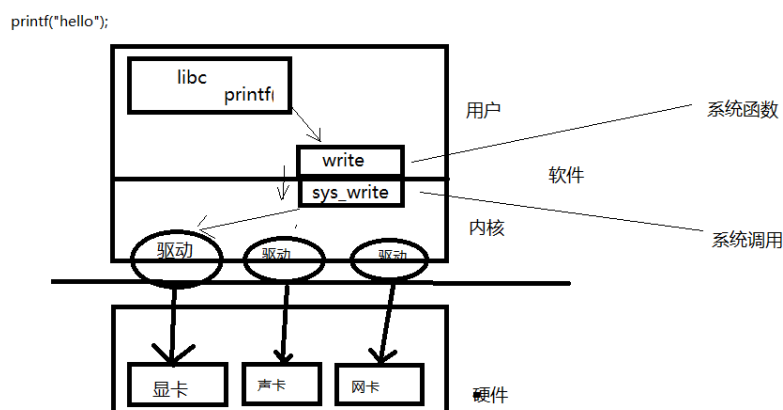


# linux系统函数编程

## 1.系统编程说在前面的话

### 1.1 系统调用

系统调用：由操作系统实现并提供给外部应用程序的编程接口。(Application Programming Interface, API)。是应用程序同系统之间数据交互的桥梁。



### 1.2 C标准库文件IO函数

`fopen`、`fclose`、`fseek`、`fgets`、`fputs`、`fread`、`fwrite`.....

`r` 只读、`r+`读写

`w`只写并截断为0、`w+`读写并截断为0

`a`追加只写、`a+`追加读写

## 2.IO

I/O: input&output, 是一切实现的基础。

- stdio标准I/O
- sysio系统调用I/O(文件IO)

我们除了可以使用系统IO来对话kernel, 也可以通过标准IO来对话kernel, 但是标准IO依赖于系统IO来实现。

linux有一套系统IO, windows有一套系统I/O, 他们的系统IO函数可能是不相同的。`printf`是标准IO, 在windows中是`printf`, linux也是`printf`, 但内核可能不太一样。比如`fopen`函数时标准I/O, linux的系统函数是`open()`函数, 但windows上是`openfile()`函数, 他们依赖的系统调用函数时不一样的。

- 标准IO的使用好处:
  - 移植性好
  - 合并系统调用 (大部分是好处)

注：当两个I/O都能使用的时候，我们优先使用标准I/O。

注：linux shell指令就是调用系统内核。

## 2.1 标准IO

标准IO中有一个类型贯穿始终，FILE（其实就是个结构体）

常用的标准函数：

```
fopen();
fclose();
// 字符操作
fgetc();
fputc();

// 字符串操作
fputs();
fgets();
// 二进制的操作
fread();
fwrite();

// printf和scanf一系列的函数
printf();
scanf();

// 文件位置和指针的操作
fseek();
ftell();
rewind();

fflush();
```

### 2.1.1 fopen

函数原型：

```
FILE *fopen(const char *pathname, const char *mode);
```

```
FOPEN(3) Linux Programmer's Manual FOPEN(3)
NAME
    fopen, fdopen, freopen - stream open functions
SYNOPSIS
    #include <stdio.h>
    FILE *fopen(const char *pathname, const char *mode);
    FILE *fdopen(int fd, const char *mode);
    FILE *freopen(const char *pathname, const char *mode, FILE *stream);
    Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
    fdopen(): _POSIX_C_SOURCE
```

功能：打开stream流。

返回值：FILE结构体

- 与fopen常一起使用的两个函数

```

NAME
    perror - print a system error message

SYNOPSIS
#include <stdio.h>

void perror(const char *s);

#include <errno.h>

const char * const sys_errlist[];
int sys_nerr;
int errno; /* Not really declared this way; see errno(3) */

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```

STRERROR(3) Linux Programmer's Manual

```

NAME
    strerror, strerror_r, strerror_l - return string describing error number

SYNOPSIS
#include <string.h>

char *strerror(int errnum);

int strerror_r(int errnum, char *buf, size_t buflen);
/* XSI-compliant */

char *strerror_r(int errnum, char *buf, size_t buflen);
/* GNU-specific */

char *strerror_l(int errnum, locale_t locale);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```

```

#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<string.h>

int main()
{
    FILE *fp;
    fp = fopen("tmp","r");
    if(fp == NULL)
    {
        /*
         * 当Linux环境运行的时候，有三个流打开，
         *      extern FILE *stdin  : 标准输入
         *      extern FILE *stdout : 标准输出
         *      extern FILE *stderr : 标准错误
         * 当程序开始运行的时候，与stdin、stdout和stderr相关的文件描述符分别是0、1、2；
         */
        //fprintf(stderr,"fopen() failed! errno = %d\n",errno);
        /*
         * 打印系统错误信息，自动关联全局变量errno
         */
        //perror("fopen error reason: ");

        /*char* str = strerror(errno);
        printf("%s\n",str);*/

        // 推荐perror和strerror两种方式。

        fprintf(stderr,"fopen error: %s\n",strerror(errno));
        exit(1);
    }

    puts("ok!");
}

```

```
    exit(0);  
}
```

#### 2.1.1.1 fopen函数内部分析

fopen函数的内部一定是对FILE结构体的参数进行操作，程序如下，那么存在一个问题：

**fopen的返回值是一个指针，那么这个指针指向的内存是那一块？** 是堆区，栈区，还是静态区？

```
FILE *fopen(const char *path, const char *mode)  
{  
    FILE *tmp = NULL;  
    tmp = malloc (sizeof(FILE));  
    tmp ->__ = __;  
    .....  
  
    return tmp;  
}
```

**问题分析：**指针指向的内存空间肯定不是栈区，其次内存空间不可能是静态常量区，因为常量区是一次申明多次使用，如果FILE放在静态常量区，打开一个文件正常，打开第二个文件的话，就会覆盖第一次文件的参数值，那么第一次打开的文件就会出错，所以也不可能是静态区。那只能是堆区了，每次进入fopen的函数，都会malloc一块FILE结构体大小的内存。从源码可以知道，fclose中必定有个free()操作。

**总结：**如果函数有逆操作，而且返回值是指针，那么数据一般都是放在堆上的。

#### 2.1.1.2 fopen函数最多可以打开多少个

- maxfopen.c

```
#include<stdio.h>  
#include<stdlib.h>  
#include<errno.h>  
#include<string.h>  
  
int main()  
{  
    FILE *fp = NULL;  
    int count = 0;  
  
    while(1)  
    {  
        fp = fopen("tmp", "r");  
        if(fp == NULL)  
        {  
            perror("fopen : ");  
            break;  
        }  
        count++;  
    }  
    printf("count = %d\n", count);  
    exit(0);  
}
```

```
}
```

```
marz@ubuntu1:~/cpp/io/stdio$ ls
errno.c  fopen  fopen.c  maxfopen  maxfopen.c  tmp
marz@ubuntu1:~/cpp/io/stdio$ ./maxfopen
fopen : : Too many open files
count = 1021
```

可以使用“ulimit -a”命令查看一个进程的参数

```
marz@ubuntu1:~/cpp/io/stdio$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size               (blocks, -f) unlimited
pending signals         (-i) 15242
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files              (-n) 1024
pipe size               (512 bytes, -p) 8
POSIX message queues    (bytes, -q) 819200
real-time priority      (-r) 0
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes      (-u) 15242
virtual memory          (kbytes, -v) unlimited
file locks              (-x) unlimited
```

为什么只能打开1021个，因为一个进程在启动的时候，在不更改当前系统环境的情况下，默认会打开 stdout、stdin、stderr这三个stream(流) 也会打开。

所以只能打开1021个FILE流。

可以通过ulimit命令更改点这个数值，比如 "ulimit -n 2048"。

### 2.1.3 fclose函数

### 2.1.4 fgetc和fputc函数

强调一下：一个函数的返回值是指针，要马上多问自己一句，这个指针指向的内容是静态区上的内容还是堆上的内容？

- man getchar

```
NAME
    fgetc, fgets, getc, getchar, ungetc - input of characters and strings

SYNOPSIS
    #include <stdio.h>

    int fgetc(FILE *stream);

    char *fgets(char *s, int size, FILE *stream);

    int getc(FILE *stream);

    int getchar(void);

    int ungetc(int c, FILE *stream);

DESCRIPTION
    fgetc() reads the next character from stream and returns it as an unsigned char cast to an int, or EOF on end of file or error.

    getc() is equivalent to fgetc() except that it may be implemented as a macro which evaluates stream more than once.

    getchar() is equivalent to getc(stdin).
```

从man手册我们可以知道，getchar、getc与fgetc的功能相同。从函数定义来看，getc和fgetc除了名字不同，其他都相同。

getc用于早期内核宏定义中，fgetc是函数定义。

- man putchar

```
NAME
    fputc, fputs, putc, putchar, puts - output of characters and strings

SYNOPSIS
    #include <stdio.h>

    int fputc(int c, FILE *stream);

    int fputs(const char *s, FILE *stream);

    int putc(int c, FILE *stream);

    int putchar(int c);

    int puts(const char *s);

DESCRIPTION
    fputc() writes the character c, cast to an unsigned char, to stream.

    fputs() writes the string s to stream, without its terminating null byte ('\0').

    putc() is equivalent to fputc() except that it may be implemented as a macro which evaluates stream more than once.

    putchar(c) is equivalent to putc(c, stdout).

    puts() writes the string s and a trailing newline to stdout.
```

**小功能：**实现linux的cp指令的功能

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

/*
    实现linux的cp指令的功能，
    cp src dest
    使得编译后能： ./mycp srcfile destfile 实现copy功能
*/
int main(int argc,char **argv)
{
    FILE *fps,*fpd;
    int ch;

    fps = fopen(argv[1],"r"); // 打开源文件，但是没有写的必要
    if(fps == NULL)
    {
        perror("fopen()");
        exit(1);
    }

    fpd = fopen(argv[2],"w");
    if(fpd == NULL)
    {
        perror("fopen()");
        exit(1);
    }

    while(1)
    {
        ch = fgetc(fps);
        if(ch == EOF) // EOF通常系统定义为-1
            break;
        fputc(ch,fpd);
    }
}
```

```

    }

    fclose(fpd);
    fclose(fps);
    exit(0);
}

```

```

marz@ubuntu1:~/cpp/io/stdio$ make mycpy
cc      mycpy.c      -o mycpy
marz@ubuntu1:~/cpp/io/stdio$ ls
errno.c  fopen.c  maxfopen.c  mycpy  mycpy.c  tmp
marz@ubuntu1:~/cpp/io/stdio$ ./mypy /etc/services /tmp/out
marz@ubuntu1:~/cpp/io/stdio$ diff /etc/services /tmp/out

```

实现cp的功能，通过diff指令（diff以逐行的方式，比较文本文件的异同处）查看他们的区别，没有弹出内容证明两个文件没有区别。

## 2.1.5 fgets和fputs函数

gets函数不建议使用，因为越界了也不会报错。

- fgets：一个字符串操作，行缓冲的模式的操作函数。

### SYNOPSIS

```

#include <stdio.h>
char *fgets(char *s, int size, FILE *stream);

```

### DESCRIPTION

fgets() reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.

fgets的结束有两种情况，这个函数会正常结束：

```

#define SIZE 5
char buf[SIZE]
fget(buf,SIZE,stream)

```

知识点：在C语言中，字符串总是以'\0'作为结尾，所以'\0'也被称为字符串结束标志，或者字符串结束符。

第一种情况：读到了（SIZE-1）个字节

第二种情况：读到了'\n'

读取一个字符串："abcdef"，那么fgets只会读到d的位置，文件指针定位到e的位置。

读取一个字符串："ab"，那么fgets会读取到 'a','b','\n','\0'，因为文件打开的时候就有一个换行符。

读取一个字符串："abcd"，用上述代码读取，会读取两次

第一次：a b c d '\0'

第二次：'\n' '\0'

- fputs

#### SYNOPSIS

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

#### DESCRIPTION

fputs() writes the string *s* to stream, without its terminating null byte ('\0').

**小功能：**实现linux的cp指令的功能，用fgetc和fputs函数

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

#define BUFSIZE 1024

int main(int argc,char **argv)
{
    FILE *fps,*fpd;
    char buf[BUFSIZE];
    if(argc < 3)
    {
        fprintf(stderr,"Usage:%s <src_file> <dest_file>\n",argv[0]);
    }

    fps = fopen(argv[1],"r"); // 打开源文件，但是没有写的必要
    if(fps == NULL)
    {
        fclose(fps);
        perror("fopen()");
        exit(1);
    }

    fpd = fopen(argv[2],"w");
    if(fpd == NULL)
    {
        fclose(fpd);
        perror("fopen()");
        exit(1);
    }

    while(fgets(buf,BUFSIZE,fps) != NULL)
    {
        fputs(buf,fpd);
    }
    -
    fclose(fpd);
```



```
fclose(fps);
exit(0);
}
```

## 2.1.6 fread和fwrite函数

- fread & fwrite

```
NAME
    fread, fwrite - binary stream input/output

SYNOPSIS
    #include <stdio.h>

    size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

    size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE
    *stream);

DESCRIPTION
    The function fread() reads nmemb items of data, each size bytes long,
    from the stream pointed to by stream, storing them at the location given by
    ptr.

    The function fwrite() writes nmemb items of data, each size bytes
    long, to the stream pointed to by stream, obtaining them from the location
    given by ptr.

RETURN
    On success, fread() and fwrite() return the number of items read or
    written. This number equals the number of bytes transferred only when size
    is 1. If an error occurs, or the end of the file is reached, the return
    value is a short item count (or zero).
```

fread和fwrite读取或者写入的数据大小为size\*nmemb的大小，他们比较适合数据对齐的场景，进行批量的操作。但是也存在缺点，那就是读取/写入的时候不会进行边界校验，进行批量操作的时候，如果第一个字节缺失的话，后面的数据就都会对不上了。

```
fread(buf, size, nmemb, stream)
```

假设buf是10个字节的char\* 数组

可以： fread(buf, 1, 10, stream)

也可以： fread(buf, 10, 1, stream)

第一种情况，当我们文件汇中数据量非常充足，

fread(buf, 1, 10, stream) 读取10个对象 每个对象大小为一个字节

fread(buf, 10, 1, stream) 读取1个对象 对象的大小为10个字节

第二种情况：文件只有5个字节

`fread(buf,1,10,stream)` 读取10个对象，但是文件只有5个对象，那么该函数能读到5个对象，大小为5个组合’  
`fread(buf,10,1,stream)` 读一个对象，对象大小必须为10个字节，但对象只有5个字节，不够一个对象，所以返回值为零

所以`fread`只能操作这种数据比较整齐的情况。

所以为了保险起见，老师推荐单字节（1个字节）来使用，就是`fread(buf,1,10,stream)`这种操作方式。

**小功能：**实现linux的cp指令的功能，用`fread`和`fwrite`函数

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

#define BUFSIZE 1024

int main(int argc,char **argv)
{
    FILE *fps,*fpd;
    char buf[BUFSIZE];
    int n;
    if(argc < 3)
    {
        fprintf(stderr,"Usage:%s <src_file> <dest_file>\n",argv[0]);
    }

    fps = fopen(argv[1],"r"); // 打开源文件，但是没有写的必要
    if(fps == NULL)
    {
        fclose(fps);
        perror("fopen()");
        exit(1);
    }

    fpd = fopen(argv[2],"w");
    if(fpd == NULL)
    {
        fclose(fpd);
        perror("fopen()");
        exit(1);
    }

    //不太建议写成 fread(buf,1,BUFSIZE,fps) fwrite(buf,1,BUFSIZE,fpd)
    // 因为你无法确定时候读取到BUFSIZE个的数据，而应该写成下面的方式
    // 读取到n个数据，就写入n个数据。
    while(( n = fread(buf,1,BUFSIZE,fps)) > 0)
    {
        fwrite(buf,1,n,fpd);
    }
    fclose(fpd);
    fclose(fps);
    exit(0);
}
```

## 2.1.7 标准IO-printf和scanf族函数

- IO-printf

### NAME

printf, fprintf, dprintf, sprintf, snprintf, vprintf, vfprintf, vdprintf, vsprintf, vsnprintf - formatted output conversion

### SYNOPSIS

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

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int dprintf(int fd, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

```
#include <stdarg.h>
```

```
int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vdprintf(int fd, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

printf()函数将数据按照指定的格式输出到标准输出

更推荐fprintf()函数，可以指定FILE \*stream流中(stdin、stdout、stderr)，可以将不同的数据输出到不同的流中，还可以将信息输入到文件中。

- sprintf函数，将"... "参数按指定格式输出到char \*str中，但是无法知道char \*str的长度。所以snprintf就是对sprintf的升级，指定了size的大小。

### 小功能：sprintf函数

atoi函数：把字符串转换为整型的函数

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

/**
 * 把123456这样一个字符串转换为整型数
 */
int main()
{
    // 如果字符串中有字母的话，那么就会拿到字母或者尾0前为止。
    char str[] = "1234a6";
    printf("%d\n",atoi(str));
    exit(0);
}

//-----
//-----
// 可以把sprintf当做atoi的反函数来看，c库中没有提供整型转字符串的函数。
//-----
//-----
```

```

#include<stdio.h>
#include<stdlib.h>

int main()
{
    int year = 2014, month=5, day=13;

    char buf[1024];
    // 这样也能生成字符串假象，但是数据不能当做字符串来处理。
    printf("%d-%d-%d\n", year, month, day);

    sprintf(buf, "%d-%d-%d", year, month, day);
    puts(buf);

    exit(0);
}

```

- scanf

```

NAME
    scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf - input format
    conversion

SYNOPSIS
    #include <stdio.h>

    int scanf(const char *format, ...);
    int fscanf(FILE *stream, const char *format, ...);
    int sscanf(const char *str, const char *format, ...);

    #include <stdarg.h>

    int vscanf(const char *format, va_list ap);
    int vsscanf(const char *str, const char *format, va_list ap);
    int vfscanf(FILE *stream, const char *format, va_list ap);

```

scanf一系列函数中，要慎重使用%s，因为后面会跟上&a这样的参数，但是你根本不知道输入的参数有多长。这是scanf在使用上最大的缺点之一。

## 2.1.8 标准IO-fseeko和ftello

```

NAME
    fgetpos, fseek, fsetpos, ftell, rewind - reposition a stream

SYNOPSIS
    #include <stdio.h>

    int fseek(FILE *stream, long offset, int whence);
    long ftell(FILE *stream);

    void rewind(FILE *stream);

    int fgetpos(FILE *stream, fpos_t *pos);

    int fsetpos(FILE *stream, const fpos_t *pos);

```

ftell返回值为long，以32位系统为例，long的数据范围为 $-2^{32} \sim 2^{32}-1$ 这个范围，但是文件的定位符没有负的，所以ftell的文件大小不能超过 $2^{23}-1$ 。这是fseek和ftell最致命的一部分。所以有了fseeko和ftello

```

NAME
    fseeko, ftello - seek to or report file position

SYNOPSIS
    #include <stdio.h>

    int fseeko(FILE *stream, off_t offset, int whence);
    off_t ftello(FILE *stream);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

    fseeko(), ftello():
        _FILE_OFFSET_BITS == 64 || _POSIX_C_SOURCE >= 200112L
        (defining the obsolete _LARGEFILE_SOURCE macro also works)

```

## 2.1.9 文件位置函数和缓冲区刷新函数

- 文件位置指针操作函数：

\*\*

<b>NAME</b>	<code>fgetpos</code> , <code>fseek</code> , <code>fsetpos</code> , <code>ftell</code> , <code>rewind</code> - reposition a stream
<b>SYNOPSIS</b>	<pre> #include &lt;stdio.h&gt;  int fseek(FILE *stream, long offset, int whence);  long ftell(FILE *stream);  void rewind(FILE *stream);  int fgetpos(FILE *stream, fpos_t *pos);  int fsetpos(FILE *stream, const fpos_t *pos); </pre>
<b>DESCRIPTION</b>	<p>The <code>fseek()</code> function sets the file position indicator for the stream pointed to by <code>stream</code>. The new position, measured in bytes, is obtained by adding <code>offset</code> bytes to the position specified by <code>whence</code>. If <code>whence</code> is set to <code>SEEK_SET</code>, <code>SEEK_CUR</code>, or <code>SEEK_END</code>, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the <code>fseek()</code> function clears the end-of-file indicator for the stream and undoes any effects of the <code>ungetc(3)</code> function on the same stream.</p> <p>The <code>ftell()</code> function obtains the current value of the file position indicator for the stream pointed to by <code>stream</code>.</p> <p>The <code>rewind()</code> function sets the file position indicator for the stream pointed to by <code>stream</code> to the beginning of the file. It is equivalent to:</p> <pre>(void) fseek(stream, 0L, SEEK_SET)</pre> <p>except that the error indicator for the stream is also cleared (see <code>clearerr(3)</code>).</p> <p>The <code>fgetpos()</code> and <code>fsetpos()</code> functions are alternate interfaces equivalent to <code>ftell()</code> and <code>fseek()</code> (with <code>whence</code> set to <code>SEEK_SET</code>), setting and storing the current value of the file offset into or from the object referenced by <code>pos</code>. On some non-UNIX systems, an <code>fpos_t</code> object may be a complex object and these routines may be the only way to portably reposition a text stream.</p>
<b>RETURN VALUE</b>	<p>The <code>rewind()</code> function returns no value. Upon successful completion, <code>fgetpos()</code>, <code>fseek()</code>, <code>fsetpos()</code> return 0, and <code>ftell()</code> returns the current offset. Otherwise, -1 is returned and <code>errno</code> is set to indicate the error.</p>

`whence`的取值: `SEEK_SET` `SEEK_CUR` `SEEK_END`

- flush函数

```

#include<stdio.h>
#include<stdlib.h>

int main()
{
    int i;
    printf("Before while");
    while(1);
    printf("After while");

    exit(0);
}

```

现象：编译后执行可执行文件，终端不会用任何输出。

`printf`一些列函数往标准终端来输出的时候，标准终端是典型的缓冲模式，一般是换行的时候或者一行满了的时候刷新缓冲区。

所以在写`print`的时候，如果没有特殊的要求，要加上个"`\n`"。

```

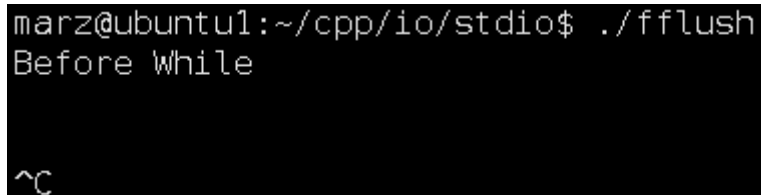
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int i;
    //    printf("Before while\n");
}

```

```
printf("Before while");  
fflush(stdout);  
while(1);  
printf("After while");  
fflush(NULL); // 代表字符串结束的时候 NULL='\0'  
exit(0);  
}
```

使用fflush函数和\n的效果是一样的。



```
marz@ubuntu1:~/cpp/io/stdio$ ./fflush  
Before while  
^C
```

- 缓冲区的作用：

大多数情况下是好事，合并系统调用。

- 行缓冲：换行的时候刷新，满了的时候刷新，强制刷新（标准输出是这样的，因为是终端设备）
- 全缓冲：满了的时候刷新，强制刷新（默认，只要不是终端设备）。
- 无缓冲：如stderr（出错就立即输出），需要立即输出的内容

与缓冲相关的函数：setbuf()，但一般不建议不设置该函数。

## 2.1.10 标准IO-临时文件

临时文件，主要考虑两个问题：

1. 如何不冲突
2. 及时销毁

两个函数，能创建临时文件。

- tmpnam
- tmpfile

---

## 2.2 系统IO/文件IO

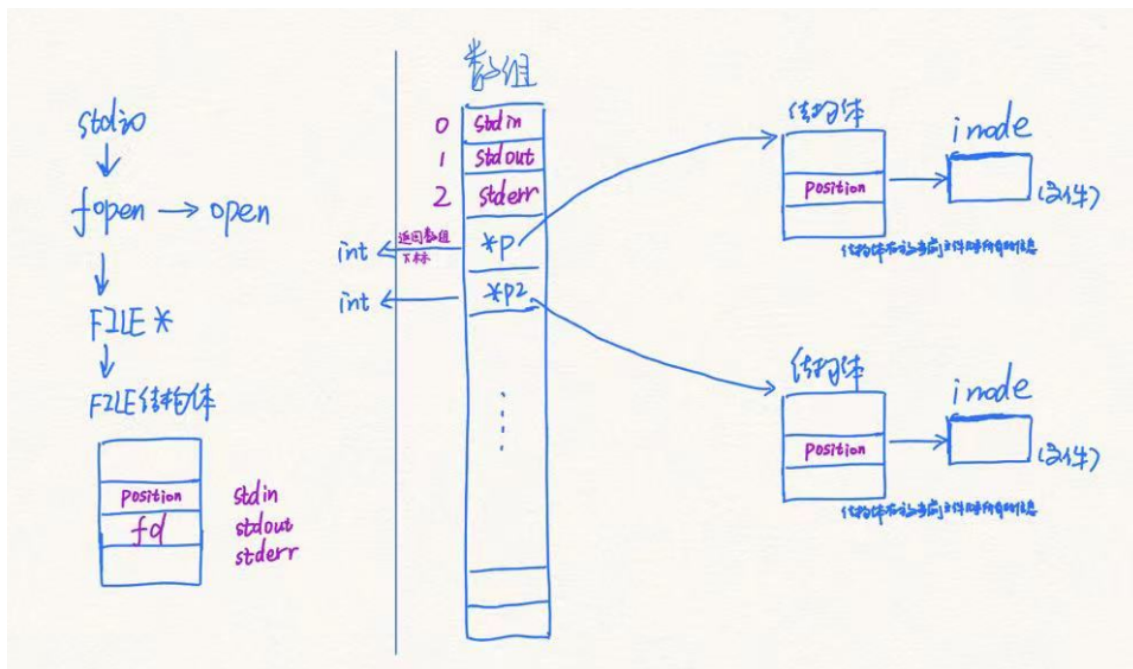
FILE类型是系统IO中贯穿始终的类型，

### 2.2.1 文件描述符实现原理

- 文件描述符的概念

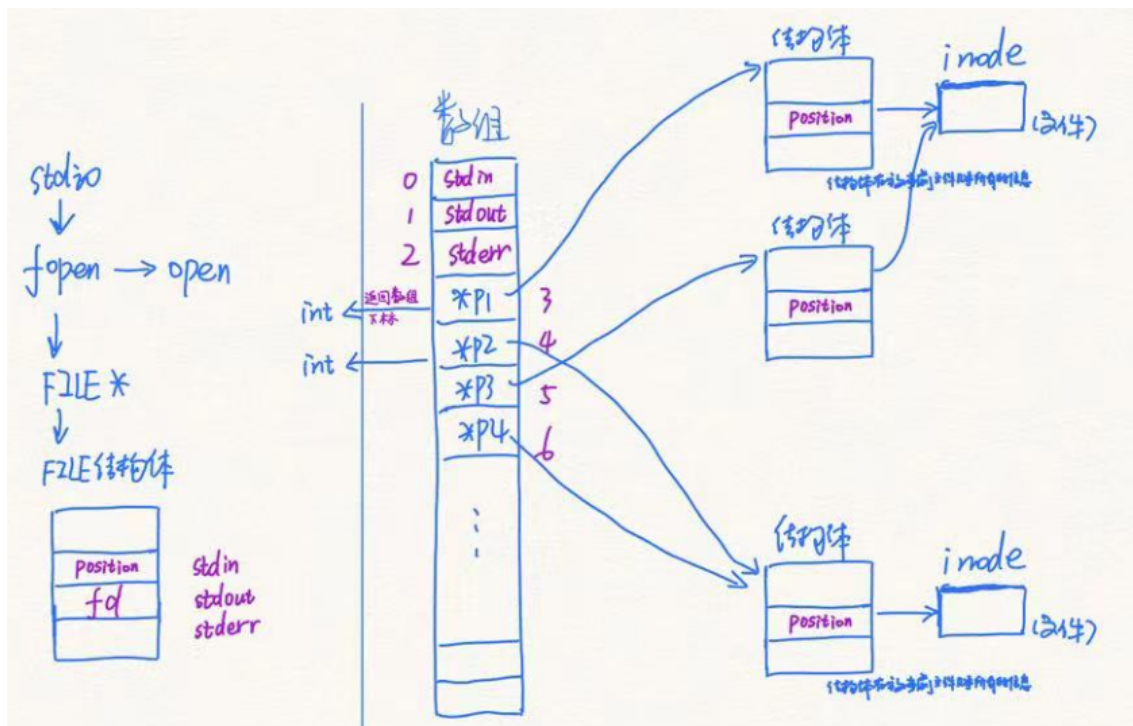
**文件描述符 (fd, file descriptor) 是文件IO中贯穿始终的类型**

整型，数组下标。



- 打开磁盘上的每一个文件都有唯一的标识符，我们称文件为inode。
- 每次打开文件都会关联一个结构体，你要操作的所有属性几乎都存放在这里。能读能写能定位，所以结构体里面一定有一个position的参数。**把文件结构体对应的地址或指针保存到一个数组中去，给用户返回的是一个整型数。**下次对文件进行操作的时候，就拿这个整型数就可以进行了。通过整型数就能定位数据下标对应的结构体指针，然后就可以操作该文件对应的结构体了。
- `fopen`会产生结构体FILE，`open`函数也会产生结构体，这两个结构体中有很多都是相同的。`fopen`可以打开多个文件，所以FILE结构体中一定有fd(文件描述符)。
- 数组的长度是1024，在没有修改默认环境的情况下，通过`ulimit`命令修改就是在修改数组的大小，数组的长度决定了你能打开的文件个数。`fopen`会产生三个标准fd (`stdin`、`stdout`、`stderr`)，这三个流(stream)会存放到数组0、1、2的三个位置上，所以标准输入输出才能正常使用。打开文件的时候，**文件描述符优先使用当前可用范围内最小的。**当打开第一个文件的时候，fd会从3开始。
- 这个数组是存在进程空间里面的，每一个进程都会有一个这样的数组。图示的是一个进程打开两个文件的状况。
- 一个文件打开多少次就会产生多少个结构体。谁执行`open`都会产生一个结构体，哪怕打开同一个文件，也会产生一个结构体。
- 会存在的问题





- 同一个文件，在一个空间（进程中）连续打开两次或者更多次数，如p1,p3就代表多次打开同一个文件的状况。那么两个文件描述符都是关联的同一个文件。close() p1的时候不影响p4操作文件，同理p4的时候。
- 复制p2到6的位置。p2和p4他们就关联的同一结构体，指向同一结构体的起始位置。那么close() p2的时候，p4是不是不能再用结构体了？所以一个文件的结构体中一定有一个打开计数，来反应我的结构体被几个指针引用。

## 2.2.2 文件IO操作：open、close、read、write、lseek

前面的fopen内部实现都是open，前面的fgets和fputs类函数的内部实现都是read和write。fseek内部实现时lseek。

总之，前面介绍的函数都是简历在这五个函数之上的。

**小功能：**用read和write实现文件的读写

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

#define BUFSIZE 1024
/**
 * 用标准IO来实现文件的读写
 */
int main(int argc, char** argv)
{
    int sfd, dfd;
    char buf[BUFSIZE];
    int len; // read的返回值
    int ret; // write的返回值
    int pos;
```

```

if(argc < 3)
{
    fprintf(stderr,"Usage:%s <src_file> <dest_file> \n",argv[0]);
    exit(1);
}

// 打开两个文件
sfd = open(argv[1],O_RDONLY);
if(sfd < 0)
{
    perror("open()");
    exit(1);
}
// 写的文件不一定存在
dfd = open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,0777);
if(dfd<0)
{
    close(sfd);
    perror("open()");
    exit(1);
}
// 再循环中进行读写
while(1)
{
    len = read(sfd,buf,BUFSIZE);
    if(len < 0)// 读文件出错
    {
        perror("read");
        break;
    }
    if(len == 0)
    {
        break;
    }
    // 当len > 0的时候才是读到了有效自己
    // 这里可能出现write的值<len的情况，这样就会丢失数据，可以进行优化，加一个while循环
    pos = 0;
    while(len > 0)
    {
        ret = write(dfd,buf+pos,len);
        if(ret < 0)
        {
            perror("write");
            exit(1);
        }
        pos += ret;
        len -= ret;
    }
}

// 关闭文件
close(dfd);
close(sfd);

```

```
    exit(0);  
}
```

### 2.2.3 系统IO与标准IO的区别

主要在：响应速度和吞吐量上

系统IO：响应速度快，只要有数据来就响应操作。

标准IO：有缓冲机制，只有当flush、换行、文件结束的时候在后响应。

每执行一次系统IO，都要从用户态切换到kernel态。比较费时。

**面试题：**如何使一个程序变快？

**注意：**标准IO和系统IO不可以混用

```
// 混用测试  
  
#include<stdio.h>  
#include<stdlib.h>  
#include <unistd.h>  
  
int main()  
{  
    // 往终端打印a  
    putchar('a');  
    // 这里的第一个参数1 = stdout,往终端打印b  
    write(1,"b",1);  
  
    putchar('a');  
    write(1,"b",1);  
  
    putchar('a');  
    write(1,"b",1);  
  
    printf("\n");  
    exit(0);  
}
```

```
bbbaamarz@ubuntu1:~/cpp/io/sysio$ make ab  
cc      ab.c      -o ab  
marz@ubuntu1:~/cpp/io/sysio$ ./ab  
bbbaaa
```

从结果来看，并不是我们程序的输出顺序。通过strace命令来查看可执行文件，可以帮我们来看一个可执行文件系统调用是如何发生的？

```
marz@ubuntu1:~/cpp/io/sysio$ strace ./ab
execve("./ab", ["/.ab"], 0x7ffcada7ee90 /* 26 vars */) = 0
brk(NULL) = 0x5595d4b18000
arch_prctl(0x3001 /* ARCH ??? */, 0x7fff0f174370) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/opt/module/dynamicLib/lib/tls/haswell/x86_64/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
stat("/opt/module/dynamicLib/lib/tls/haswell/x86_64", 0x7fff0f1735c0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/opt/module/dynamicLib/lib/tls/haswell/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
stat("/opt/module/dynamicLib/lib/tls/haswell", 0x7fff0f1735c0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/opt/module/dynamicLib/lib/tls/x86_64/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
stat("/opt/module/dynamicLib/lib/tls/x86_64", 0x7fff0f1735c0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/opt/module/dynamicLib/lib/tls/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
arch_prctl(ARCH_SET_FS, 0x7fd7fd966540) = 0
mprotect(0x7fd7fd95b000, 16384, PROT_READ) = 0
mprotect(0x5595d3224000, 4096, PROT_READ) = 0
mprotect(0x7fd7fd9a5000, 4096, PROT_READ) = 0
munmap(0x7fd7fd967000, 68167) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
brk(NULL) = 0x5595d4b18000
brk(0x5595d4b39000) = 0x5595d4b39000
write(1, "b", 1b) = 1
write(1, "b", 1b) = 1
write(1, "b", 1b) = 1
write(1, "aaa\n", 4aaa) = 4
exit_group(0) = ?
```

可以明显的看出，3个a是一次写入的。

- 标准IO和系统IO转换函数：
  - fileno：把标准IO的操作转换成系统IO的操作

```
int fileno(FILE *stream);
```

```
The function fileno() examines the argument stream and returns the integer file descriptor used to implement this stream. The file descriptor is still owned by stream and will be closed when fclose(3) is called. Duplicate the file descriptor with dup(2) before passing it to code that might close it.
```

- fdopen：把一个已经打开的fd（文件描述符），指定他的操作方式，然后封装成FILE类型来操作。就是把系统IO转换成标准IO。

## 2.2.4 IO的效率问题

```
marz@ubuntu1:~/cpp/io/sysio$ time ./mycpy /etc/services /tmp/out
real    0m0.004s
user    0m0.000s
sys     0m0.003s
```

time指令，测试后面这句话它的执行时间。

real time：理论上的值是user+sys+一点点时间（操作系统调度等待时间，被中断打断的时间等）

user time：在user层消耗的时间

sys time：当前进程执行时，在core层所花费的时间

**任务：**将mycpy.c程序进行更改，将BUFSIZE的值放大，记录并观察进程消耗的时间。注意性能最佳拐点出现时的BUFSIZE值，以及何时程序会出问题。

可以测试不同的BUFSIZE值的所花费的时间。（128B~ 16M，找一个大文件3G~5G），填充下面的表格，找出性能的拐点。

BUFSIZE	real	user	sys	循环次数

### 2.2.5 文件共享

面试题：写程序删除一个文件的第10行

补充函数：truncate

### 2.2.6 原子操作

原子操作：不可分割的操作

原子操作的作用：解决竞争和冲突

### 2.2.7 程序中的重定向：dup, dup2

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

#define FNAME "/tmp/out"

// 将hello输出的文件中
// 使用puts函数会默认将内容输出到终端上
int main()
{
    int fd;
    fd = open(FNAME,O_WRONLY|O_CREAT|O_TRUNC,0777);
    if(fd<0)
    {
        perror("OPEN");
        exit(1);
    }
    close(1);
    dup(fd);

    close(fd);

    /*****下面代码不能变*****/
    puts("hello!");
```

```
    exit(0);  
}
```

上面程序存在两个问题

- 如果fd就是1，那么dup(fd)就出错了
- 如果close(1)的时候，fd=1有可能被其他的进程占用。也就是close和dup操作不原子。

可以使用dup2(oldfd,newfd)来代替

```
if(fd<0)  
{  
    perror("OPEN");  
    exit(1);  
}  
//close(1);  
//dup(fd);  
dup2(fd,1); // dup2是上面两句话的原子操作
```

当dup2的oldfd为1的时候，他不会做任何操作。如果当fd为1的时候，下面close(fd)不能冒然关闭。

```
int main()  
{  
    int fd;  
    fd = open(FNAME,O_WRONLY|O_CREAT|O_TRUNC,0777);  
    if(fd<0)  
    {  
        perror("OPEN");  
        exit(1);  
    }  
    //close(1);  
    //dup(fd);  
  
    dup2(fd,1); // dup2是上面两句话的原子操作  
  
    if(fd != 1)  
        close(fd);  
  
    /*****下面代码不能变*****/  
    puts("hello!");  
    exit(0);  
}
```

## 2.2.8 同步: sync、fsync、fdata、sync

- fcntl(): 文件描述符所变的魔术几乎都来源于该函数

```
NAME
    fcntl - manipulate file descriptor

SYNOPSIS
    #include <unistd.h>
    #include <fcntl.h>

    int fcntl(int fd, int cmd, ... /* arg */);

DESCRIPTION
    fcntl() performs one of the operations described below on the open file descriptor fd. The operation is determined by cmd.
```

- ioctl(): 设备相关的内容

```
NAME
    ioctl - control device

SYNOPSIS
    #include <sys/ioctl.h>

    int ioctl(int fd, unsigned long request, ...);

DESCRIPTION
    The ioctl() system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with ioctl() requests. The argument fd must be an open file descriptor.

    The second argument is a device-dependent request code. The third argument is an untyped pointer to memory. It's traditionally char *argp (from the days before void * was valid C), and will be so named for this discussion.

    An ioctl() request has encoded in it whether the argument is an in parameter or out parameter, and the size of the argument argp in bytes. Macros and defines used in specifying an ioctl() request are located in the file <sys/ioctl.h>.
```

- /dev/fd/目录: 虚目录, 显示的是当前进程的文件描述符信息