1012文件描述符

文件分为两种：

1. 被打开的文件
2. 没有被打开的文件

```
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！！！！

如何证明？
我们可以看源代码！

文件：
磁盘文件（没有被打开）

内存文件（被进程在内存中打开的）

文件描述符的本质是数组下标！！

struct task_struct

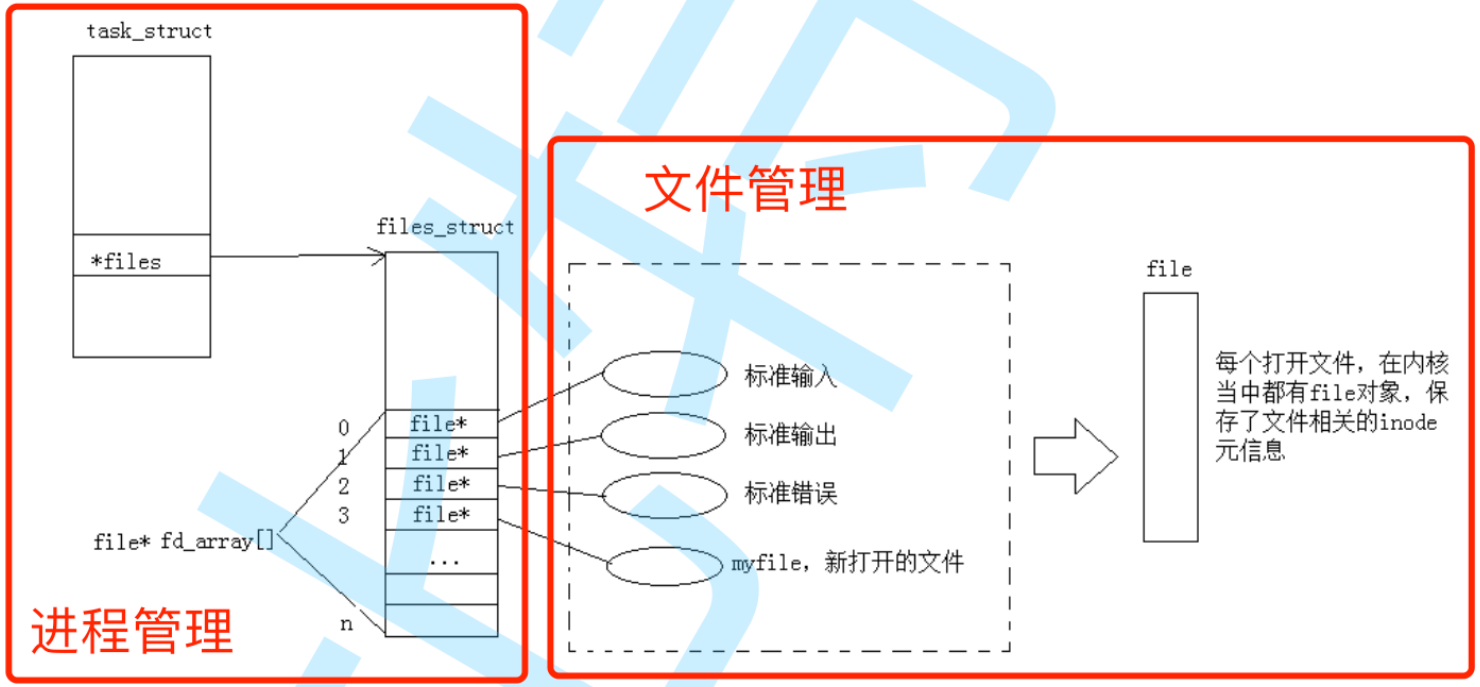
```
/* open file information */
  struct files_struct *files;
```

```
/*
 * Open file table structure
 */
struct files_struct {
  /*
   * read mostly part
   */
    atomic_t count;
    struct fdtable *fdt;
    struct fdtable fdtab;
  /*
   * written part on a separate cache line in SMP
   */
    spinlock_t file_lock ____cacheline_aligned_in_smp;
    int next_fd;
    struct embedded_fd_set close_on_exec_init;
    struct embedded_fd_set open_fds_init;
    struct file * fd_array[NR_OPEN_DEFAULT];
};
```

指针数组

```
struct file {
    /*
     * fu_list becomes invalid after
     * fu_rcuhead for RCU freeing
     */
    union {
```

```
#define BITS_PER_LONG 64
```



现在我们正式开始学习文件描述符

文件描述符fd

> - 通过对open函数的学习，我们知道了文件描述符就是一个小整数

0 & 1 & 2

> - Linux进程默认情况下会有3个缺省打开的文件描述符，分别是标准输入0， 标准输出1， 标准错误2.
> - 0,1,2对应的物理设备一般是：键盘，显示器，显示器
>
> 所以输入输出还可以采用如下方式：

```c
int main()
{
    int fd = open("log.txt",O_WRONLY|O_CREAT|O_TRUNC);
    if(fd < 0)
    {
        perror("open");
        return 1;
    }
    printf("fd: %d\n",fd);
    close(fd);
    return 0;
}
```

```
问题  输出  调试控制台  终端  端口
[yufc@VM-12-12-centos 1012]$ ./myfile
fd: 3
[yufc@VM-12-12-centos 1012]$
```

如果我们直接打印 fd
肯定是3
这个没有疑问 --- 因为0，1，2已经被打开了

如果我们故意关掉0呢？那就是0

```c
int main()
{
    close(0);
    int fd = open("log.txt",O_WRONLY|O_CREAT|O_TRUNC);
    if(fd < 0)
    {
        perror("open");
        return 1;
    }
    printf("fd: %d\n",fd);
    close(fd);
    return 0;
}
```

关掉0

```
问题  输出  调试控制台  终端  端口
[yufc@VM-12-12-centos 1012]$ make
gcc -o myfile myfile.c
[yufc@VM-12-12-centos 1012]$ ./myfile
open: Permission denied
[yufc@VM-12-12-centos 1012]$ sudo ./myfile
[sudo] password for yufc:
fd: 0
[yufc@VM-12-12-centos 1012]$
```

这里 fd 的分配规则是：

最小的，没有被占用的文件描述符！

如果我们把1关掉
为什么就不打印了？
因为：1原本是stdout

那么 --- 我们会得到什么现象呢？

```c
int main()
{
    close(1);
    int fd = open("log.txt",O_WRONLY|O_CREAT|O_TRUNC);
    if(fd < 0)
    {
        perror("open");
        return 1;
    }
    printf("fd: %d\n",fd);
    //close(fd);
    return 0;
}
```

```
问题  输出  调试控制台  终端  端口

[yufc@VM-12-12-centos 1012]$ make
gcc -o myfile myfile.c
[yufc@VM-12-12-centos 1012]$ ./myfile
open: Permission denied
[yufc@VM-12-12-centos 1012]$ sudo ./myfile
[yufc@VM-12-12-centos 1012]$ cat log.txt
fd: 1
[yufc@VM-12-12-centos 1012]$
```

当我们把 close 屏蔽掉之后

我们发现 --- 打印到 log.txt 里面去了

为什么？

如果我们把 close 放开
发现没有打印到log.txt里面 也没有在屏幕上
如果我们加上fflush(stdout)
发现打印到log.txt里面了！

```c
int main()
{
    close(1);
    int fd = open("log.txt",O_WRONLY|O_CREAT|O_TRUNC);
    if(fd < 0)
    {
        perror("open");
        return 1;
    }
    printf("fd: %d\n",fd);
    fflush(stdout);
    close(fd);
    return 0;
}
```

```
问题  输出  调试控制台  终端  端口

[yufc@VM-12-12-centos 1012]$ make
gcc -o myfile myfile.c
[yufc@VM-12-12-centos 1012]$ sudo ./myfile
[yufc@VM-12-12-centos 1012]$ cat log.txt
fd: 1
[yufc@VM-12-12-centos 1012]$
```

从上面的两个现象

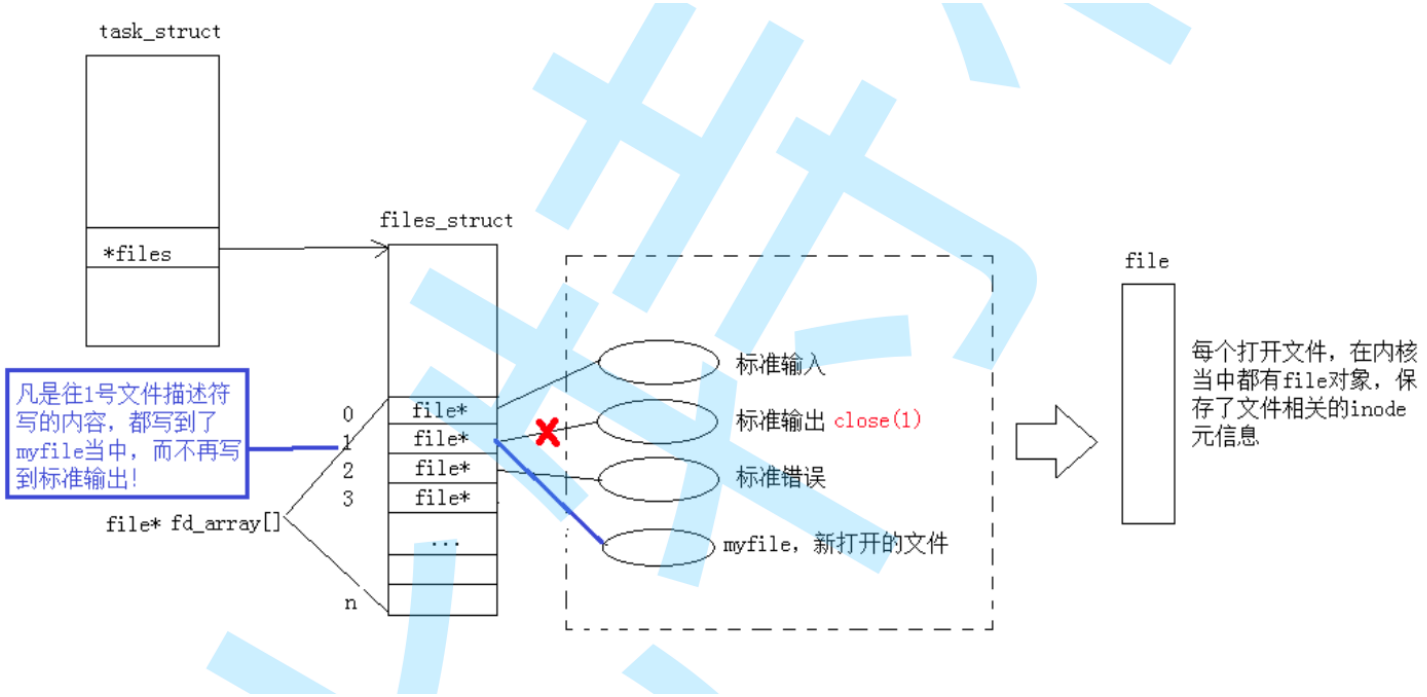
我们可以得到如下结论：

本来往显示器上面的内容
都显示到 log.txt 里面了
这个就是输出重定向！

为什么，因为stdout的file_struct里面封装的fd
就是1
他不认识stdout
他只认识1
现在log.txt是1 --- 所以打印到 log.txt！

按照这个理论，
那如果我把0关掉
那是不是就是可以构建一个输入重
定向呢？

输出重定向的原理：这个图很重要！



```c
28  //输入重定向
29  int main()
30  {
31      char buffer[64];
32      fgets(buffer,sizeof buffer,stdin);
33      printf("%s\n",buffer);
34      return 0;
35  }
```

问题 输出 调试控制台 终端 端口

```
[yufc@VM-12-12-centos 1012]$ make
gcc -o myfile myfile.c
[yufc@VM-12-12-centos 1012]$ ./myfile
```

此时光标卡住了

因为键盘没有输入

```c
29  int main()
30  {
31      close(0);//把stdin删掉
32      int fd = open("log.txt",O_RDONLY);
33      if(fd<0)
34      {
35          perror("open");
36          return 1;
37      }
38      printf("fd: %d\n",fd);//此时就是0了
39      char buffer[64];
40      fgets(buffer,sizeof buffer,stdin);
41      printf("%s\n",buffer);
42
43      close(fd);
44      return 0;
45  }
```

问题 输出 调试控制台 终端 端口

```
[yufc@VM-12-12-centos 1012]$ make
gcc -o myfile myfile.c
[yufc@VM-12-12-centos 1012]$ ./myfile
fd: 0
hello world

[yufc@VM-12-12-centos 1012]$
```

输入重定向！

```c
//追加重定向
int main()
{
    close(1);
    // int fd = open("log.txt",O_WRONLY|O_TRUNC|O_CREAT);
    int fd = open("log.txt",O_WRONLY|O_APPEND|O_CREAT);
    if(fd<0)
    {
        perror("open");
        return 1;
    }
    fprintf(stdout,"you can see me, success\n"); //这里本来是要往显示器打印的
    //现在是要往 log.txt 里面去打印
    return 0;
}
```

把清空选项改成追加就行了！

问题 输出 调试控制台 终端 端口

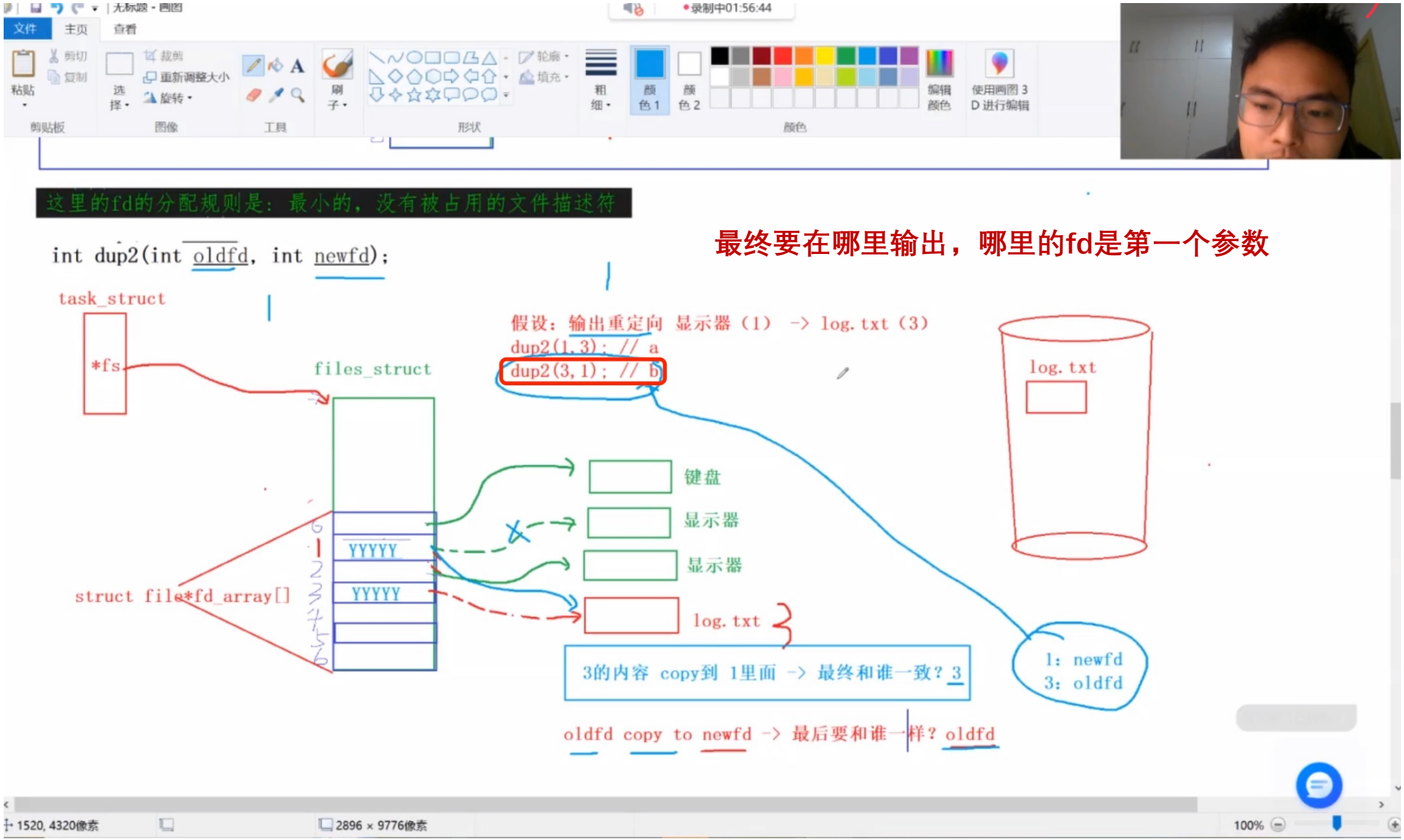
```
yufc@VM-12-12-centos:~/bit/1012$ sudo cat log.txt
you can see me, success
yufc@VM-12-12-centos:~/bit/1012$ ./myfile
open: Permission denied
yufc@VM-12-12-centos:~/bit/1012$ sudo ./myfile
yufc@VM-12-12-centos:~/bit/1012$ sudo cat log.txt
you can see me, success
you can see me, success
yufc@VM-12-12-centos:~/bit/1012$
```

这个就是追加重定向！

当然，重定向不是这样实现的！
我们这种方式仅仅只是利用了文件描述符的特点而已。
有没有一种方式，可以让我们的不用关闭别人的，夜晚冲重定向呢？
肯定是有的！

重定向的系统调用 dup2

```
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:
```


这里的fd的分配规则是：最小的，没有被占用的文件描述符
int dup2(int oldfd, int newfd);
最终要在哪里输出，哪里的fd是第一个参数
task_struct
*fs
files_struct
假设：输出重定向 显示器（1） -> log.txt（3）
dup2(1,3); // a
dup2(3,1); // b
log.txt
键盘
显示器
显示器
log.txt
struct file*fd_array[]
0
1
2
3
4
5
6
YYYYY
YYYYY
3的内容 copy到 1里面 -> 最终和谁一致？3
1：newfd
3：oldfd
oldfd copy to newfd -> 最后要和谁一样？oldfd

```c
//用dup2重定向

//功能
//myfile helloworld   --- 把helloworld写到文件里
int main(int argc, char* argv[])
{
    if(argc!=2)return 2;
    int fd = open("log.txt",O_WRONLY|O_CREAT|O_TRUNC);
    if(fd < 0)
    {
        perror("open");
        return 1;
    }
    //重定向!
    dup2(fd,1);
    printf("%s\n",argv[1]); //stdout -> 1
    return 0;
}
```

```
问题  输出  调试控制台  终端  端口

yufc@VM-12-12-centos:~/bit/1012$ ./myfile helloworld
open: Permission denied
yufc@VM-12-12-centos:~/bit/1012$ sudo ./myfile helloworld
[sudo] password for yufc:
yufc@VM-12-12-centos:~/bit/1012$ cat log.txt
cat: log.txt: Permission denied
yufc@VM-12-12-centos:~/bit/1012$ sudo cat log.txt
helloworld
yufc@VM-12-12-centos:~/bit/1012$
```

这就是输出重定向！

当然，我们把open的选项改一下
就能改成追加重定向了

现在我们要来一些小问题

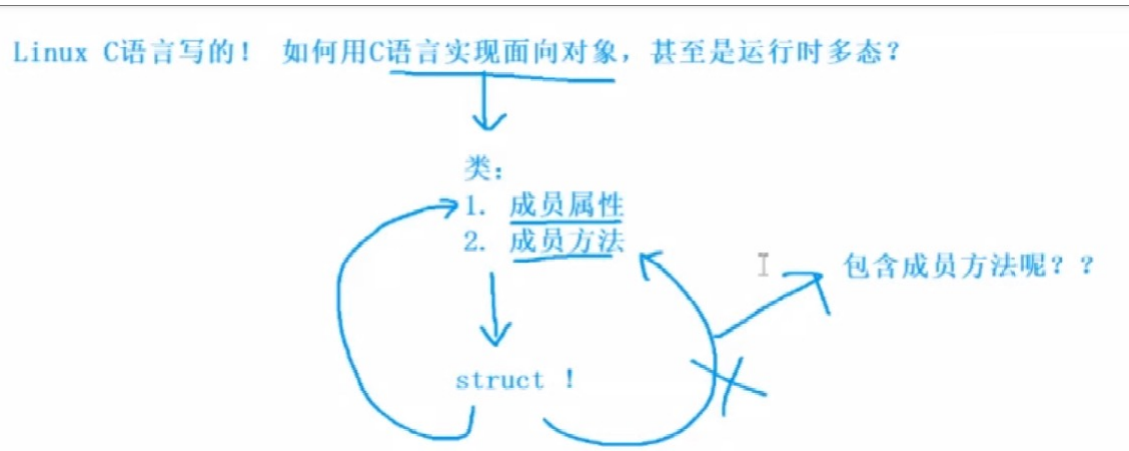
如果把左边这一份代码中加上close(fd);
字符串也可以正常输出到log.txt里面的

至于为什么，我们上面那种先close的重定向方法，最后close之后，就不能成功重定向，而左边代码的方法没问题。
这其实是dup2的一个特性
涉及到缓冲区的概念

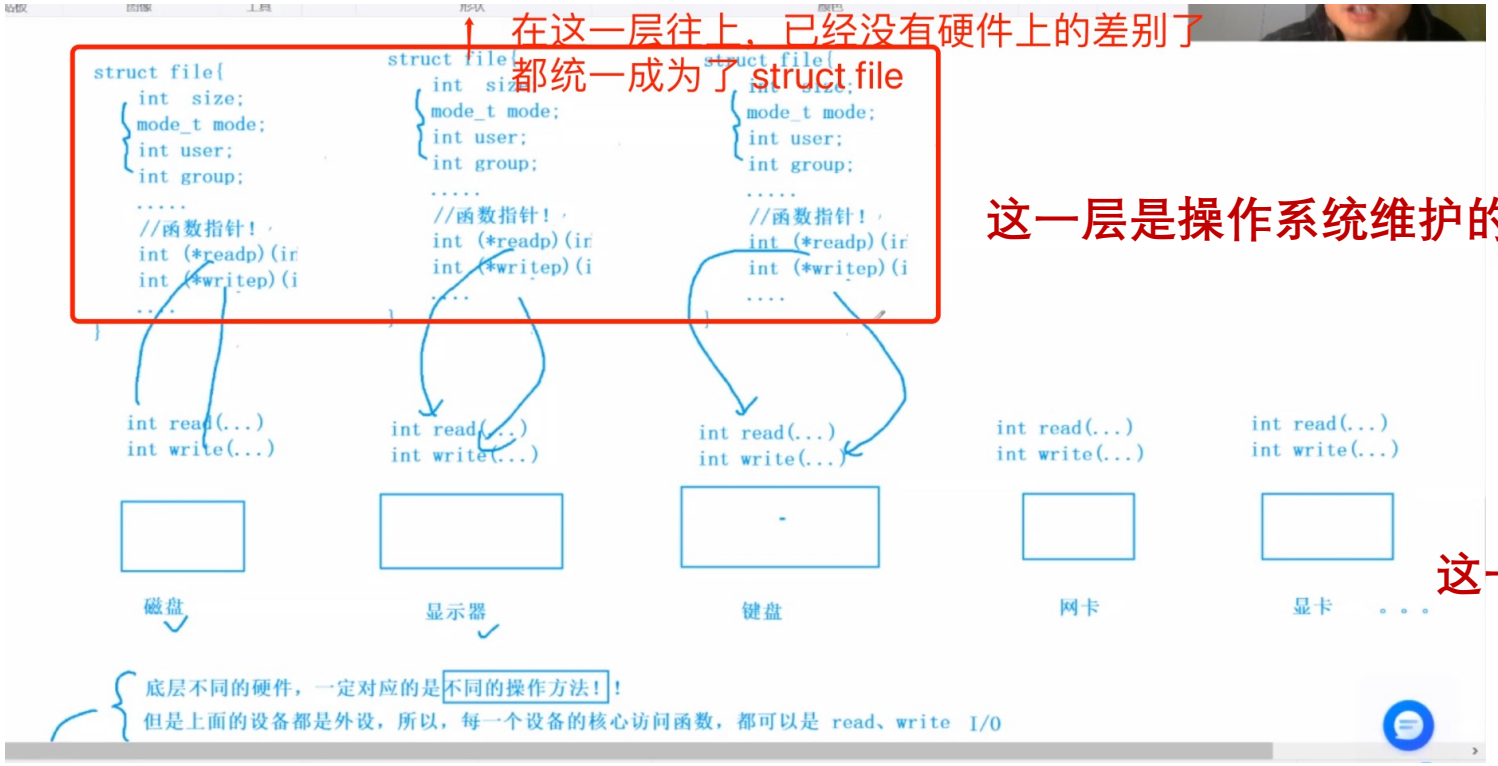
如何理解：一切皆文件

这个是linux的设计哲学 --- 体现在操作系统的软件设计层面的

如何理解呢？我们知道Linux是C语言写的
现在反问一个问题：
如何用C语言设计面向对象，甚至是运行时多态？




```
struct file{
    int  size;
    mode_t mode;
    int user;
    int group;
    .....
    //函数指针！
    int (*readp)(int fd, void*buffer, int len);
    int (*writep)(int fd, void*buffer, int len);
    ....
}
```



访问同一种类型
却可以进行不同的方法
这叫多态！
而linux这种一切皆文件的设计，叫做VFS
virtual File System

缓冲区

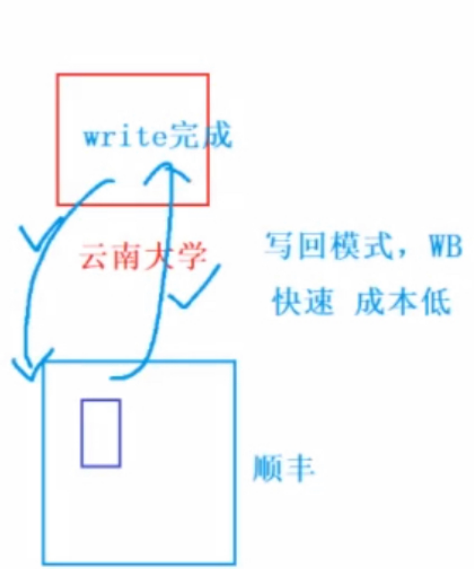
1. 什么是缓冲区，就是一段内存空间（谁提供？用户？语言？OS？）

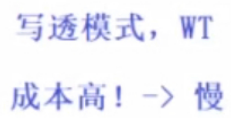
2. 为什么要有缓冲区？

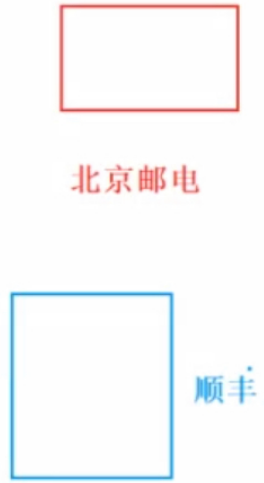
WT模式：写透模式 --- 直接把东西交过去，成本高，慢

WB模式：写回模式 --- 快速，成本低

缓冲区：提高整机效率
主要是为了提高用户的响应速度







缓冲区在哪里？

C？
OS？

这个代码我们都能懂，很简单

```c
//缓冲区在哪?
int main()
{
    //C语言
    printf("hello printf\n");
    fprintf(stdout,"hello fprintf\n");
    const char* s = "hello fputs\n";
    fputs(s,stdout);

    //系统
    const char *ss = "hello write\n";
    write(1,ss,strlen(ss));

    fork(); //创建子进程

    return 0;
}
```

```
问题  输出  调试控制台  终端  端口

yufc@VM-12-12-centos:~/bit/1012$ make
gcc -o myfile myfile.c
yufc@VM-12-12-centos:~/bit/1012$ ./myfile
hello printf
hello fprintf
hello fputs
hello write
yufc@VM-12-12-centos:~/bit/1012$
```

Ok
现在，我们把 ./myfile 的结果重定向一下到 log.txt 里面去

```
yufc@VM-12-12-centos:~/bit/1012$ ./myfile > log.txt
yufc@VM-12-12-centos:~/bit/1012$ cat log.txt
hello write
hello printf
hello fprintf
hello fputs
hello printf
hello fprintf
hello fputs
yufc@VM-12-12-centos:~/bit/1012$
```

为什么是这些话？7条？
如果我们把fork注释掉，是以下结果：

只有4条，为什么？一定和fork有关

```
yufc@VM-12-12-centos:~/bit/1012$ cat log1.txt
hello write
hello printf
hello fprintf
hello fputs
yufc@VM-12-12-centos:~/bit/1012$
```

常见的缓冲区刷新策略：
1. 立即刷新
2. 行刷新（行缓冲）\n \r
3. 满刷新（全缓冲）

特殊情况：
1. 用户强制刷新（fflush）
2. 进程退出

缓冲策略 = 一般 + 特殊

我们先把现象解释一下：

首先
write 只打印了一次
其他的打印了两次

为什么？
我们下节课再讲！
讲清楚之后，我们可以也可以回答一些尚未解答的现象了！