

1010文件描述符

## Linux认为 一切皆文件

什么叫做文件呢？

**站在系统的角度，能够被input读取，或者能够output写出的设备就叫做文件！**

狭义文件：普通的磁盘文件

**广义上的文件：显示器，键盘，网课，声卡等几乎所有的外设都可以被称之为文件**

今天的计划：

1. 复习一下C接口
2. 直接使用系统接口
3. 分析系统接口的细节，引入fd（文件描述符）
4. 周边概念的理解（fd的理解，fd和FILE的关系，fd的分配规则，fd和重定向，缓冲区...）

```

5 int main()
6 {
7     FILE* fp = fopen("log.txt", "w");
8     if(fp == NULL)
9     {
10         perror("fopen");
11         return 1;
12     }
13     //进行文件操作
14     //...
15     fclose(fp);
16     return 0;
17 }

```

这个log.txt会在哪里创建？

当前路径？

当前路径是什么路径？

源代码的路径？

这个是不对的

我们实验发现 --- myfile在哪里被执行，log.txt就会在哪里被创建 --- log.txt是相对路径！

我们可以通过以下方式找到当前进程的工作目录：

```

5 int main()
6 {
7     FILE* fp = fopen("log.txt", "w");
8     if(fp == NULL)
9     {
10         perror("fopen");
11         return 1;
12     }
13     //进行文件操作
14     //...
15     fclose(fp);
16     while(1) sleep(1);
17     return 0;
18 }

```

The screenshot shows a terminal window with the following content:

```

Makefile - yufc [SSH: YfcTencent]
bit > 1010 > Makefile
1 myfile:myfile.c
2 gcc -o $@ $^
3 .PHONY:clean

[yufc@VM-12-12-centos 1010]$ make
gcc -o myfile myfile.c
[yufc@VM-12-12-centos 1010]$ ./myfile

[yufc@VM-12-12-centos 1010]$ ps aux | head -1 && ps aux | grep -v grep | grep myfile
PPID PTID PGID SID TTY TPGID STAT UID TIME COMMAND
23095 29077 29077 23095 pts/6 29077 S+ 1001 0:00 ./myfile

[yufc@VM-12-12-centos 1010]$ ls /proc/29077 -l
total 0
dr-xr-xr-x 2 yufc yufc 0 Jan 26 13:57 attr
-rw-r--r-- 1 yufc yufc 0 Jan 26 13:57 autogroup
-r----- 1 yufc yufc 0 Jan 26 13:57 auxv
-r----- 1 yufc yufc 0 Jan 26 13:57 cgroup
-w----- 1 yufc yufc 0 Jan 26 13:57 clear_refs
-r----- 1 yufc yufc 0 Jan 26 13:54 cmdline
-rw-r--r-- 1 yufc yufc 0 Jan 26 13:57 comm
-rw-r--r-- 1 yufc yufc 0 Jan 26 13:57 coredump_filter
-r----- 1 yufc yufc 0 Jan 26 13:57 cpuset
lrwxrwxrwx 1 yufc yufc 0 Jan 26 13:54 cwd -> /home/yufc/bit/1010
-r----- 1 yufc yufc 0 Jan 26 13:57 environ
lrwxrwxrwx 1 yufc yufc 0 Jan 26 13:54 exe -> /home/yufc/bit/1010/myfile
dr-x----- 2 yufc yufc 0 Jan 26 13:54 fd
dr-x----- 2 yufc yufc 0 Jan 26 13:57 fdinfo
-rw-r--r-- 1 yufc yufc 0 Jan 26 13:57 gid_map
-r----- 1 yufc yufc 0 Jan 26 13:57 io
-r----- 1 yufc yufc 0 Jan 26 13:57 limits
-rw-r--r-- 1 yufc yufc 0 Jan 26 13:57 loginuid
dr-x----- 2 yufc yufc 0 Jan 26 13:57 map_files
-r----- 1 yufc yufc 0 Jan 26 13:57 maps
-rw----- 1 yufc yufc 0 Jan 26 13:57 mem
-r----- 1 yufc yufc 0 Jan 26 13:57 mountinfo
-r----- 1 yufc yufc 0 Jan 26 13:57 mounts
-r----- 1 yufc yufc 0 Jan 26 13:57 mountstats
dr-xr-xr-x 5 yufc yufc 0 Jan 26 13:57 net
dr-x--x--x 2 yufc yufc 0 Jan 26 13:57 ns

```

The red box highlights the `cwd` entry in the `ls /proc/29077 -l` output, which points to `/home/yufc/bit/1010`. A red arrow points to this entry with the text "这个就是当前的工作目录".

```
//进行文件操作
const char *s1 = "hello fwrite\n"; //不要+1, \0是C语言的规定, 文件要遵守吗? 文件只需要保存有效数据!!
fwrite(s1, strlen(s1), 1, fp);
const char *s2 = "hello fprintf\n";
fprintf(fp, "%s", s2);
const char *s3 = "hello fputs\n";
fputs(s3, fp);
```

一些小技巧：

```
• [yufc@VM-12-12-centos 1010]$ ./myfile
• [yufc@VM-12-12-centos 1010]$ cat log.txt
hello fwrite
hello fprintf
hello fputs
把空重定向到log.txt, 相当于清空
• [yufc@VM-12-12-centos 1010]$ > log.txt
• [yufc@VM-12-12-centos 1010]$ cat log.txt
• [yufc@VM-12-12-centos 1010]$ echo "hello world" > log.txt
• [yufc@VM-12-12-centos 1010]$ cat log.txt
hello world
○ [yufc@VM-12-12-centos 1010]$
```

```
//按行读取
//fgets -> C的接口 -> s(string) -> 会自动在字符串结尾添加\0
char line[64];
while(fgets(line, sizeof(line), fp) != NULL)
{
    fprintf(stdout, "%s", line);
}
```

```
make: myfile is up to date.
• [yufc@VM-12-12-centos 1010]$ ./myfile
• [yufc@VM-12-12-centos 1010]$ make
gcc -o myfile myfile.c
• [yufc@VM-12-12-centos 1010]$ ./myfile
hello world
hello world
hello world
hello world
hello world
hello world
○ hello world[yufc@VM-12-12-centos 1010]$
```

既然我们的程序可以用来输出内容  
那是不是我们可以自己写一个cat命令了呢？

```
int main(int argc, char* argv[])
{
    if(argc!=2)
    {
        printf("argv error!\n");
        return 1;
    }
    FILE* fp = fopen(argv[1], "r");
    if(fp == NULL)
    {
        perror("fopen");
        return 2;
    }
    //进行文件操作
    // const char *s1 = "hello fwrite\n"; //不要+1, \0是C语言的规定, 文件要遵守吗? 文件只需要保存有效数据!
    // fwrite(s1, strlen(s1), 1, fp);
    // const char *s2 = "hello fprintf\n";
    // fprintf(fp, "%s", s2);
    // const char *s3 = "hello fputs\n";
    // fputs(s3, fp);

    //按行读取
    //fgets -> C的接口 -> s(string) -> 会自动在字符串结尾添加\0
    char line[64];
    while(fgets(line, sizeof(line), fp) != NULL)
    {
        fprintf(stdout, "%s", line);
    }

    fclose(fp);
    return 0;
}
```

## 打开名为 argv[1]的文件

[illegible]

其实这就是一个简单的 cat 命令

打开三个标准输入输出流：

1. stdin
2. stdout
3. stderr

#### SYNOPSIS

```
#include <stdio.h>
```

```
extern FILE *stdin;  
extern FILE *stdout;  
extern FILE *stderr;
```

## 学习系统调用接口

C库函数 fopen fclose fread fwrite

系统调用 open close read write

OPEN(2)

Linux Programmer's Manual

## 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);
```

### O\_APPEND

The file is opened in append mode. end of the file, as if with lseek(2) tems if more than one process app support appending to a file, so the without a race condition.

### O\_ASYNC

Enable signal-driven I/O: generate fcntl(2)) when input or output becomes available only for terminals, pseudo See fcntl(2) for further details.

### O\_CLOEXEC (Since Linux 2.6.23)

Enable the close-on-exec flag for the program to avoid additional fcntl(2) ally, use of this flag is essential fcntl(2) F\_SETFD operation to sections where one thread opens a file fork(2) plus execve(2).

### O\_CREAT

If the file does not exist it will effective user ID of the process.

这些是选项

这个就是操作系统传递标记位置的方案！

一个概念：

如何给函数传递标志位置

每一个标记为对应一个比特位

实现的总体思路大致如代码所示：

//用int中不重复的一个bit，就可以标识一种状态

```
#define ONE 0x1 //0001
#define TWO 0x2 //0010
#define THREE 0x4 //0100
```

```
void show(int flags)
{
    if(flags & ONE) printf("hello one!\n");
    if(flags & TWO) printf("hello two!\n");
    if(flags & THREE) printf("hello three!\n");
}

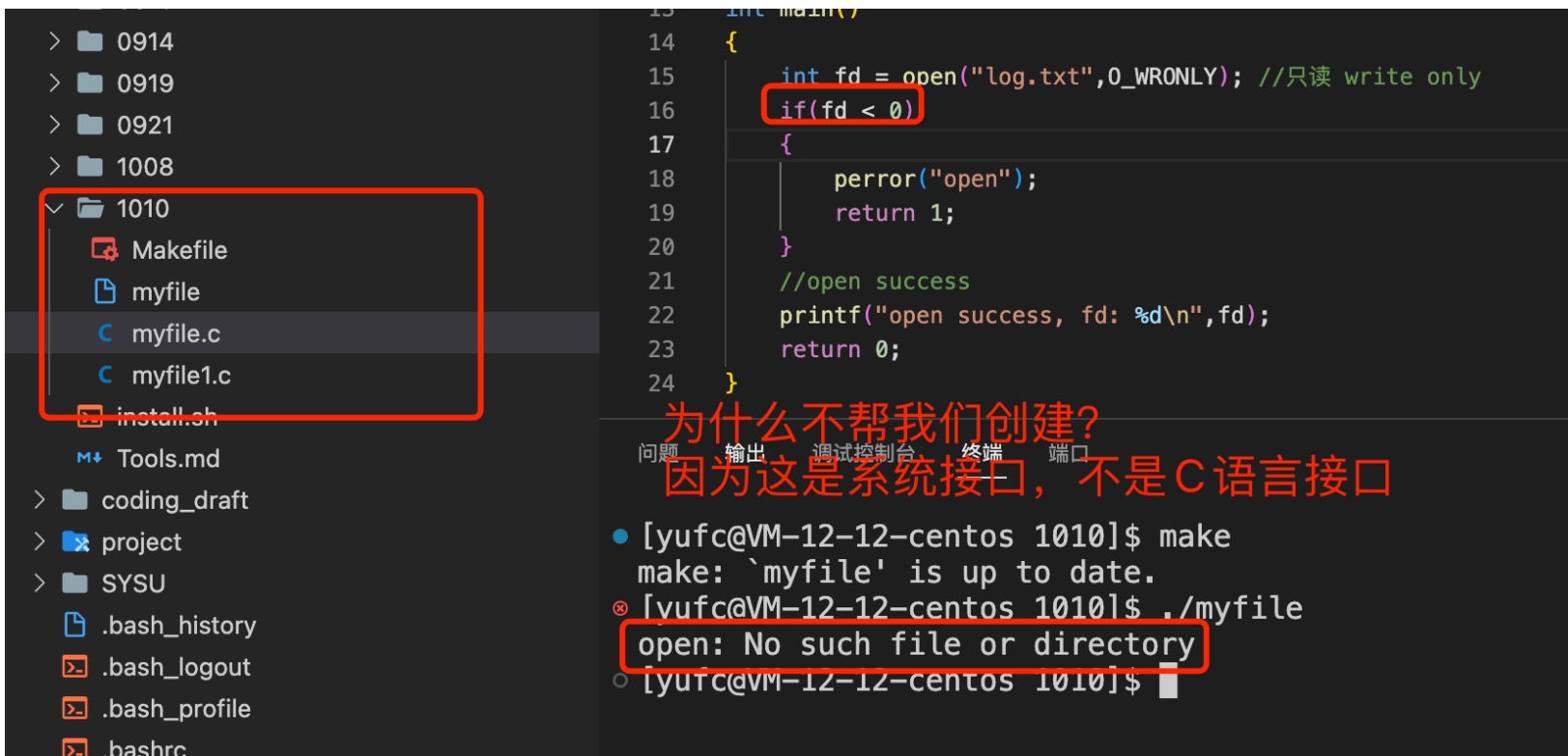
int main()
{
    show(ONE);
    show(TWO);
    show(ONE | TWO);
    show(ONE | TWO | THREE);
    return 0;
}
```

## RETURN VALUE

`open()` and `creat()` return the new file descriptor, or -1 if an error occurred (in which case, `errno` is set appropriately).

文件描述符

如果打开成功 --- 返回文件描述符，如果打开失败，返回-1



```
13 int main()
14 {
15     int fd = open("log.txt", O_WRONLY); //只读 write only
16     if (fd < 0)
17     {
18         perror("open");
19         return 1;
20     }
21     //open success
22     printf("open success, fd: %d\n", fd);
23     return 0;
24 }
```

为什么不帮我们创建？  
因为这是系统接口，不是C语言接口

```
• [yufc@VM-12-12-centos 1010]$ make
make: `myfile' is up to date.
Ⓜ [yufc@VM-12-12-centos 1010]$ ./myfile
open: No such file or directory
○ [yufc@VM-12-12-centos 1010]$
```

这个不是C接口！O\_WRONLY只负责写，如果没有这个文件，是打不开的！



```

12
13 int main()
14 {
15     int fd = open("log.txt", O_WRONLY|O_CREAT); //只读 write only
16     if(fd < 0)
17     {
18         perror("open");
19         return 1;
20     }
21     //open success
22     printf("open success, fd: %d\n", fd);
23     return 0;
24 }

```

问题 输出 调试控制台 终端 端口

```

• [yufc@VM-12-12-centos 1010]$ make
gcc -o myfile myfile.c
• [yufc@VM-12-12-centos 1010]$ ./myfile
open success, fd: 3
• [yufc@VM-12-12-centos 1010]$

```

```

• [yufc@VM-12-12-centos 1010]$ make
gcc -o myfile myfile.c
• [yufc@VM-12-12-centos 1010]$ ./myfile
open success, fd: 3
• [yufc@VM-12-12-centos 1010]$ ll
total 24
-r-srwx--- 1 yufc yufc 0 Jan 26 15:08 log.txt
-rw-rw-r-- 1 yufc yufc 64 Jan 26 14:24 Makefile
-rwxrwxr-x 1 yufc yufc 8504 Jan 26 15:08 myfile
-rw-rw-r-- 1 yufc yufc 1301 Jan 26 14:55 myfile1.c
-rw-rw-r-- 1 yufc yufc 372 Jan 26 15:08 myfile.c
• [yufc@VM-12-12-centos 1010]$

```

我们带上  
O\_CREAT  
就能创建了

但是我们发现，  
创建出来的这个文件的权限怎么是个奇怪的东西呢？

所以，不像我们C接口创建出来的那么整齐

所以，光光创建是不够的！

```

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

```

所以，仅仅读取的时候用上面这个接口  
如果涉及到创建的时候  
我们一般用下面这个接口

第三个参数代表权限！

```

12
13 int main()
14 {
15     int fd = open("log.txt", O_WRONLY|O_CREAT, 0666); //rw-rw-rw-
16     if(fd < 0)
17     {
18         perror("open");
19         return 1;
20     }
21     //open success
22     printf("open success, fd: %d\n", fd);
23     return 0;
24 }

```

问题 输出 调试控制台 终端 端口

```

• [yufc@VM-12-12-centos 1010]$ ./myfile
open success, fd: 3
• [yufc@VM-12-12-centos 1010]$ ll
total 24
---Sr-x--T 1 yufc yufc    0 Jan 26 15:14 log.txt
-rw-rw-r-- 1 yufc yufc   64 Jan 26 14:24 Makefile
-rwxrwxr-x 1 yufc yufc  8504 Jan 26 15:13 myfile
-rw-rw-r-- 1 yufc yufc  1301 Jan 26 14:55 myfile1.c
-rw-rw-r-- 1 yufc yufc   372 Jan 26 15:08 myfile.c
○ [yufc@VM-12-12-centos 1010]$

```

为什么还不对呢

因为系统设置了umask  
把权限过滤掉了

我们可以通过一个系统接口  
更改当前进程的umask

```

13 int main()
14 {
15     umask(0); //把umask设置成0
16     int fd = open("log.txt", O_WRONLY|O_CREAT, 0666); //rw-rw-rw-
17     if(fd < 0)
18     {
19         perror("open");
20         return 1;
21     }
22     //open success
23     printf("open success, fd: %d\n", fd);
24     return 0;
25 }

```

问题 输出 调试控制台 终端 端口

```

• [yufc@VM-12-12-centos 1010]$ make
gcc -o myfile myfile.c
• [yufc@VM-12-12-centos 1010]$ ./myfile
open success, fd: 3
• [yufc@VM-12-12-centos 1010]$ ll
total 24
-rw-rw-rw- 1 yufc yufc    0 Jan 26 15:19 log.txt
-rw-rw-r-- 1 yufc yufc   64 Jan 26 14:24 Makefile
-rwxrwxr-x 1 yufc yufc  8552 Jan 26 15:19 myfile
-rw-rw-r-- 1 yufc yufc  1301 Jan 26 14:55 myfile1.c
-rw-rw-r-- 1 yufc yufc   383 Jan 26 15:19 myfile.c
○ [yufc@VM-12-12-centos 1010]$

```

此时，我们就能正确地创建了一个权限为666的文件了

如果这个文件已经有了

我们就使用两个参数的open就行了  
不需要三个参数的

带上O\_RDONLY选项 --- read only

```
12
13  int main()
14  {
15      umask(0);
16      // int fd = open("log.txt", O_WRONLY|O_CREAT, 0666); //rw-rw-rw-
17      int fd = open("log.txt", O_RDONLY)
18      if (fd < 0)
19      {
20          perror("open");
21          return 1;
22      }
23      //open success
24      printf("open success, fd: %d\n", fd);
25      return 0;
26  }
```

问题 输出 调试控制台 终端 端口

```
• [yufc@VM-12-12-centos 1010]$ make
gcc -o myfile myfile.c
• [yufc@VM-12-12-centos 1010]$ ./myfile
open success, fd: 3
• [yufc@VM-12-12-centos 1010]$
```

既然现在文件已经打开了

那怎么关闭呢？

```
CLOSE(2) Linux Progn

NAME
    close - close a file descriptor

SYNOPSIS
    #include <unistd.h>
    int close(int fd);

DESCRIPTION
```

开关我们搞定了

现在我们要往文件里面写东西

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

```
12
13 int main()
14 {
15     umask(0);
16     int fd = open("log.txt", O_WRONLY | O_CREAT, 0666); //rw-rw-rw-
17     // int fd = open("log.txt", O_RDONLY);
18     if (fd < 0)
19     {
20         perror("open");
21         return 1;
22     }
23     //open success
24     printf("open success, fd: %d\n", fd);
25     //写东西
26     const char* s = "hello write\n";
27     write(fd, s, strlen(s)); //要不要+1 不用!
28
29     //关闭文件
30     close(fd);
31     return 0;
32 }
33
```

问题 输出 调试控制台 终端 端口

```
• [yufc@VM-12-12-centos 1010]$ ./myfile
open success, fd: 3
• [yufc@VM-12-12-centos 1010]$ cat log.txt
hello write
• [yufc@VM-12-12-centos 1010]$
```

此时我们就把字符串写进去了  
此时，我们把代码改一下，字符串改成aa  
再运行一下

问题 输出 调试控制台 终端 端口

```
• [yufc@VM-12-12-centos 1010]$ make
gcc -o myfile myfile.c
• [yufc@VM-12-12-centos 1010]$ ./myfile
open success, fd: 3
• [yufc@VM-12-12-centos 1010]$ cat log.txt
aallo write
• [yufc@VM-12-12-centos 1010]$
```

这里是不会帮你清空的

想要系统帮我们清空  
还要带上一个选项

```
12
13 int main()
14 {
15     umask(0);
16     int fd = open("log.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666); //rw-rw-rw-
17     // int fd = open("log.txt", O_RDONLY);
18     if (fd < 0)
19     {
20         perror("open");
21         return 1;
22     }
23     //open success
24     printf("open success, fd: %d\n", fd);
25     //写东西
26     const char* s = "aa";
27     write(fd, s, strlen(s)); //要不要+1 不用!
28
29
30     //关闭文件
31     close(fd);
32     return 0;
33 }
```

还要带上 O\_TRUNC

所以，我们上层  
fopen带上w选项

其实底层做了这么多事情

那如果我想要往文件中追加呢？

问题 输出 调试控制台 终端 端口

- [yufc@VM-12-12-centos 1010]\$ ./myfile  
open success, fd: 3
- [yufc@VM-12-12-centos 1010]\$ cat log.txt
- aa[yufc@VM-12-12-centos 1010]\$

此时，才会帮我们清空

当我们上层调用a（追加）的时候，下层应该是这个样子的

```
12
13 int main()
14 {
15     umask(0);
16     // int fd = open("log.txt",O_WRONLY|O_CREAT|O_TRUNC,0666); //rw-rw-rw-
17     // int fd = open("log.txt",O_RDONLY);
18     int fd = open("log.txt",O_WRONLY|O_CREAT|O_APPEND,0666);
19     if(fd < 0)
20     {
21         perror("open");
22         return 1;
23     }
24     //open success
25     printf("open success, fd: %d\n",fd);
26     //写东西
27     const char* s = "aa";
28     write(fd,s,strlen(s)); //再不要+1 不用!
```

把 O\_TRUNC 换成 O\_APPEND

问题 输出 调试控制台 终端 端口

```
• [yufc@VM-12-12-centos 1010]$ make
gcc -o myfile myfile.c
• [yufc@VM-12-12-centos 1010]$ ./myfile
open success, fd: 3
• [yufc@VM-12-12-centos 1010]$ ./myfile
open success, fd: 3
• [yufc@VM-12-12-centos 1010]$ ./myfile
open success, fd: 3
• [yufc@VM-12-12-centos 1010]$ ./myfile
open success, fd: 3
○ [yufc@VM-12-12-centos 1010]$
• [yufc@VM-12-12-centos 1010]$ cat log.txt
• aaaaaaaaaa[yufc@VM-12-12-centos 1010]$
```

现在，我们来认识一下 读文件的接口  
read

```
Linux Programmer's Manual

NAME
    read - read from a file descriptor

SYNOPSIS
    #include <unistd.h>

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

DESCRIPTION
```

read  
的返回值我们到进程通信再说  
现在我们先不关心

read是不会给我们加 \0 的

```
myfile1.c  myfile.c  log.txt  ×

bit > 1010 > log.txt
1  hello read
2  hello read
3  |
```

```
12
13 int main()
14 {
15     umask(0);
16     // int fd = open("log.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666); //rw-rw-rw-
17     int fd = open("log.txt", O_RDONLY);
18     // int fd = open("log.txt", O_WRONLY|O_CREAT|O_APPEND, 0666);
19     if(fd < 0)
20     {
21         perror("open");
22         return 1;
23     }
24     //open success
25     printf("open success, fd: %d\n", fd);
26     // //写东西
27     // const char* s = "aa";
28     // write(fd, s, strlen(s)); //要不要+1 不用!
29
30     //读东西
31     char buffer[64];
32     memset(buffer, '\0', sizeof(buffer)); //保证字符串的结尾是\0 read是不会帮我们加的
33     read(fd, buffer, sizeof(buffer));
34
35     printf("%s", buffer);
36 }
```

```
问题  输出  调试控制台  终端  端口

[yufc@VM-12-12-centos 1010]$ make
gcc -o myfile myfile.c
[yufc@VM-12-12-centos 1010]$ ./myfile
open success, fd: 3
hello read
[yufc@VM-12-12-centos 1010]$
```

今天的计划：

1. 复习一下C接口
2. 直接使用系统接口
3. 分析系统接口的细节，引入fd（文件描述符）
4. 周边概念的理解（fd的理解，fd和FILE的关系，fd的分配规则，fd和重定向，缓冲区...）



### 3. 分析系统接口的细节，引入fd(文件描述符)

录制中02:36:27

上面的代码，如何深入理解呢？

```
[whb@VM-0-3-centos lesson19]$ ./myfile
```

```
open success, fd: 3
```

```
open success, fd: 4
```

```
open success, fd: 5
```

```
open success, fd: 6
```

1. 0, 1, 2 去哪里了呢??

2. 为什么是这样的数据?

stdin, stdout, stderr -> FILE \*

0

标准输入

1

标准输出

2

标准错误

现在我们去证明

0, 1, 2就是标准的三个流

```
12
13
14 int main()
15 {
16     fprintf(stdout, "hello stdout\n");
17     const char* s = "hello 1\n";
18
19     write(1, s, strlen(s));
20     return 0;
21 }
```

问题 输出 调试控制台 终端 端口

- [yufc@VM-12-12-centos 1010]\$ make  
gcc -o myfile myfile.c
- [yufc@VM-12-12-centos 1010]\$ ./myfile  
hello stdout  
hello 1
- [yufc@VM-12-12-centos 1010]\$

此时的问题是

FILE\* 和文件描述符的 int 是什么关系？

首先，我们要解决：

FILE\* 中的 FILE 是什么？

FILE是一个结构体！

是谁提供呢？C标准库提供的！

C中文件相关库函数内部一定会调用系统调用！

那么在系统角度，认FILE，还是认fd？

系统只认fd

FILE结构体里面必定封装了fd！

stdin stdout stderr  
内部必定有fd

```
int main()
{
    printf("stdin: %d\n", stdin->_fileno);
    printf("stdout: %d\n", stdout->_fileno);
    printf("stderr: %d\n", stderr->_fileno);
    return 0;
}
```

```
• [yufc@VM-12-12-centos 1010]$ make
gcc -o myfile myfile.c
• [yufc@VM-12-12-centos 1010]$ ./myfile
stdin: 0
stdout: 1
stderr: 2
• [yufc@VM-12-12-centos 1010]$
```

所以，fd 是什么呢？

当进程要访问文件，必须要打开文件！

一个进程可以打开多个文件吗？

当然可以！

一般而言 进程：打开的文件 = 1:n ？

文件要被访问，前提是加载到内存中，才能被直接访问的！

所以当多个进程都打开多个文件

则系统红会存在大量的打开的文件

因此我们的操作系统也要把如此多的文件管理起来！

同样！先描述再组织！

在内核中，OS内部要为了管理每一个被打开的文件，构建struct file{}

```
struct file
{
    struct file* next;
    struct file* prev;
    //包含了一个被打开的文件的几乎所有的内容 (不仅仅包含属性)
}
```

创建struct file的对象，充当一个被打开的文件  
再用双链表组织起来！

所以存在一个数组！

**是一个什么数组呢？ struct file\* array[32]**

所以！通过数组下标 --- 可以找到file结构体的哈希索引！

fd 在内核中，本质是一个数组下标！！！！