基础IO

本节重点

· 复习C文件IO相关操作

· 认识文件相关系统调用接口

· 认识文件描述符,理解重定向

· 对比fd和FILE，理解系统调用和库函数的关系

· 理解文件和内核文件缓冲区

· 自定义shell新增重定向功能

· 理解Glibc的IO库

1. 理解"文件"

1-1 狭义理解

· 文件在磁盘里

· 磁盘是永久性存储介质，因此文件在磁盘上的存储是永久性的

· 磁盘是外设（即是输出设备也是输入设备）

· 磁盘上的文件 本质是对文件的所有操作，都是对外设的输入和输出 简称 IO

1-2 广义理解

· Linux 下一切皆文件（键盘、显示器、网卡、磁盘…… 这些都是抽象化的过程）（后面会讲如何去理解）

1-3 文件操作的归类认知

· 对于 0KB 的空文件是占用磁盘空间的

· 文件是文件属性（元数据）和文件内容的集合（文件 = 属性（元数据）+ 内容）

· 所有的文件操作本质是文件内容操作和文件属性操作

1-4 系统角度

· 对文件的操作本质是进程对文件的操作

· 磁盘的管理者是操作系统

· 文件的读写本质不是通过 C 语言 / C++ 的库函数来操作的（这些库函数只是为用户提供方便），而是通过文件相关的系统调用接口来实现的

2. 回顾C文件接口

2-1 hello.c打开文件

代码块

```c
#include <stdio.h>
int main()
{
    FILE *fp = fopen("myfile", "w");
    if (!fp)
    {
        printf("fopen error!\n");
    }

    while (1)
        ;

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

打开的myfile文件在哪个路径下?

· 在程序的当前路径下，那系统怎么知道程序的当前路径在哪里呢?

可以使用 ls /proc/[进程id] -l 命令查看当前正在运行进程的信息：

代码块

```
[hyb@VM-8-12-centos io]$ ps ajx | grep myProc
506729 533463 533463 506729 pts/249 533463 R+ 1002 7:45 ./myProc
536281 536542 536541 536281 pts/250 536541 R+ 1002 0:00 grep --
color=auto myProc
[hyb@VM-8-12-centos io]$ ls /proc/533463 -l
total 0
......
-r--r--r-- 1 hyb hyb 0 Aug 26 17:01 cpuset
lrwxrwxrwx 1 hyb hyb 0 Aug 26 16:53 cwd -> /home/hyb/io
-r-------- 1 hyb hyb 0 Aug 26 17:01 environ
```

```
lrwxrwxrwx 1 hyb hyb 0 Aug 26 16:53 exe -> /home/hyb/io/myProc
dr-x------ 2 hyb hyb 0 Aug 26 16:54 fd
......
```

其中：

· cwd：指向当前进程运行目录的一个符号链接。

· exe：指向启动当前进程的可执行文件（完整路径）的符号链接。

打开文件，本质是进程打开，所以，进程知道自己在哪里，即便文件不带路径，进程也知道。由此OS就能知道要创建的文件放在哪里。

3. 系统文件I/O

打开文件的方式不仅仅是fopen，ifstream等流式，语言层的方案，其实系统才是打开文件最底层的方案。不过，在学习系统文件IO之前，先要了解下如何给函数传递标志位，该方法在系统文件IO接口中会使用到:

代码块

```c
#include <stdio.h>
#define ONE 0001   // 0000 0001
#define TWO 0002   // 0000 0010
#define THREE 0004 // 0000 0100

void func(int flags)
{
    if (flags & ONE)
        printf("flags has ONE! ");

    if (flags & TWO)
        printf("flags has TWO! ");

    if (flags & THREE)
        printf("flags has THREE! ");

    printf("\n");
}

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

```
27  }
```

操作文件，除了上小节的C接口（当然，C++也有接口，其他语言也有），我们还可以采用系统接口来进行文件访问， 先来直接以系统代码的形式，实现和上面一模一样的代码。

3-2 hello.c 写文件:

代码块

```
1   #include <stdio.h>
2   #include <sys/types.h>
3   #include <sys/stat.h>
4   #include <fcntl.h>
5   #include <unistd.h>
6   #include <string.h>
7   int main()
8   {
9       umask(0);
10      int fd = open("myfile", O_WRONLY | O_CREAT, 0644);
11      if (fd < 0)
12      {
13          perror("open");
14          return 1;
15      }
16
17      int count = 5;
18      const char *msg = "hello bit!\n";
19      int len = strlen(msg);
20
21      while (count--)
22      {
23          write(fd, msg, len); // fd: 后面讲， msg: 缓冲区首地址， len: 本次读取，期望
24                               // 写入多少个字节的数据。 返回值：实际写了多少字节数据
25      }
26
27      close(fd);
28      return 0;
29  }
```

3-3 hello.c读文件

代码块

```
1   #include <stdio.h>
2   #include <sys/types.h>
```

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main()
{
    int fd = open("myfile", O_RDONLY);
    if (fd < 0)
    {
        perror("open");
        return 1;
    }

    const char *msg = "hello bit!\n";
    char buf[1024];

    while (1)
    {
        ssize_t s = read(fd, buf, strlen(msg)); // 类比write
        if (s > 0)
        {
            printf("%s", buf);
        }
        else
        {
            break;
        }
    }

    close(fd);
    return 0;
}
```

3-4 接口介绍

open `man open`

代码块

```c
#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);
pathname: 要打开或创建的目标文件
```

```
flags：打开文件时，可以传入多个参数选项，用下面的一个或者多个常量进行“或”运算，构成flags。

参数：
    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

返回值：
    成功：新打开的文件描述符
    失败：-1
```

mode_t理解：直接 man 手册，比什么都清楚。

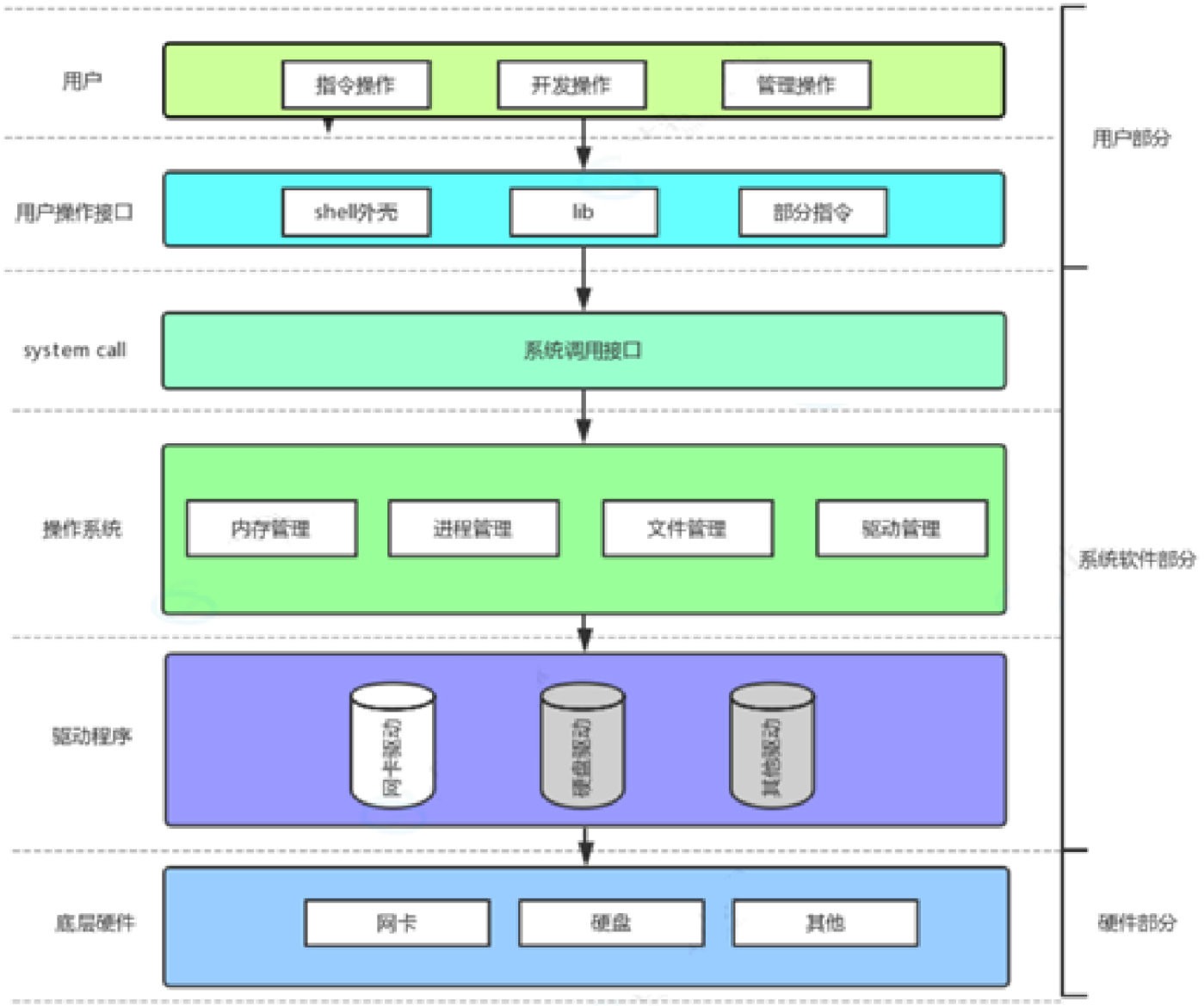
open 函数具体使用哪个，和具体应用场景相关，如目标文件不存在，需要open创建，则第三个参数表示创建文件的默认权限，否则，使用两个参数的open。

write read close lseek ,类比C文件相关接口。

3-5 open函数返回值

在认识返回值之前，先来认识一下两个概念: 系统调用 和 库函数

- 上面的 fopen fclose fread fwrite 都是C标准库当中的函数，我们称之为库函数（libc）。
- 而 open close read write lseek 都属于系统提供的接口，称之为系统调用接口
- 回忆一下我们讲操作系统概念时，画的一张图



系统调用接口和库函数的关系，一目了然。

所以，可以认为， f# 系列的函数，都是对系统调用的封装，方便二次开发。

3-6 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

3-6-1 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

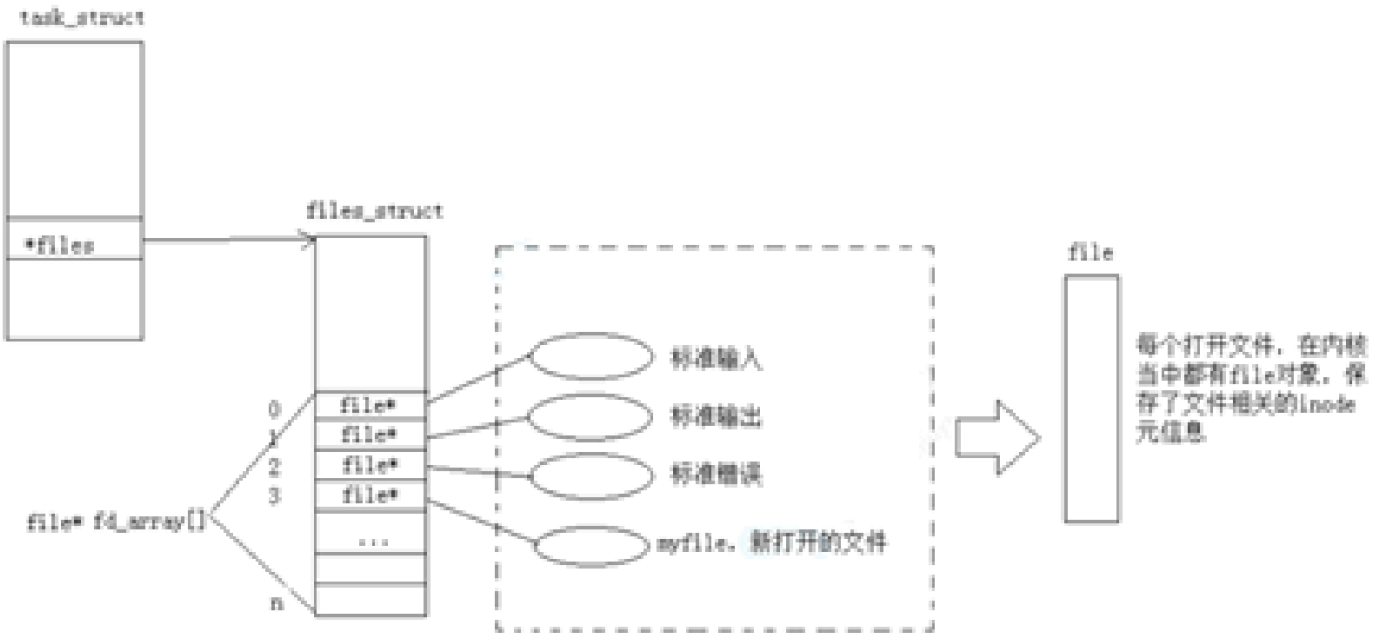
所以输入输出还可以采用如下方式：

代码块

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

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

int main()
{
    char buf[1024];
    ssize_t s = read(0, buf, sizeof(buf));
    if (s > 0)
    {
        buf[s] = 0;
        write(1, buf, strlen(buf));
        write(2, buf, strlen(buf));
    }
    return 0;
}
```



而现在知道，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。于是就有了file结构体。表示一个已经打开的文件对象。而进程执行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含一个指针数组，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。

3-6-2 文件描述符的分配规则

直接看代码：

```c
#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;
}
```

输出发现是 fd: 3

关闭0或者2，再看

代码块

```c
#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;
}
```

发现是结果是： fd: 0 或者 fd 2 ，可见，文件描述符的分配规则：在files_struct数组当中，找到当前没有被使用的最小的一个下标，作为新的文件描述符。

3-6-3 重定向

那如果关闭1呢？看代码：

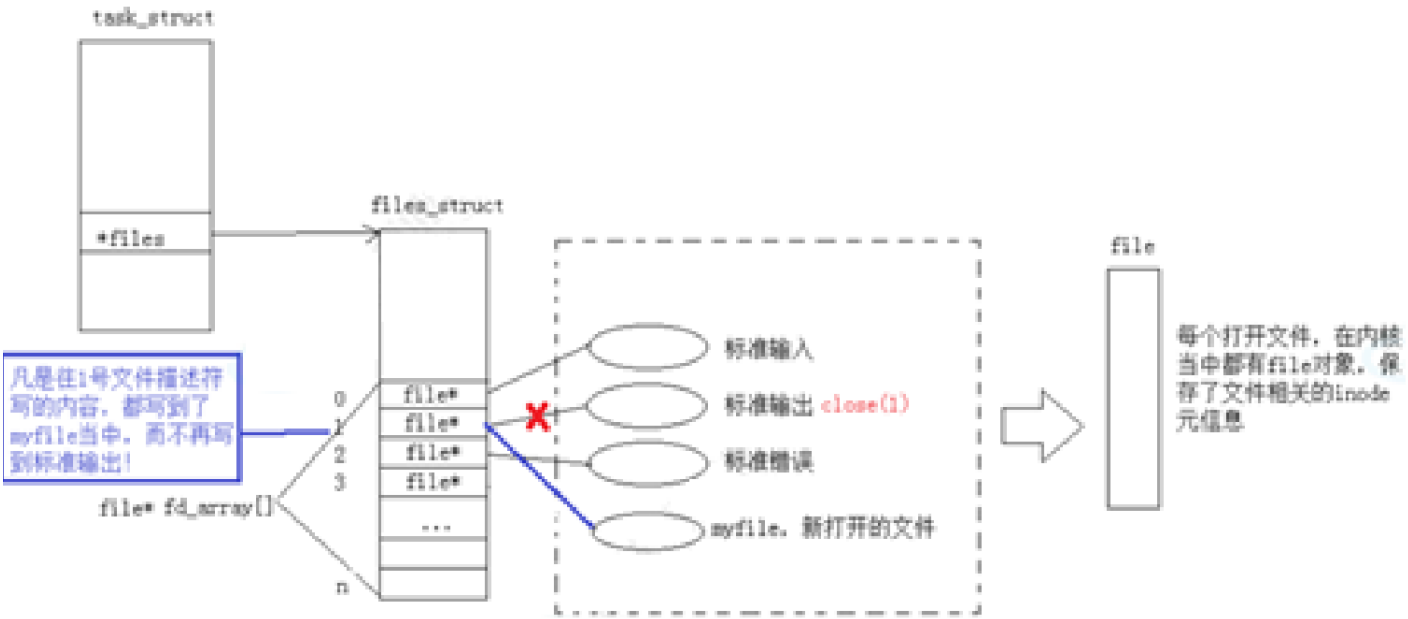
代码块

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
int main()
{
    close(1);
    int fd = open("myfile", O_WRONLY | O_CREAT, 00644);
    if (fd < 0)
    {
        perror("open");
        return 1;
    }
    printf("fd: %d\n", fd);
    fflush(stdout);

    close(fd);
    exit(0);
}
```

此时，我们发现，本来应该输出到显示器上的内容，输出到了文件 myfile 当中，其中，fd＝1。这种现象叫做输出重定向。常见的重定向有: > , >> , <

那重定向的本质是什么呢?



3-6-4 使用 dup2 系统调用

函数原型如下:

代码块

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

示例代码

代码块

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

int main()
{
    int fd = open("./log", O_CREAT | O_RDWR);

    if (fd < 0)
    {
        perror("open");
        return 1;
    }

    close(1);
    dup2(fd, 1);

    for (;;)
    {
        char buf[1024] = {0};
        ssize_t read_size = read(0, buf, sizeof(buf) - 1);

        if (read_size < 0)
        {
            perror("read");
            break;
        }

        printf("%s", buf);
        fflush(stdout);
    }

    return 0;
}
```

printf是C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myfifile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。那追加和输入重定向如何完成呢?

4. 理解“一切皆文件”

首先，在windows中是文件的东西，它们在linux中也是文件；其次一些在windows中不是文件的东西，比如进程、磁盘、显示器、键盘这样硬件设备也被抽象成了文件，你可以使用访问文件的方法访问它们获得信息；甚至管道，也是文件；将来我们要学习网络编程中的socket（套接字）这样的东西，使用的接口跟文件接口也是一致的。

这样做最明显的好处是，开发者仅需要使用一套 API 和开发工具，即可调取 Linux 系统中绝大部分的资源。举个简单的例子，Linux 中几乎所有读（读文件，读系统状态，读PIPE）的操作都可以用 read 函数来进行；几乎所有更改（更改文件，更改系统参数，写 PIPE）的操作都可以用 write 函数来进行。

之前我们讲过，当打开一个文件时，操作系统为了管理所打开的文件，都会为这个文件创建一个file结构体，该结构体定义在

/usr/src/kernels/3.10.0-1160.71.1.el7.x86_64/include/linux/fs.h 下，以下展示了该结构部分我们关系的内容：

代码块

```
struct file
{
    ... 1 2 3 4 struct inode *f_inode; /* cached value */
    const struct file_operations *f_op;
    ... atomic_long_t f_count;                          // 表示打开文件的引用计数，如果有多个文件指针指向它，就会增加f_count的值。
    unsigned int f_flags; // 表示打开文件的权限
    fmode_t f_mode;                                     // 设置对文件的访问模式,例如: 只读，只写等。所有的标志在头文件<fcntl.h> 中定义
    loff_t f_pos; // 表示当前读写文件的位置
    ...
} __attribute__((aligned(4))); /* lest something weird decides that 2 is OK
                                  */
```

值得关注的是 struct file 中的 f_op 指针指向了一个 file_operations 结构体，这个结构体中的成员除了 struct module* owner 其余都是函数指针。该结构和 struct file 都在fs.h下。

代码块

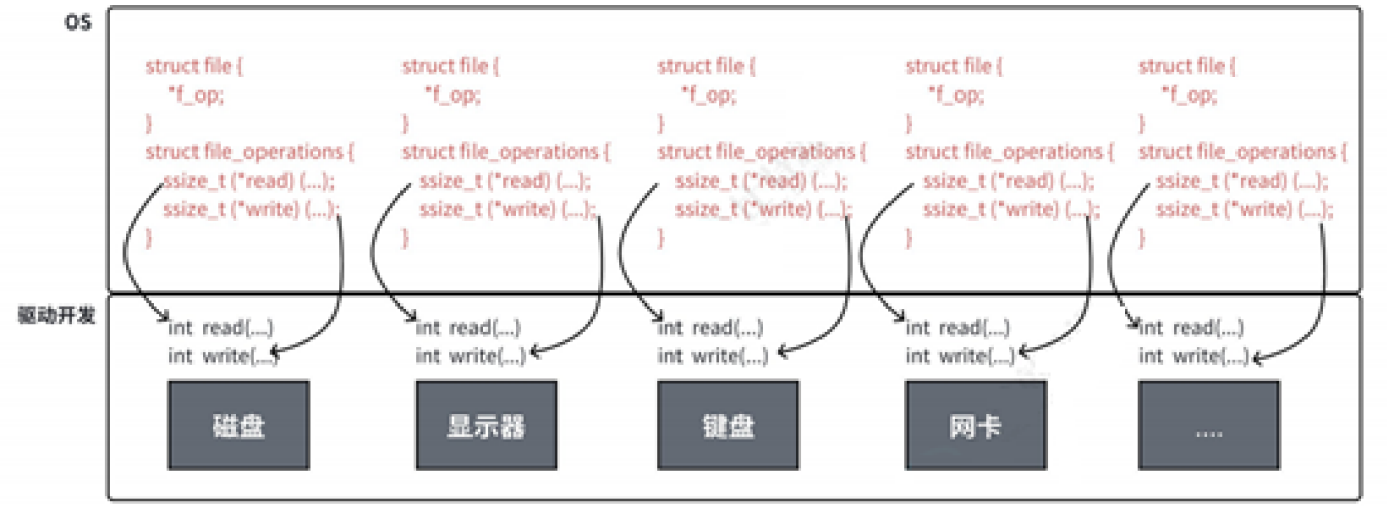
```
struct file_operations {
```

```
2  struct module *owner;
3  //指向拥有该模块的指针;
4  loff_t (*llseek) (struct file *, loff_t, int);
5  //llseek 方法用作改变文件中的当前读/写位置, 并且新位置作为(正的)返回值.
6  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
7  //用来从设备中获取数据. 在这个位置的一个空指针导致 read 系统调用以 -EINVAL("Invalid
   argument") 失败. 一个非负返回值代表了成功读取的字节数( 返回值是一个"signed size" 类
   型, 常常是目标平台本地的整数类型).
8  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
9  //发送数据给设备. 如果 NULL, -EINVAL 返回给调用 write 系统调用的程序. 如果非负,返回值
   代表成功写的字节数.
10 ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long,
11 loff_t);
12 //初始化一个异步读 -- 可能在函数返回前不结束的读操作.
13 ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned
14 long, loff_t);
15 //初始化设备上的一个异步写.
16 int (*readdir) (struct file *, void *, filldir_t);
17 //对于设备文件这个成员应当为 NULL; 它用来读取目录, 并且仅对**文件系统**有用.
18 unsigned int (*poll) (struct file *, struct poll_table_struct *);
19 int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
20 long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
21 long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
22 int (*mmap) (struct file *, struct vm_area_struct *);
23 //mmap 用来请求将设备内存映射到进程的地址空间. 如果这个方法是 NULL, mmap 系统调用返回 -
   ENODEV.
24 int (*open) (struct inode *, struct file *);
25 //打开一个文件
26 int (*flush) (struct file *, fl_owner_t id);
27 //flush 操作在进程关闭它的设备文件描述符的拷贝时调用;
28 int (*release) (struct inode *, struct file *);
29 //在文件结构被释放时引用这个操作. 如同 open, release 可以为 NULL.
30 int (*fsync) (struct file *, struct dentry *, int datasync);
31 //用户调用来刷新任何挂着的数据.
32 int (*aio_fsync) (struct kiocb *, int datasync);
33 int (*fasync) (int, struct file *, int);
34 int (*lock) (struct file *, int, struct file_lock *);
35 //lock 方法用来实现文件加锁; 加锁对常规文件是必不可少的特性, 但是设备驱动几乎从不实现它.
36 ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *,
37 int);
38 unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned
39 long, unsigned long, unsigned long);
40 int (*check_flags)(int);
41 int (*flock) (struct file *, int, struct file_lock *);
42 ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t
43 *, size_t, unsigned int);
44 ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
```

```
45    size_t, unsigned int);
46    int (*setlease)(struct file *, long, struct file_lock **);
47    };
```

file_operation 就是把系统调用和驱动程序关联起来的关键数据结构，这个结构的每一个成员都 对应着一个系统调用。读取 file_operation 中相应的函数指针，接着把控制权转交给函数，从而完成了Linux设备驱动程序的工作。

介绍完相关代码，一张图总结：



上图中的外设，每个设备都可以有自己的read、write，但一定是对应着不同的操作方法！！但通过 `struct file` 下 `file_operation` 中的各种函数回调，让我们开发者只用file便可调取 Linux 系 统中绝大部分的资源！！这便是“linux下一切皆文件”的核心理解。

5. 缓冲区

5-1 什么是缓冲区

缓冲区是内存空间的一部分。也就是说，在内存空间中预留了一定的存储空间，这些存储空间用来缓冲输入或输出的数据，这部分预留的空间就叫做缓冲区。缓冲区根据其对应的是输入设备还是输出设备，分为输入缓冲区和输出缓冲区。

5-2 为什么要引入缓冲区机制

读写文件时，如果不会开辟对文件操作的缓冲区，直接通过系统调用对磁盘进行操作(读、写等)，那么每次对文件进行一次读写操作时，都需要使用读写系统调用来处理此操作，即需要执行一次系统调用，执行一次系统调用将涉及到CPU状态的切换，即从用户空间切换到内核空间，实现进程上下文的切换，这将损耗一定的CPU时间，频繁的磁盘访问对程序的执行效率造成很大的影响。

为了减少使用系统调用的次数，提高效率，我们就可以采用缓冲机制。比如我们从磁盘里取信息，可以在磁盘文件进行操作时，可以一次从文件中读出大量的数据到缓冲区中，以后对这部分的访问就不

需要再使用系统调用了，等缓冲区的数据取完后再去磁盘中读取，这样就可以**减少磁盘的读写次数，再加上计算机对缓冲区的操作大 快于对磁盘的操作，故应用缓冲区可大 提高计算机的运行速度。**

又比如，我们使用打印机打印文档，由于打印机的打印速度相对较慢，我们先把文档输出到打印机相应的缓冲区，打印机再自行逐步打印，这时我们的CPU可以处理别的事情。可以看出，缓冲区就是一块内存区，它用在输入输出设备和CPU之间，用来缓存数据。它使得低速的输入输出设备和高速的CPU能够协调工作，避免低速的输入输出设备占用CPU，解放出CPU，使其能够高效率工作。

5-3 缓冲类型

标准I/O提供了3种类型的缓冲区。

> · 全缓冲区：这种缓冲方式要求填满整个缓冲区后才进行I/O系统调用操作。对于磁盘文件的操作通常使用全缓冲的方式访问。
>
> · 行缓冲区：在行缓冲情况下，当在输入和输出中遇到换行符时，标准I/O库函数将会执行系统调用操作。当所操作的流涉及一个终端时（例如标准输入和标准输出），使用行缓冲方式。因为标准I/O库每行的缓冲区长度是固定的，所以只要填满了缓冲区，即使还没有遇到换行符，也会执行I/O系统调用操作，默认行缓冲区的大小为1024。
>
> · 无缓冲区：无缓冲区是指标准I/O库不对字符进行缓存，直接调用系统调用。标准出错流stderr通常是不带缓冲区的，这使得出错信息能够尽快地显示出来。

除了上述列举的默认刷新方式，下列特殊情况也会引发缓冲区的刷新：

> 1. 缓冲区满时；
> 2. 执行flush语句；
> 3. 进程结束

代码块

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

int main()
{
    close(1);
    int fd = open("log.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (fd < 0)
    {
        perror("open");
        return 0;
    }
```

```
17
18      printf("hello world: %d\n", fd);
19      close(fd);
20      return 0;
21  }
22
```

我们本来想使用重定向思维，让本应该打印在显示器上的内容写到“log.txt”文件中，但我们发现，程序运行结束后，文件中并没有被写入内容：

代码块

```
1  [hyb@VM-8-12-centos buffer]$ ./myfile
2  [hyb@VM-8-12-centos buffer]$ ls
3  log.txt makefile myfile myfile.c
4  [hyb@VM-8-12-centos buffer]$ cat log.txt
5  [hyb@VM-8-12-centos buffer]$
```

这是由于我们将1号描述符重定向到磁盘文件后，缓冲区的刷新方式成为了全缓冲。而我们写入的内容并没有填满整个缓冲区，导致并不会将缓冲区的内容刷新到磁盘文件中。怎么办呢？可以使用fflush强制刷新下缓冲区。

代码块

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <fcntl.h>
6  #include <unistd.h>
7
8  int main()
9  {
10     close(1);
11     int fd = open("log.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);
12     if (fd < 0)
13     {
14         perror("open");
15         return 0;
16     }
17
18     printf("hello world: %d\n", fd);
19     fflush(stdout);
20     close(fd);
21     return 0;
```

```
22  }
23
```

还有一种解决方法，刚好可以验证一下stderr是不带缓冲区的，代码如下：

代码块

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

int main()
{
    close(2);
    int fd = open("log.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (fd < 0)
    {

        perror("open");
        return 0;
    }

    perror("hello world");
    close(fd);
    return 0;
}
```

这种方式便可以将2号文件描述符重定向至文件，由于stderr没有缓冲区，“hello world”不用fflash就可以写入文件：

代码块

```
[hyb@VM-8-12-centos buffer]$ ./myfile
[hyb@VM-8-12-centos buffer]$ cat log.txt
hello world: Success
```

5-4 FILE

· 因为IO相关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访问的。

· 所以C库当中的FILE结构体内部，必定封装了fd。

来段代码在研究一下：

代码块

```c
#include <stdio.h>
#include <string.h>
int main()
{
    const char *msg0 = "hello printf\n";
    const char *msg1 = "hello fwrite\n";
    const char *msg2 = "hello write\n";

    printf("%s", msg0);
    fwrite(msg1, strlen(msg1), 1, stdout);
    write(1, msg2, strlen(msg2));

    fork();

    return 0;
}
```

运行出结果：

代码块

```
hello printf
hello fwrite
hello write
```

但如果对进程实现输出重定向呢？ `./hello > file` ， 我们发现结果变成了：

代码块

```
hello write
hello printf
hello fwrite
hello printf
hello fwrite
```

我们发现 printf 和 fwrite （库函数）都输出了2次，而 write 只输出了一次（系统调用）。为什么呢？肯定和fork有关！

> · 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 fwrite 库函数+会自带缓冲区（进度条例子就可以说明），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。
>
> · 而我们放在缓冲区中的数据，就不会被立即刷新，甚至fork之后
>
> · 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。

我来总结一下：

对于只输出到显示器上，因为输出到显示器是行刷新，所以代码的输出结果按行输出，所以到fork前，所有输出已经执行完成，fork出的子进程继承的缓冲区是空的。

对于重定向到指定文件，因为这时不是输出到显示器上了，行刷新变成全刷新了，所以库函数执行的输出会存到用户缓冲区中。当执行到fork时，子进程这时就能够看到之前输出的结果，所以在输出到文件时库函数会刷新子进程与父进程的缓冲区，也就是输出两次。

但是为什么系统调用不会输出两次呢？其实也就是fork出的子进程不会继承内核缓冲区，导致子进程看到的缓冲区是空的，系统调用输出也只会刷新父进程的内核缓冲中的内容，所以只输出一次。

如果fork前都是输出到文件呢？这是结果不变，原理与上面类似，直接就是全缓冲。

综上： 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，不过不在我们讨论范围之内。

那这个缓冲区谁提供呢？ 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。