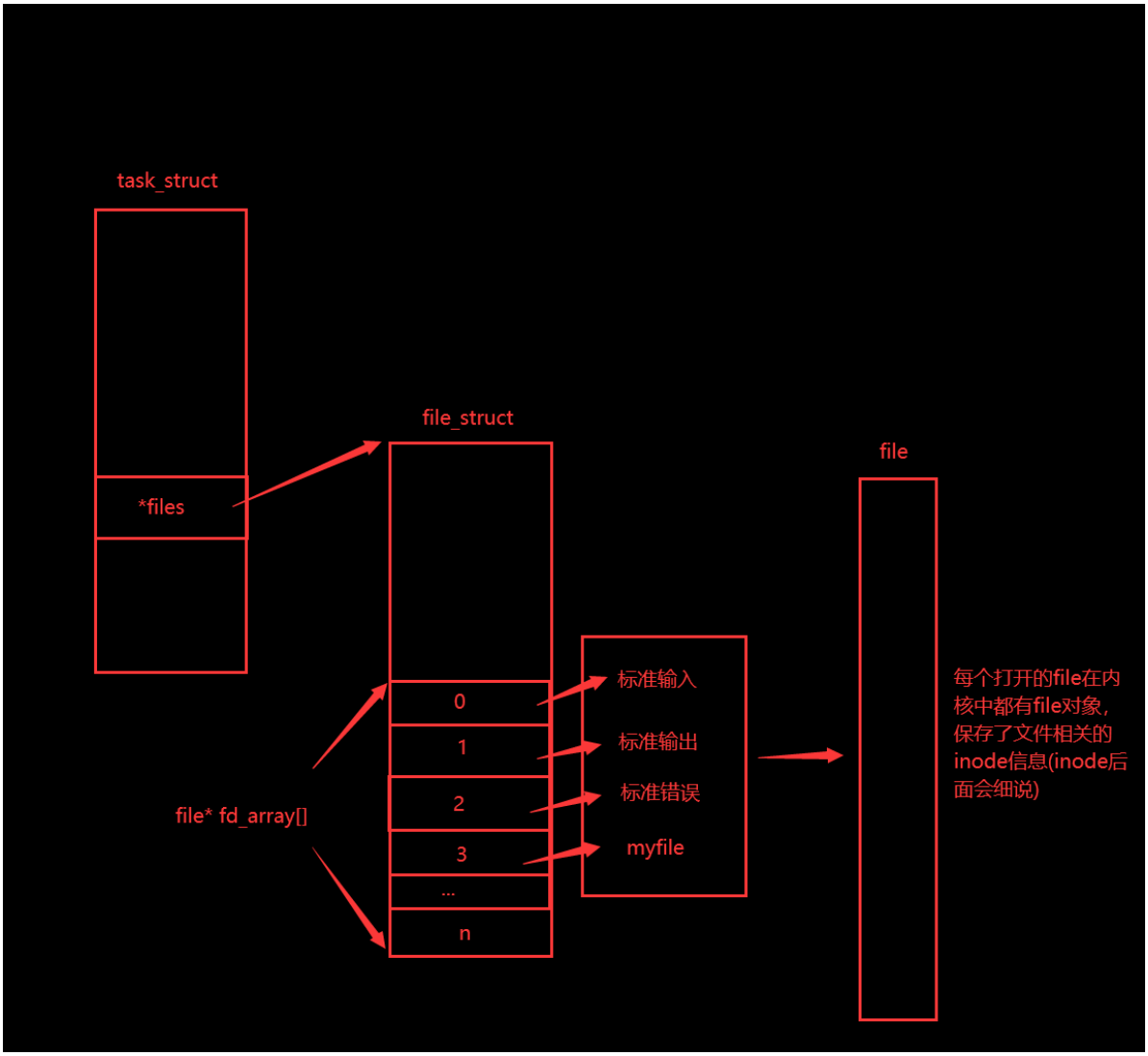
2：标准错误，显示器

所以输入输出还有这种方式：




```c
#include<stdio.h>
#include<string.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
int main()
{
  const char str[]="hello word\n";
  write(1,str,strlen(str));
  write(2,str,strlen(str));
}
```





现在就知道了，文件描述符就是从0开始的小整数，当我们打开文件时，操作系统在内存中要创建相应的数据结构来描述目标文件，这就有了file结构体。进程执行open系统调用，所以必须让进程和文件关联起来。每个进程都有一个指针*files, 指向一张表files_struct,该表最重要的部分就是包涵一个指针数组，每个元素都是一个指向打开文件的指针，所以，本质上，文件描述符就是该数组的下标。所以，只要拿着文件描述符，就可以找到对应的文件

```
struct file{
 //包含了打开文件的相关信息
 //链接属性
    file* fd_array[];
}
```

文件描述符的分配规则

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
    int fd = open("myfile", O_RDONLY);
    if(fd < 0)
    {
        perror("open");
        return 1;
```

```
    }
    printf("fd: %d\n", fd);
    close(fd);
    return 0;
}
//结果fd：3
//关闭0,或者2
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
    close(0);
    //close(2);
    int fd = open("myfile", O_RDONLY);
    if(fd < 0)
    {
        perror("open");
        return 1;
    }
    printf("fd: %d\n", fd);
    close(fd);
    return 0;
}
//结果fd:0，fd：2
```

由此可见文件描述符分配规则：files_struct数组当中，找到当前没有被使用的最小的一个下标，作为新文件描述符

重定向

在此之前我们先来看一个现象：

接下来就开始完善我们的另外一个代码

```c
int main()
{
  //实现重定向
  //我们知道了文件描述符分配规则
  close(1);//关闭标准输出
  int fd = open("./myfile.txt",O_WRONLY | O_CREAT,0644);
  if(fd<0)
    return -1;
  printf("hello word\n");
  fprintf(stdout,"hello stdout\n");
  fprintf(stderr,"hello stderr\n");
}
```

printf是C语言库当中的IO函数，一般往 stdout 中输出，但是stdout底层访问文件的时候，找的还是fd:1, 但此时，fd:1下标所表示内容，已经变成了myfile的地址，不再是显示器文件的地址，所以，输出的任何消息都会往文件中写入，进而完成输出重定向

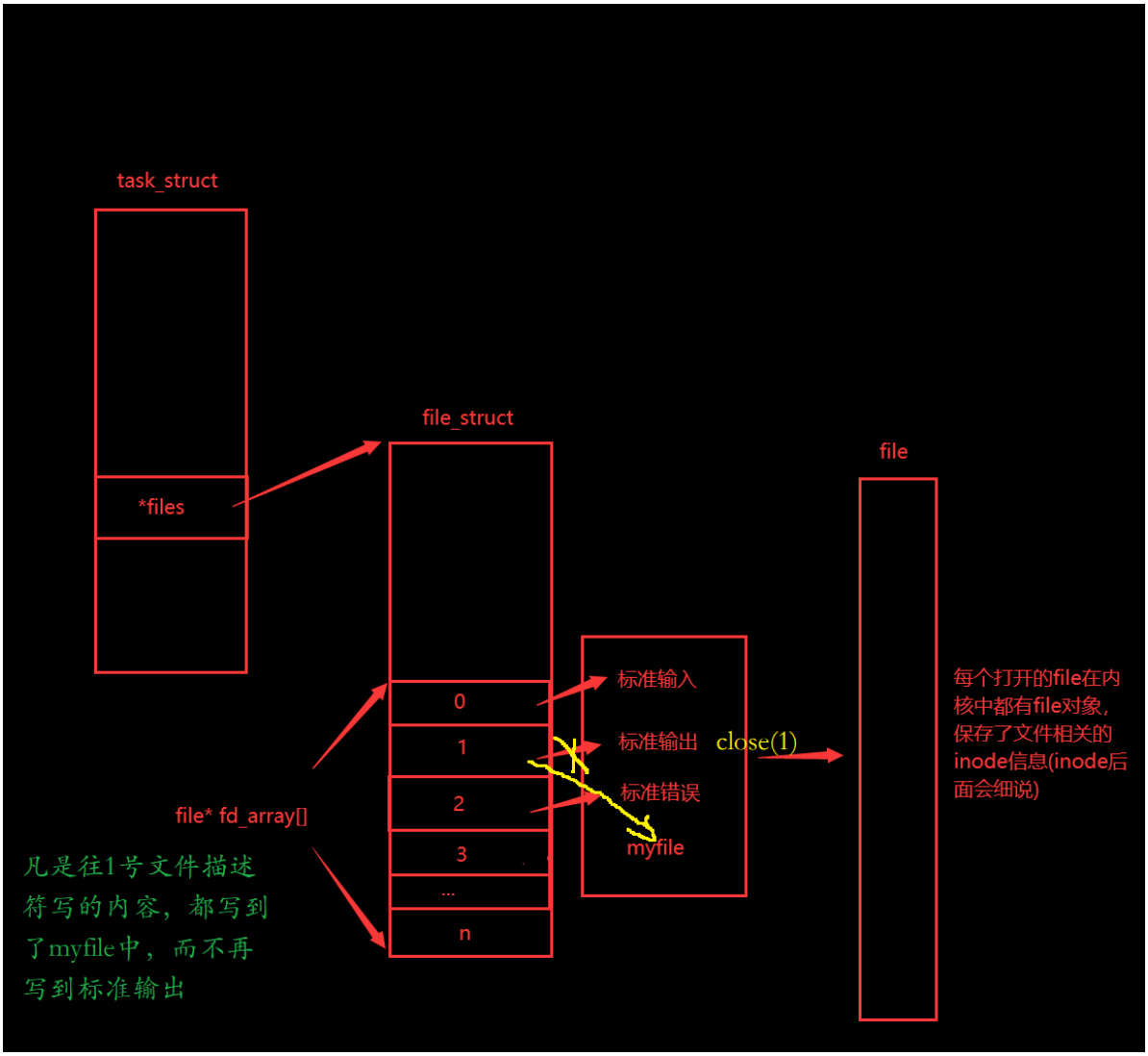
```
[xifeng@VM-16-14-centos 文件描述符]$ cat test.c | head -20
#include<stdio.h>
#include<string.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
int main()
{
  //实现重定向
  //我们知道了文件描述符分配规则
  close(1);//关闭标准输出
  int fd = open("./myfile.txt",O_WRONLY | O_CREAT,0644);
  if(fd<0)
    return -1;
  printf("hello word\n");
  fprintf(stdout,"hello stdout\n");
  fprintf(stderr,"hello stderr\n");
}


[xifeng@VM-16-14-centos 文件描述符]$ ./test
hello stderr
[xifeng@VM-16-14-centos 文件描述符]$ cat myfile.txt
hello word
hello stdout
[xifeng@VM-16-14-centos 文件描述符]$
```

与上面的重定向结果一致

还有一个细节，这里并没有在最后close(fd),否则就是另外一个现象了，这个现象后面会说，解决方法也很简单在关闭fd之前先刷新一下缓冲区就好了

```
[xifeng@VM-16-14-centos 文件描述符]$ ./test
hello stderr
[xifeng@VM-16-14-centos 文件描述符]$ cat myfile.txt
[xifeng@VM-16-14-centos 文件描述符]$ cat test.c | head -20
#include<stdio.h>
#include<string.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
int main()
{
  //实现重定向
  //我们知道了文件描述符分配规则
  close(1);//关闭标准输出
  int fd = open("./myfile.txt",O_WRONLY | O_CREAT,0644);
  if(fd<0)
    return -1;
  printf("hello word\n");
  fprintf(stdout,"hello stdout\n");
  fprintf(stderr,"hello stderr\n");
  // fflush(stdout);//如果不加上刷新缓冲区会导致myfile.txt没有数据
  close(fd);
}
[xifeng@VM-16-14-centos 文件描述符]$ ./test
hello stderr
[xifeng@VM-16-14-centos 文件描述符]$ cat myfile.txt
hello word
hello stdout
[xifeng@VM-16-14-centos 文件描述符]$ cat test.c | head -20
#include<stdio.h>
#include<string.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
int main()
{
  //实现重定向
  //我们知道了文件描述符分配规则
  close(1);//关闭标准输出
  int fd = open("./myfile.txt",O_WRONLY | O_CREAT,0644);
  if(fd<0)
    return -1;
  printf("hello word\n");
  fprintf(stdout,"hello stdout\n");
  fprintf(stderr,"hello stderr\n");
  fflush(stdout);//如果不加上刷新缓冲区会导致myfile.txt没有数据
  close(fd);
}
```


到这里重定向的本质到底是什么呢?



就是将本应该写到标准输出里面的东西写到标准输入中去

- 小知识：2&>1 就是将标准错误重定向到标准输出

```
[xifeng@VM-16-14-centos 文件描述符]$ ./test
hello word
hello stdout
hello stderr
[xifeng@VM-16-14-centos 文件描述符]$ ./test &> myfile.txt
[xifeng@VM-16-14-centos 文件描述符]$ cat myfile.txt
hello stderr
hello word
hello stdout
```

dup2系统调用

```
DUP(2)                                Linux Programmer's Manual                                DUP(2)

NAME
       dup, dup2, dup3 - duplicate a file descriptor

SYNOPSIS
       #include <unistd.h>

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

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <fcntl.h>              /* Obtain O_* constant definitions */
       #include <unistd.h>

       int dup3(int oldfd, int newfd, int flags);

DESCRIPTION
       These system calls create a copy of the file descriptor oldfd.

       dup() uses the lowest-numbered unused descriptor for the new descriptor.

       dup2() makes newfd be the copy of oldfd, closing newfd first if necessary, but note the following:

       *  If oldfd is not a valid file descriptor, then the call fails, and newfd is not closed.

       *  If oldfd is a valid file descriptor, and newfd has the same value as oldfd, then dup2() does noth-
          ing, and returns newfd.

       After a successful return from one of these system calls, the old and new  file  descriptors  may  be
       used interchangeably.  They refer to the same open file description (see open(2)) and thus share file
       offset and file status flags; for example, if the file offset is modified by using lseek(2) on one of
       the descriptors, the offset is also changed for the other.

       The  two  descriptors do not share file descriptor flags (the close-on-exec flag).  The close-on-exec
       flag (FD_CLOEXEC; see fcntl(2)) for the duplicate descriptor is off.
```

```c
//头文件
#include<unistd.h>
//我们一般使用dup2比较的多
int dup2(int oldfd,int newfd);
//可以思考一下含义
//就是让newfd成为oldfd的副本
//以下是个例子
-----------------------------------------------------
#include<stdio.h>
#include<string.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
int main()
{
 // 系统调用还提供了dup*类型的函数，这里我们主要了解dup2
 // int dup2(int oldfd, int newfd);
  int fd = open("./myfile.txt",O_WRONLY | O_CREAT,0644);
  if(fd<0)
  {
    perror("open:");
    return -1;
  }
  dup2(fd,1);
  printf("hello word\n");
  fprintf(stdout,"hello stdout\n");
  fprintf(stderr,"hello stderr\n");
  fflush(stdout);//如果不加上刷新缓冲区会导致myfile.txt没有数据
  close(fd);
}
```

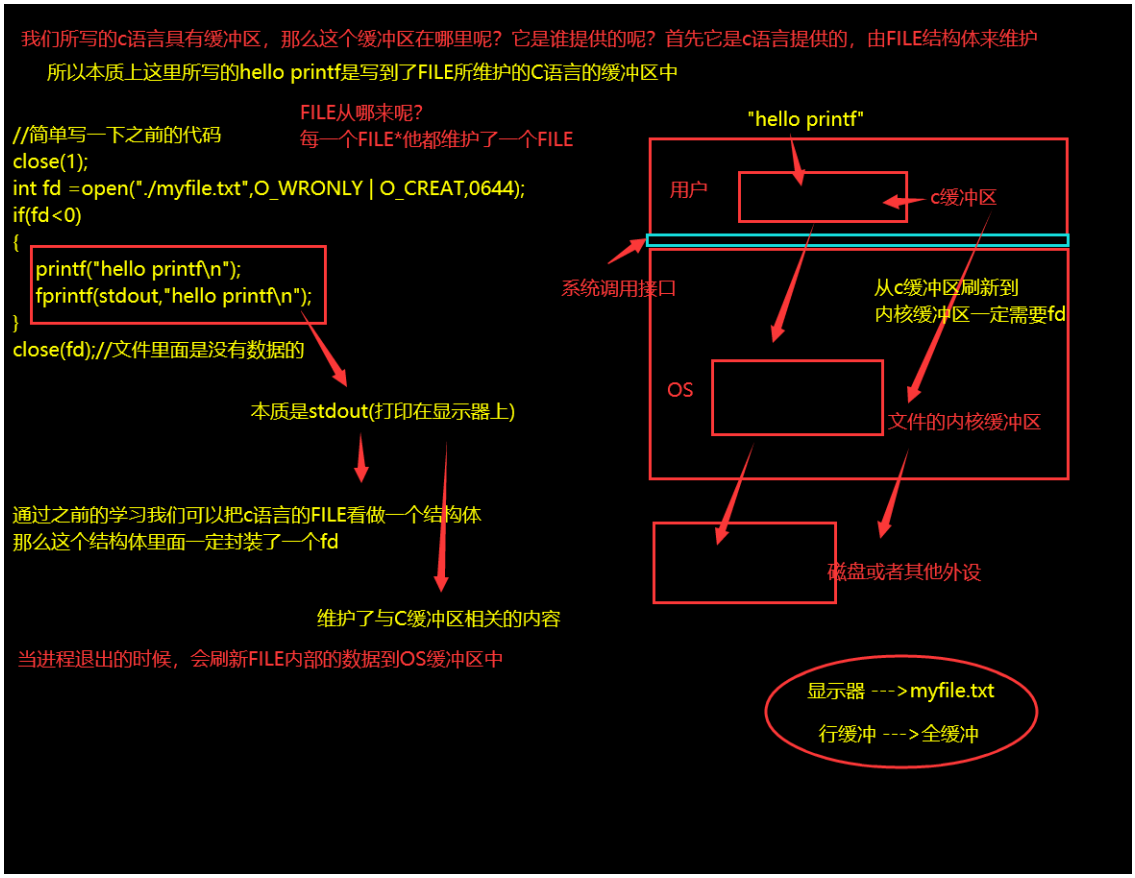
注:

- 如果newfd被打开了，首先会关闭newfd，如果oldfd不是有效的文件描述符则调用失败，并且newfd没有关闭
- 如果oldfd是有效的文件描述符并且newfd的值与oldfd相同dup2()什么都不做并返回newfd
- dup2所复制的文件描述符与原来的文件描述符共享各种文件状态。共享所有的锁定，读写位置和各项权限或flags等.

FILE

- 因为IO相关的函数与系统调用接口对应，并且库函数封装系统调用，所以本质上，访问文件都是通过fd访问
- 所以C库当中的FILE结构体内部也一定封装了fd

在这里解释一下之前为什么不加fflush直接close文件里面没有数据

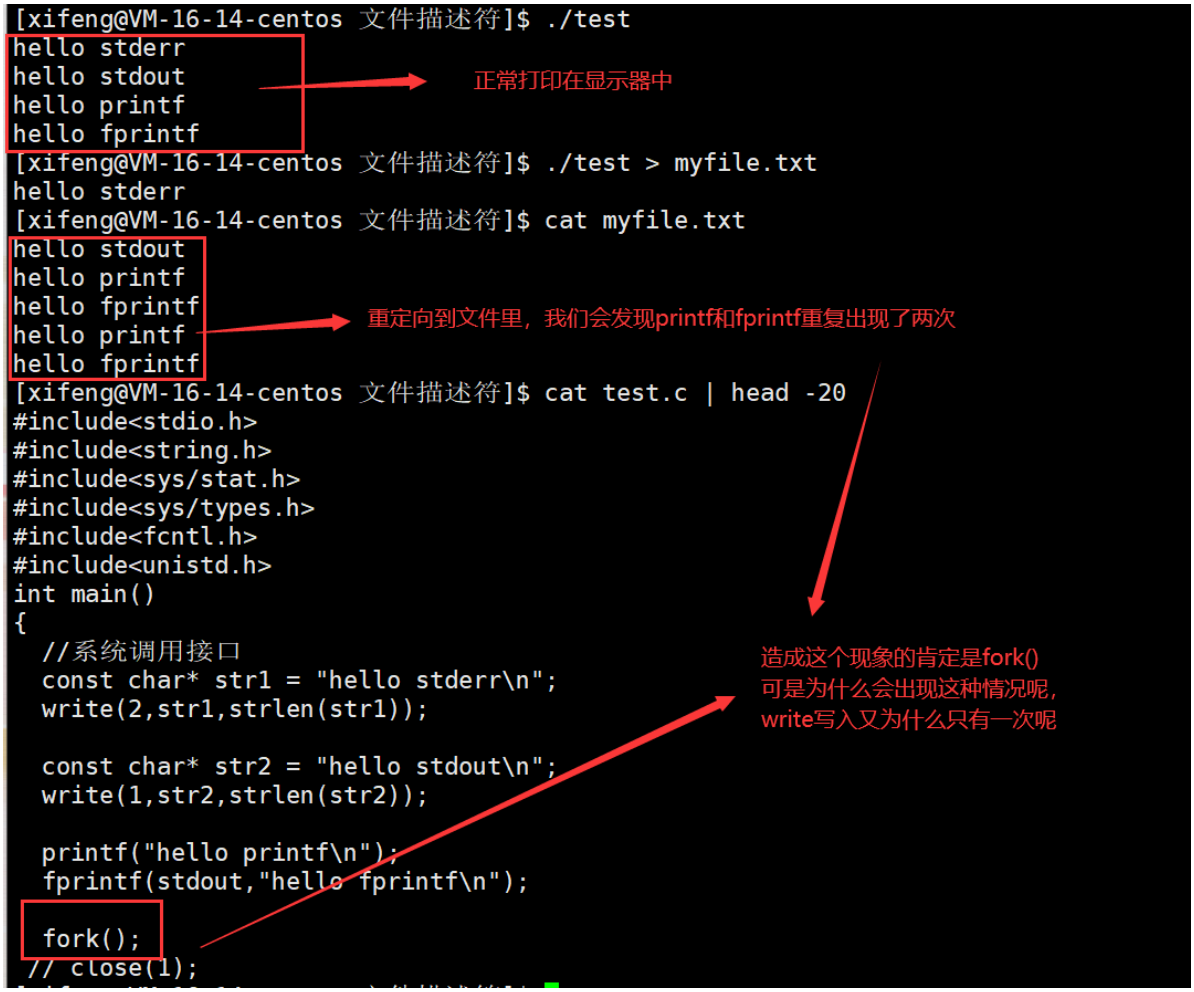
- 首先让我们先了解：用户--->OS的刷新策略
    1. 立即刷新(不缓冲)
    2. 行刷新(行缓冲也就是遇到\n刷新)--->显示器打印就是行刷新
    3. 缓冲区满了才会刷新(全缓冲)，比如往磁盘文件中写入
- 刷新策略OS--->硬件同样适用
- 

- 看完上面是不是就清晰了许多，为什么没有fflush就close掉fd文件里面没有数据呢？----->**首先因为我们进行了重定向，本质上就是将行刷新策略改为了全刷新策略，我们写的hello printf不会被立刻刷新到内核缓冲区，还存在c缓冲区中，其次就是我们在没有刷新缓冲区的情况下把文件给关闭了，这就导致没法刷新到内核缓冲区中，所以会出现文件没有数据的情况；加上fflush之所以可以是因为fflush是强制刷新缓冲区。**
- 像write这样的系统调用是直接写到内核缓冲区当中的

还有这么一个现象

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

```c
#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
int main()
{
  //系统调用接口
  const char* str1 = "hello stderr\n";
  write(2,str1,strlen(str1));

  const char* str2 = "hello stdout\n";
  write(1,str2,strlen(str2));

  printf("hello printf\n");
  fprintf(stdout,"hello fprintf\n");

  fork();
 // close(1);
}
//如果将其重定向到myfile.txt文件中时会出现这种现象
```



- 原因：首先当往显示器上面打印的时候刷新策略是行刷新，所以fork()没有影响，但是当重定向的时候，刷新策略变成了全缓冲，这个缓冲区是c缓冲区，同时也是父进程的缓冲区，父子进程共享代码和数据，当父进程结束或者子进程结束就会刷新缓冲区，谁先刷新缓冲区，谁就发生了写时拷贝，也就是为什么数据出现了两次；write是系统调用不经过用户级缓冲区，(同时也能说明之前的printf和fprintf的缓冲区一定不在OS内部)，所以不会重复打印
- 解决方法就是在fork()之前，使用fflush(stdout)强制刷新即可，立即刷新完之后缓冲区里面就不会存在数据了，这样也就不发生写时拷贝

---

stdout，cin/cout,iostream,fstream：类会包含缓冲区，这也就是为什么会有std::endl;(作用也就是刷新c++缓冲区中的数据到显示器当中)

总结：

- 一般C库函数写入文件时是全缓冲的，而写入显示器是行缓冲
- printf，fprintf 库函数会自带缓冲区，当发生重定向到普通文件时，数据的缓冲方式由行缓冲变成了全缓冲，有时甚至在fork后面刷新
- 而我们放在缓冲区中的数据，就不会被立即刷新但是进程退出之后，会统一刷新，写入文件当中。
- 但是fork的时候，父子数据会发生写时拷贝，所以当你父进程准备刷新的时候，子进程也就有了同样的一份数据，随即产生两份数据。
- write 没有变化，说明没有所谓的缓冲

printf fwrite 库函数会自带缓冲区，而 write 系统调用没有带缓冲区。另外，我们这里所说的缓冲区，都是用户级缓冲区。其实为了提升整机性能，OS也会提供相关内核级缓冲区。那这个用户级缓冲区谁提供呢？ printf fwrite 是库函数， write 是系统调用，库函数在系统调用的“上层”， 是对系统调用的“封装”，但是 write 没有缓冲区，而 printf fwrite 有，足以说明，该缓冲区是二次加上的，又因为是C，所以由C标准库提供