5. 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

```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：

```
[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。

2-2 hello.c写文件

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

int main()
{
    FILE *fp = fopen("myfile", "w");
    if(!fp){
        printf("fopen error!\n");
    }

    const char *msg = "hello bit!\n";
    int count = 5;
    while(count--){
        fwrite(msg, strlen(msg), 1, fp);
    }

    fclose(fp);

    return 0;
}
```

2-3 hello.c读文件

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

int main()
{
    FILE *fp = fopen("myfile", "r");
    if(!fp){
        printf("fopen error!\n");
        return 1;
    }

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

```

```
    while(1){
        //注意返回值和参数，此处有坑，仔细查看man手册关于该函数的说明
        ssize_t s = fread(buf, 1, strlen(msg), fp);
        if(s > 0){
            buf[s] = 0;
            printf("%s", buf);
        }
        if(feof(fp)){
            break;
        }
    }

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

稍作修改，实现简单cat命令:

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

int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        printf("argv error!\n");
        return 1;
    }
    FILE *fp = fopen(argv[1], "r");
    if(!fp){
        printf("fopen error!\n");
        return 2;
    }
    char buf[1024];
    while(1){
        int s = fread(buf, 1, sizeof(buf), fp);
        if(s > 0){
            buf[s] = 0;
            printf("%s", buf);
        }
        if(feof(fp)){
            break;
        }
    }

```

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

2-4 输出信息到显示器，你有哪些方法

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

int main()
{
    const char *msg = "hello fwrite\n";
    fwrite(msg, strlen(msg), 1, stdout);

    printf("hello printf\n");
    fprintf(stdout, "hello fprintf\n");
    return 0;
}
```

2-5 stdin & stdout & stderr

- C默认会打开三个输入输出流，分别是stdin, stdout, stderr
- 仔细观察发现，这三个流的类型都是FILE*, fopen返回值类型，文件指针

```c
#include <stdio.h>

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

2-6 打开文件的方式

```
r      Open text file for reading.
       The stream is positioned at the beginning of the file.

r+     Open for reading and writing.
       The stream is positioned at the beginning of the file.

w      Truncate(缩短) file to zero length or create text file for writing.
       The stream is positioned at the beginning of the file.
```

```
w+     Open for reading and writing.
       The file is created if it does not exist, otherwise it is truncated.
       The stream  is  positioned at the beginning of the file.

a      Open for appending (writing at end of file).
       The file is created if it does not exist.
       The stream is positioned at the end of the file.

a+     Open for reading and appending (writing at end of file).
       The file is created if it does not exist.  The initial file  position
       for reading is at the beginning of the file,
       but output is always appended to the end of the file.
```

如上，是我们之前学的文件相关操作。还有 `fseek` `ftell` `rewind` 的函数，在C部分已经有所涉猎，请同学们自行复习。

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：

3-1 一种传递标志位的方法

```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;
```

```
20 }
```

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:

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

int main()
{
    umask(0);
    int fd = open("myfile", O_WRONLY|O_CREAT, 0644);
    if(fd < 0){
        perror("open");
        return 1;
    }

    int count = 5;
    const char *msg = "hello bit!\n";
    int len = strlen(msg);

    while(count--){
        write(fd, msg, len);//fd: 后面讲， msg: 缓冲区首地址， len: 本次读取，期望写入多少个字节的数据。 返回值：实际写了多少字节数据
    }

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

3-3 hello.c读文件

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

```c
#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`

```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。

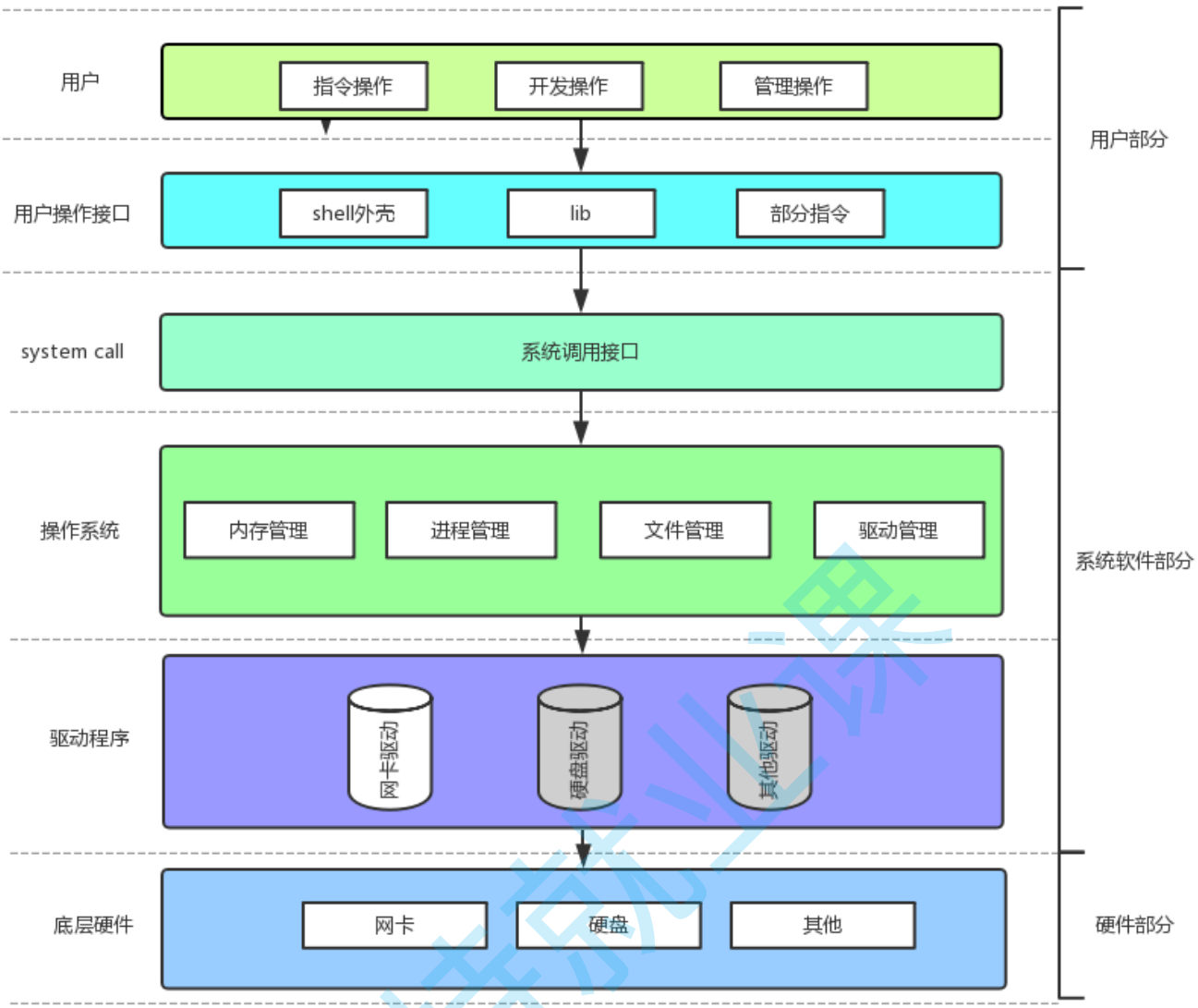
`open` 函数具体使用哪个，和具体应用场景相关，如目标文件不存在，需要open创建，则第三个参数表示创建文件的默认权限,否则，使用两个参数的open。

`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

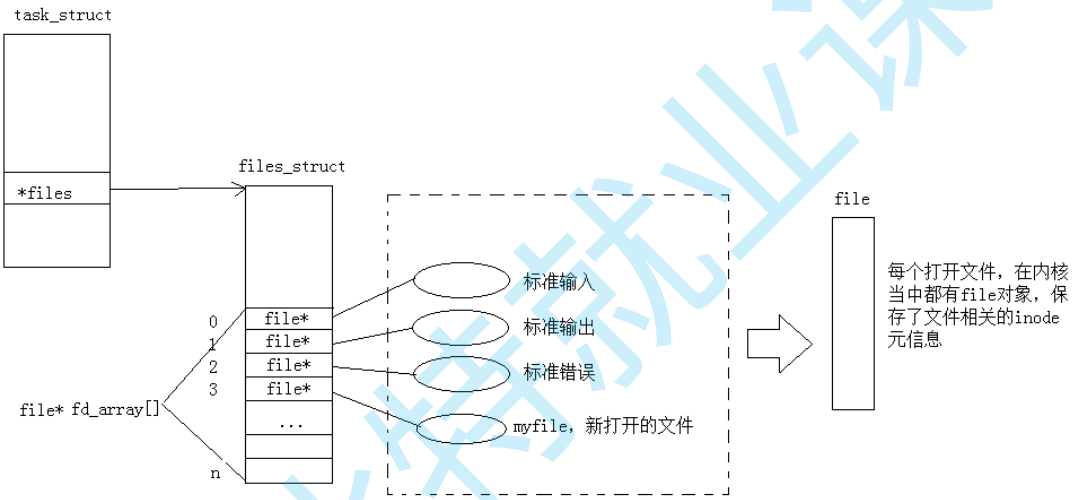
> - 通过对open函数的学习，我们知道了文件描述符就是一个小整数

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：

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

```c
#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;
}
```



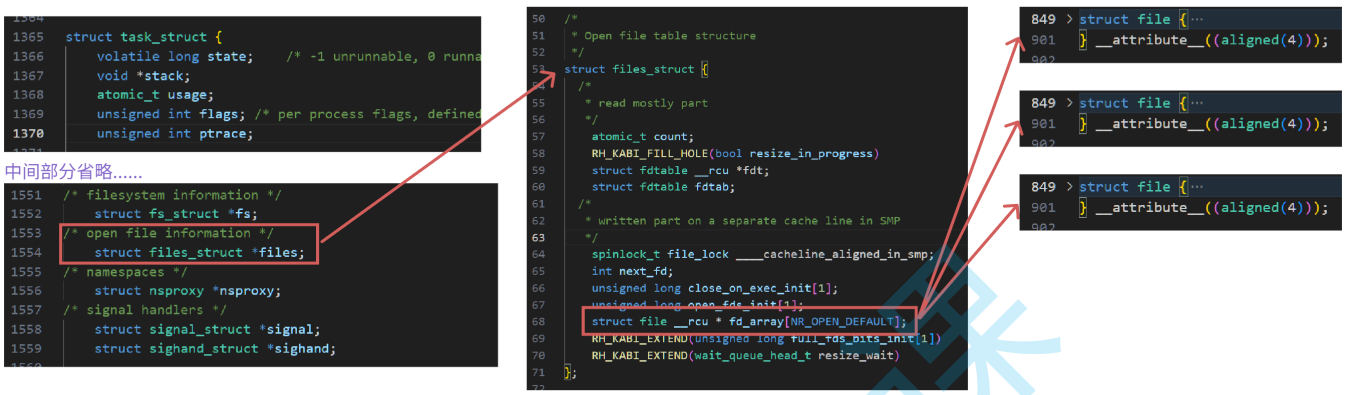
而现在知道，文件描述符就是从0开始的小整数。当我们打开文件时，操作系统在内存中要创建相应的数据结构来描述目标文件。于是就有了file结构体。表示一个已经打开的文件对象。而进程执行open系统调用，所以必须让进程和文件关联起来。每个进程都有一个指针*files, 指向一张表files_struct,该表最重要的部分就是包含一个指针数组，每个元素都是一个指向打开文件的指针！所以，本质上，文件描述符就是该数组的下标。所以，只要拿着文件描述符，就可以找到对应的文件。

对于以上原理结论我们可通过内核源码验证：

首先要找到`task_struct`结构体在内核中为位置，地址为：`/usr/src/kernels/3.10.0-1160.71.1.el7.x86_64/include/linux/sched.h`（3.10.0-1160.71.1.el7.x86_64是内核版本，可使用`uname -a`自行查看服务器配置， 因为这个文件夹只有一个，所以也不用刻意去分辨，内核版本其实也随意）

- 要查看内容可直接用vscode在windows下打开内核源代码
- 相关结构体所在位置

- `struct task_struct`：`/usr/src/kernels/3.10.0-1160.71.1.el7.x86_64/include/linux/sched.h`
- `struct files_struct`：`/usr/src/kernels/3.10.0-1160.71.1.el7.x86_64/include/linux/fdtable.h`
- `struct file`：`/usr/src/kernels/3.10.0-1160.71.1.el7.x86_64/include/linux/fs.h`



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，在看

```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

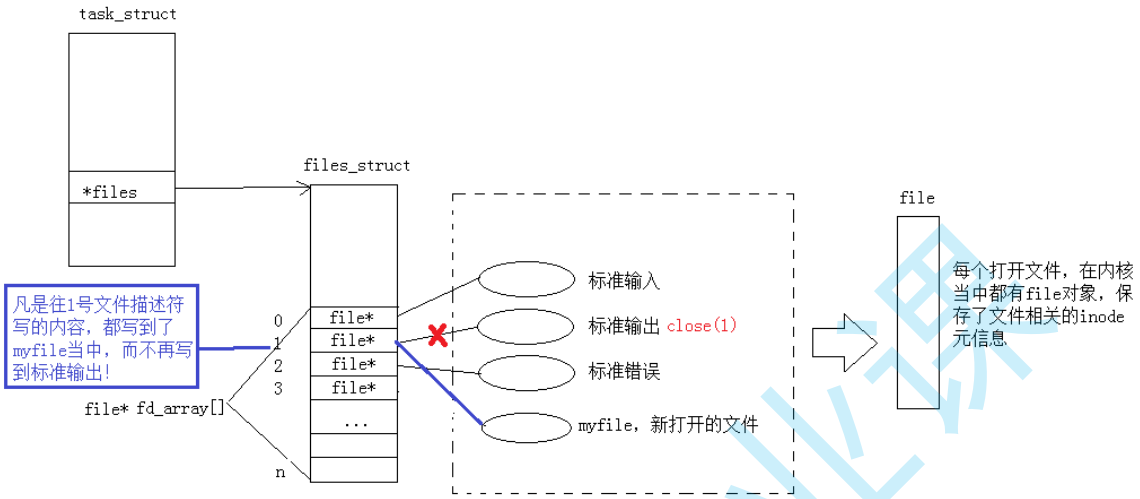
那如果关闭1呢？看代码：

```c
#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);

```

```
18      close(fd);
19      exit(0);
20  }
```

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:

```
1  #include <unistd.h>
2
3  int dup2(int oldfd, int newfd);
```

示例代码

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <fcntl.h>
4
5  int main() {
6    int fd = open("./log", O_CREAT | O_RDWR);
7    if (fd < 0) {
8        perror("open");
9        return 1;
10   }
```

```
    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?

3-6-5 在minishell中添加重定向功能

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <string>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <ctype.h>


using namespace std;

const int basesize = 1024;
const int argvnum = 64;
const int envnum = 64;
// 全局的命令行参数表
char *gargv[argvnum];
int gargc = 0;

// 全局的变量
int lastcode = 0;
```

```
25
26 // 我的系统的环境变量
27 char *genv[envnum];
28
29 // 全局的当前shell工作路径
30 char pwd[basesize];
31 char pwdenv[basesize];
32
33 // 全局变量与重定向有关
34 #define NoneRedir   0
35 #define InputRedir  1
36 #define OutputRedir 2
37 #define AppRedir    3
38
39 int redir = NoneRedir;
40 char *filename = nullptr;
41
42 // "    "file.txt
43 #define TrimSpace(pos) do{\
44     while(isspace(*pos)){\
45         pos++;\
46     }\
47 }while(0)
48
49 string GetUserName()
50 {
51     string name = getenv("USER");
52     return name.empty() ? "None" : name;
53 }
54
55 string GetHostName()
56 {
57     string hostname = getenv("HOSTNAME");
58     return hostname.empty() ? "None" : hostname;
59 }
60
61 string GetPwd()
62 {
63     if(nullptr == getcwd(pwd, sizeof(pwd))) return "None";
64     snprintf(pwdenv, sizeof(pwdenv),"PWD=%s", pwd);
65     putenv(pwdenv); // PWD=XXX
66     return pwd;
67
68     //string pwd = getenv("PWD");
69     //return pwd.empty() ? "None" : pwd;
70 }
71
```

```
72 string LastDir()
73 {
74     string curr = GetPwd();
75     if(curr == "/" || curr == "None") return curr;
76     // /home/whb/XXX
77     size_t pos = curr.rfind("/");
78     if(pos == std::string::npos) return curr;
79     return curr.substr(pos+1);
80 }
81
82 string MakeCommandLine()
83 {
84     // [whb@bite-alicloud myshell]$
85     char command_line[basesize];
86     snprintf(command_line, basesize, "[%s@%s %s]# ",\
87             GetUserName().c_str(), GetHostName().c_str(), LastDir().c_str());
88     return command_line;
89 }
90
91 void PrintCommandLine() // 1. 命令行提示符
92 {
93     printf("%s", MakeCommandLine().c_str());
94     fflush(stdout);
95 }
96
97 bool GetCommandLine(char command_buffer[], int size)   // 2. 获取用户命令
98 {
99     // 我们认为：我们要将用户输入的命令行，当做一个完整的字符串
100    // "ls -a -l -n"
101    char *result = fgets(command_buffer, size, stdin);
102    if(!result)
103    {
104        return false;
105    }
106    command_buffer[strlen(command_buffer)-1] = 0;
107    if(strlen(command_buffer) == 0) return false;
108    return true;
109 }
110
111 void ResetCommandline()
112 {
113    memset(gargv, 0, sizeof(gargv));
114    gargc = 0;
115
116    // 重定向
117    redir = NoneRedir;
118    filename = nullptr;
```

```
119 }
120
121 void ParseRedir(char command_buffer[], int len)
122 {
123     int end = len - 1;
124     while(end >= 0)
125     {
126         if(command_buffer[end] == '<')
127         {
128             redir = InputRedir;
129             command_buffer[end] = 0;
130             filename = &command_buffer[end] + 1;
131             TrimSpace(filename);
132             break;
133         }
134         else if(command_buffer[end] == '>')
135         {
136             if(command_buffer[end-1] == '>')
137             {
138                 redir = AppRedir;
139                 command_buffer[end] = 0;
140                 command_buffer[end-1] = 0;
141                 filename = &command_buffer[end]+1;
142                 TrimSpace(filename);
143                 break;
144             }
145             else
146             {
147                 redir = OutputRedir;
148                 command_buffer[end] = 0;
149                 filename = &command_buffer[end]+1;
150                 TrimSpace(filename);
151                 break;
152             }
153         }
154         else
155         {
156             end--;
157         }
158     }
159 }
160
161 void ParseCommand(char command_buffer[])
162 {
163     // "ls -a -l -n"
164     const char *sep = " ";
165     gargv[gargc++] = strtok(command_buffer, sep);
```

```
166     // =是刻意写的
167     while((bool)(gargv[gargc++] = strtok(nullptr, sep)));
168     gargc--;
169 }
170 
171 void ParseCommandLine(char command_buffer[], int len) // 3. 分析命令
172 {
173     ResetCommandline();
174     ParseRedir(command_buffer, len);
175     ParseCommand(command_buffer);
176     //printf("command start: %s\n", command_buffer);
177     // "ls -a -l -n"
178     // "ls -a -l -n" > file.txt
179     // "ls -a -l -n" < file.txt
180     // "ls -a -l -n" >> file.txt
181 
182 
183     //printf("redir: %d\n", redir);
184     //printf("filename: %s\n", filename);
185     //printf("command end: %s\n", command_buffer);
186 
187 }
188 
189 void debug()
190 {
191     printf("argc: %d\n", gargc);
192     for(int i = 0; gargv[i]; i++)
193     {
194         printf("argv[%d]: %s\n", i, gargv[i]);
195     }
196 }
197 
198 //enum
199 //{
200 //    FILE_NOT_EXISTS = 1,
201 //    OPEN_FILE_ERROR,
202 //};
203 
204 void DoRedir()
205 {
206     // 1. 重定向应该让子进程自己做!
207     // 2. 程序替换会不会影响重定向?不会
208     // 0. 先判断 && 重定向
209     if(redir == InputRedir)
210     {
211         if(filename)
212         {
```

```
213             int fd = open(filename, O_RDONLY);
214             if(fd < 0)
215             {
216                 exit(2);
217             }
218             dup2(fd, 0);
219         }
220         else
221         {
222             exit(1);
223         }
224     }
225     else if(redir == OutputRedir)
226     {
227         if(filename)
228         {
229             int fd = open(filename, O_CREAT | O_WRONLY | O_TRUNC, 0666);
230             if(fd < 0)
231             {
232                 exit(4);
233             }
234             dup2(fd, 1);
235         }
236         else
237         {
238             exit(3);
239         }
240     }
241     else if(redir == AppRedir)
242     {
243         if(filename)
244         {
245             int fd = open(filename, O_CREAT | O_WRONLY | O_APPEND, 0666);
246             if(fd < 0)
247             {
248                 exit(6);
249             }
250             dup2(fd, 1);
251         }
252         else
253         {
254             exit(5);
255         }
256     }
257     else
258     {
259         // 没有重定向,Do Nothong!
```

```
260     }
261 }
262 
263 // 在shell中
264 // 有些命令，必须由子进程来执行
265 // 有些命令，不能由子进程执行，要由shell自己执行 --- 内建命令 built command
266 bool ExecuteCommand()   // 4. 执行命令
267 {
268     // 让子进程进行执行
269     pid_t id = fork();
270     if(id < 0) return false;
271     if(id == 0)
272     {
273         //子进程
274         DoRedir();
275         // 1. 执行命令
276         execvpe(gargv[0], gargv, genv);
277         // 2. 退出
278         exit(7);
279     }
280     int status = 0;
281     pid_t rid = waitpid(id, &status, 0);
282     if(rid > 0)
283     {
284         if(WIFEXITED(status))
285         {
286             lastcode = WEXITSTATUS(status);
287         }
288         else
289         {
290             lastcode = 100;
291         }
292         return true;
293     }
294     return false;
295 }
296 
297 void AddEnv(const char *item)
298 {
299     int index = 0;
300     while(genv[index])
301     {
302         index++;
303     }
304 
305     genv[index] = (char*)malloc(strlen(item)+1);
306     strncpy(genv[index], item, strlen(item)+1);
```

```
307     genv[++index] = nullptr;
308 }
309 // shell自己执行命令，本质是shell调用自己的函数
310 bool CheckAndExecBuiltCommand()
311 {
312     if(strcmp(gargv[0], "cd") == 0)
313     {
314         // 内建命令
315         if(gargc == 2)
316         {
317             chdir(gargv[1]);
318             lastcode = 0;
319         }
320         else
321         {
322             lastcode = 1;
323         }
324         return true;
325     }
326     else if(strcmp(gargv[0], "export") == 0)
327     {
328         // export也是内建命令
329         if(gargc == 2)
330         {
331             AddEnv(gargv[1]);
332             lastcode = 0;
333         }
334         else
335         {
336             lastcode = 2;
337         }
338         return true;
339     }
340     else if(strcmp(gargv[0], "env") == 0)
341     {
342         for(int i = 0; genv[i]; i++)
343         {
344             printf("%s\n", genv[i]);
345         }
346         lastcode = 0;
347         return true;
348     }
349     else if(strcmp(gargv[0], "echo") == 0)
350     {
351         if(gargc == 2)
352         {
353             // echo $?
```

```
354             // echo $PATH
355             // echo hello
356             if(gargv[1][0] == '$')
357             {
358                 if(gargv[1][1] == '?')
359                 {
360                     printf("%d\n", lastcode);
361                     lastcode = 0;
362                 }
363             }
364             else
365             {
366                 printf("%s\n", gargv[1]);
367                 lastcode = 0;
368             }
369         }
370         else
371         {
372             lastcode = 3;
373         }
374         return true;
375     }
376     return false;
377 }
378
379 // 作为一个shell，获取环境变量应该从系统的配置来
380 // 我们今天就直接从父shell中获取环境变量
381 void InitEnv()
382 {
383     extern char **environ;
384     int index = 0;
385     while(environ[index])
386     {
387         genv[index] = (char*)malloc(strlen(environ[index])+1);
388         strncpy(genv[index], environ[index], strlen(environ[index])+1);
389         index++;
390     }
391     genv[index] = nullptr;
392 }
393
394 int main()
395 {
396     InitEnv();
397     char command_buffer[basesize];
398     while(true)
399     {
400         PrintCommandLine(); // 1. 命令行提示符
```

```
401         // command_buffer -> output
402         if( !GetCommandLine(command_buffer, basesize) )   // 2. 获取用户命令
403         {
404             continue;
405         }
406         //printf("%s\n", command_buffer);
407         //ls
408         //"ls -a -b -c -d"->"ls" "-a" "-b" "-c" "-d"
409         //"ls -a -b -c -d">hello.txt
410         //"ls -a -b -c -d">>hello.txt
411         //"ls -a -b -c -d"<hello.txt
412         ParseCommandLine(command_buffer, strlen(command_buffer)); // 3. 分析命令
413 
414         if ( CheckAndExecBuiltCommand() )
415         {
416             continue;
417         }
418 
419         ExecuteCommand();   // 4. 执行命令
420     }
421     return 0;
422 }
```

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：

```
1 struct file {
2 
3     ...
4 
5     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 */

```

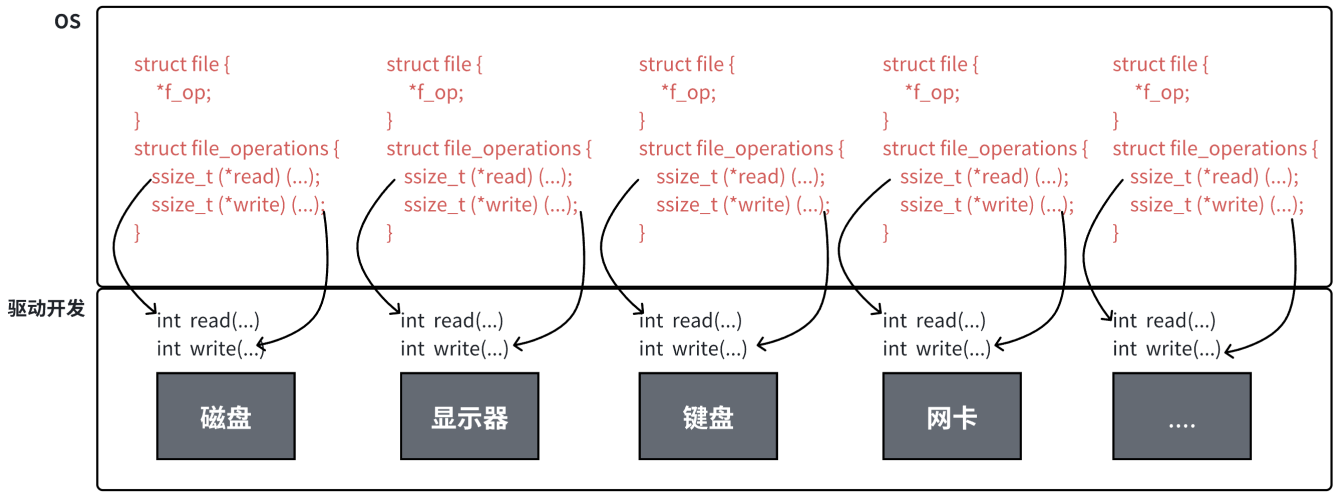
值得关注的是 `struct file` 中的 `f_op` 指针指向了一个 `file_operations` 结构体，这个结构体中的成员除了struct module* owner 其余都是**函数指针**。该结构和 `struct file` 都在fs.h下。

```
struct file_operations {
    struct module *owner;
    //指向拥有该模块的指针;
    loff_t (*llseek) (struct file *, loff_t, int);
    //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.
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    //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).
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    //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.
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    //初始化一个异步读 -- 可能在函数返回前不结束的读操作.
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    //初始化设备上的一个异步写.
    int (*readdir) (struct file *, void *, filldir_t);
    //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.
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
```

```
21      //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.
22      int (*open) (struct inode *, struct file *);
23      //打开一个文件
24      int (*flush) (struct file *, fl_owner_t id);
25      //flush 操作在进程关闭它的设备文件描述符的拷贝时调用;
26      int (*release) (struct inode *, struct file *);
27      //在文件结构被释放时引用这个操作. 如同 open, release 可以为 NULL.
28      int (*fsync) (struct file *, struct dentry *, int datasync);
29      //用户调用来刷新任何挂着的数据.
30      int (*aio_fsync) (struct kiocb *, int datasync);
31      int (*fasync) (int, struct file *, int);
32      int (*lock) (struct file *, int, struct file_lock *);
33      //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.
34      ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
35      unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
36      int (*check_flags)(int);
37      int (*flock) (struct file *, int, struct file_lock *);
38      ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
39      ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
40      int (*setlease)(struct file *, long, struct file_lock **);
41  };
42
```

`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大快于对磁盘的操作，故应用缓冲区可大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；

示例如下：

```c
#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;
    }

    printf("hello world: %d\n", fd);
    close(fd);
    return 0;
}
```

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：

```
[hyb@VM-8-12-centos buffer]$ ./myfile
[hyb@VM-8-12-centos buffer]$ ls
```

```
log.txt  makefile  myfile  myfile.c
[hyb@VM-8-12-centos buffer]$ cat log.txt
[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。

```c
#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;
    }

    printf("hello world: %d\n", fd);
    fflush(stdout);
    close(fd);
    return 0;
}
```


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：

```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");
```

```
13          return 0;
14      }
15
16      perror("hello world");
17      close(fd);
18      return 0;
19  }
```

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：

```
1  [hyb@VM-8-12-centos buffer]$ ./myfile
2  [hyb@VM-8-12-centos buffer]$ cat log.txt
3  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：

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6      const char *msg0="hello printf\n";
7      const char *msg1="hello fwrite\n";
8      const char *msg2="hello write\n";
9
10     printf("%s", msg0);
11     fwrite(msg1, strlen(msg0), 1, stdout);
12     write(1, msg2, strlen(msg2));
13
14     fork();
15
16     return 0;
17 }
```

运行出结果：

```
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：

```
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。

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，不过不再我们讨论范围之内。

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。

如果有兴趣，可以看看FILE结构体:

`typedef struct _IO_FILE FILE; 在/usr/include/stdio.h`

```
在/usr/include/libio.h
struct _IO_FILE {
  int _flags;               /* High-order word is _IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags
```

```
5
6   //缓冲区相关
7   /* The following pointers correspond to the C++ streambuf protocol. */
8   /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
9   char* _IO_read_ptr;         /* Current read pointer */
10  char* _IO_read_end;         /* End of get area. */
11  char* _IO_read_base;         /* Start of putback+get area. */
12  char* _IO_write_base;         /* Start of put area. */
13  char* _IO_write_ptr;         /* Current put pointer. */
14  char* _IO_write_end;         /* End of put area. */
15  char* _IO_buf_base;         /* Start of reserve area. */
16  char* _IO_buf_end;         /* End of reserve area. */
17  /* The following fields are used to support backing up and undo. */
18  char *_IO_save_base; /* Pointer to start of non-current get area. */
19  char *_IO_backup_base;  /* Pointer to first valid character of backup area */
20  char *_IO_save_end; /* Pointer to end of non-current get area. */
21
22  struct _IO_marker *_markers;
23
24  struct _IO_FILE *_chain;
25
26  int _fileno; //封装的文件描述符
27 #if 0
28  int _blksize;
29 #else
30  int _flags2;
31 #endif
32  _IO_off_t _old_offset; /* This used to be _offset but it's too small.  */
33
34 #define __HAVE_COLUMN /* temporary */
35  /* 1+column number of pbase(); 0 is unknown. */
36  unsigned short _cur_column;
37  signed char _vtable_offset;
38  char _shortbuf[1];
39
40  /*  char* _save_gptr;  char* _save_egptr; */
41
42  _IO_lock_t *_lock;
43 #ifdef _IO_USE_OLD_IO_FILE
44 };
```

5-5 简单设计一下libc库

my_stdio.h

```
$ cat my_stdio.h
#pragma once

#define SIZE 1024

#define FLUSH_NONE 0
#define FLUSH_LINE 1
#define FLUSH_FULL 2

struct IO_FILE
{
    int flag; // 刷新方式
    int fileno; // 文件描述符
    char outbuffer[SIZE];
    int cap;
    int size;
    // TODO
};

typedef struct IO_FILE mFILE;

mFILE *mfopen(const char *filename, const char *mode);
int mfwrite(const void *ptr, int num, mFILE *stream);
void mfflush(mFILE *stream);
void mfclose(mFILE *stream);
```

my_stdio.c

```
$ cat my_stdio.c
#include "my_stdio.h"
#include <string.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

mFILE *mfopen(const char *filename, const char *mode)
{
    int fd = -1;
    if(strcmp(mode, "r") == 0)
    {
        fd = open(filename, O_RDONLY);
    }
    else if(strcmp(mode, "w")== 0)
```

```
18      {
19          fd = open(filename, O_CREAT|O_WRONLY|O_TRUNC, 0666);
20      }
21      else if(strcmp(mode, "a") == 0)
22      {
23          fd = open(filename, O_CREAT|O_WRONLY|O_APPEND, 0666);
24      }
25      if(fd < 0) return NULL;
26      mFILE *mf = (mFILE*)malloc(sizeof(mFILE));
27      if(!mf)
28      {
29          close(fd);
30          return NULL;
31      }
32
33      mf->fileno = fd;
34      mf->flag = FLUSH_LINE;
35      mf->size = 0;
36      mf->cap = SIZE;
37
38      return mf;
39  }
40
41  void mfflush(mFILE *stream)
42  {
43      if(stream->size > 0)
44      {
45          // 写到内核文件的文件缓冲区中!
46          write(stream->fileno, stream->outbuffer, stream->size);
47          // 刷新到外设
48          fsync(stream->fileno);
49          stream->size = 0;
50      }
51  }
52
53  int mfwrite(const void *ptr, int num, mFILE *stream)
54  {
55      // 1. 拷贝
56      memcpy(stream->outbuffer+stream->size, ptr, num);
57      stream->size += num;
58
59      // 2. 检测是否要刷新
60      if(stream->flag == FLUSH_LINE && stream->size > 0 && stream-
    >outbuffer[stream->size-1]== '\n')
61      {
62          mfflush(stream);
63      }
```

```
64      return num;
65  }
66
67  void mfclose(mFILE *stream)
68  {
69      if(stream->size > 0)
70      {
71          mfflush(stream);
72      }
73      close(stream->fileno);
74  }
```

main.c

```
1   $ cat main.c
2   #include "my_stdio.h"
3   #include <stdio.h>
4   #include <string.h>
5   #include <unistd.h>
6
7   int main()
8   {
9       mFILE *fp = mfopen("./log.txt", "a");
10      if(fp == NULL)
11      {
12          return 1;
13      }
14      int cnt = 10;
15      while(cnt)
16      {
17          printf("write %d\n", cnt);
18          char buffer[64];
19          snprintf(buffer, sizeof(buffer),"hello message, number is : %d", cnt);
20          cnt--;
21          mfwrite(buffer, strlen(buffer), fp);
22          mfflush(fp);
23          sleep(1);
24      }
25      mfclose(fp);
26  }
27
```