# 流

## 简介

流可以简单理解为文件。

低级IO与标准（高级）IO的区别：

**open和fopen的区别：**

前者属于低级IO，后者是高级IO。

前者返回一个文件描述符，后者返回一个文件指针。

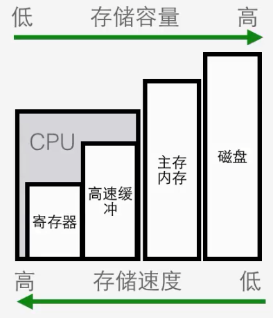
前者无缓冲，后者有缓冲。

前者与read，write等配合使用， 后者与fread，fwrite等配合使用。

后者是在前者的基础上扩充而来的，在大多数情况下，用后者。

## 文件

文件在计算机系统中的重要性：



### 内容

比较两个文件内容vimdiff file1 file2

### 属性

## IO

输入输出(I/O)是任何一种编程环境中最基本的功能。

任何系统，程序读写文件的内容前，首先在程序与文件之间建立连接。

## 流

C库提供一种机制描述程序与文件的这种连接（流即文件的连接）。

C库的IO函数就是完成流操作。

用流函数打开一个文件，它返回一个流。

### 打开流

流的数据类型：FILE类型（代表了文件的所有内部状态）

FILE对象由IO库函数内部分配、管理

在打开流的时候需要考虑：哪个文件（新建），打开的方式，光标位置

在Linux系统中使用man fopen查看：

原型：FILE \*fopen(const char \*path, const char \*mode);

参数：path表示需要打开的文件，mode表示光标所在位置以及以什么方式打开(r/r+/w/w+/a/a+)。

r：读，光标在头部

r+：读写，光标在头部

w：写，光标在文件头（对已有内容清空）

w+：读写

a：追加，写操作，光标定位到文件尾部

a+：追加，读写

示例：

#include <stdio.h>

int main(int argc,char \*\*argv)

{

FILE \*fp = NULL;

fp = fopen("a.c","w");

if ( NULL == fp )

{

perror("fopen");

return -1;

}

fclose(fp);

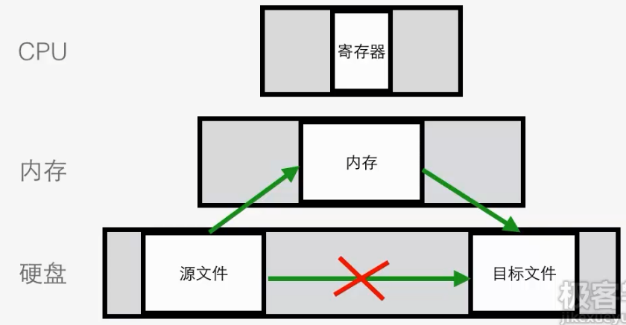
}

### 关闭流

释放IO库函数分配的资源，中断文件和程序的连接。

## 操作

### 文件拷贝



硬盘是不可寻址的，所以无法在硬盘内部直接进行数据的复制，需要借助内存寻址。

示例：

#include <stdio.h>

#include <stdlib.h> //man free, man malloc

int main(int argc,char \*\*argv)

{

int ret = -1;

//拷贝 源文件（source.c）到目标文件（dst.c)

FILE \*fp\_s = NULL;//源文件流

fp\_s = fopen("source.c","r");//只读 r

if ( NULL == fp\_s )

{

perror("fopen source.c");

return -1;

}

FILE \*fp\_d = NULL;//目标文件流

fp\_d = fopen("dst.c","w");//写w

if ( NULL == fp\_d )

{

perror("fopen dst.c");

return -1;

}

//准备内存

char \*buff = NULL;

buff = malloc(20);//堆 20字节

if( NULL == buff )

{

perror("malloc buff");

return -1;

}

//读

//从源文件默认光标（文件位置处）读取一个长度信息，存放到用户内存里面

ret = fread(buff,1,20,fp\_s);

if ( 0 == ret )

{

//bug...

perror("fread");

return -1;

}

//写

//

ret = fwrite(buff,1,20,fp\_d);

if ( 0 == ret )

{

//bug...

perror("fwrite");

return -1;

}

free(buff);

buff = NULL;

fclose(fp\_s);

fp\_s = NULL;

fclose(fp\_d);

fp\_d = NULL;

}

### 读写

#### fgets

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

可以处理行信息，与fputs成对使用。fread/fwrite是操作任意大小，fgets/fputs是操作行大小

#### fputs

#### getc

getc/putc操作的是字节

#### putc

#### fread

原型：size\_t fread(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

参数：ptr文件指针（没有操作指针位置的参数，即只能接收已经设置好的文件指针，即只能从当前光标处操作），size\*nmemb表示需要读取的文件大小（不应该超过用户空间分配的内存大小，否则容易产生内存越界的风险）

返回值：成功—读取的数据量，失败—比较短的数据值（没有错误，但是快到文件尾部的时候返回，比如总共30字节，一次读20字节，读两次会超出10，返回10）或0

当遇到失败的时候，应该调用feof(3)和ferror(3)判断是哪一种错误，feof表示是否到达文件尾部，ferror用来判断错误。

示例：

#include <stdio.h>

#include <stdlib.h> //man free, man malloc

#include <string.h>

int main(int argc,char \*\*argv)

{

int ret = -1;

//拷贝源文件（source.c）到目标文件（dst.c)

FILE \*fp\_s = NULL;//源文件流

fp\_s = fopen("source.c","r");//只读 r

if ( NULL == fp\_s )

{

perror("fopen source.c");

return -1;

}

FILE \*fp\_d = NULL;//目标文件流

fp\_d = fopen("dst.c","w");//写w

if ( NULL == fp\_d )

{

perror("fopen dst.c");

return -1;

}

//准备内存

char \*buff = NULL;

buff = malloc(20);//堆 20字节

if( NULL == buff )

{

perror("malloc buff");

return -1;

}

while(1)

{

memset(buff,'\0',20);//内存每次完成后清0（防止该次读取20，下次读取10，导致下次读取的时候有第一次的10个残留信息）

//读

//从源文件默认光标（文件位置处）读取一个长度信息，存放到用户内存里面

ret = fread(buff,1,20,fp\_s);

if ( 20 > ret )

{

if ( feof(fp\_s))//ferror

{

//文件尾

ret = fwrite(buff,1,ret,fp\_d);

if ( 0 == ret )

{

//bug...

perror("fwrite");

return -1;

}

break;

}else

{

//出错

perror("fread");

return -1;

}

}

//写

//

ret = fwrite(buff,1,ret,fp\_d);

if ( 0 == ret )

{

//bug...

perror("fwrite");

return -1;

}

}

free(buff);

buff = NULL;

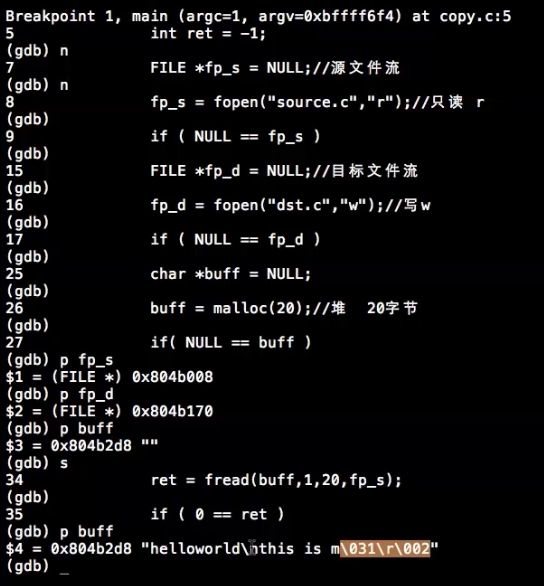
fclose(fp\_s);

fp\_s = NULL;

fclose(fp\_d);

fp\_d = NULL;

}

 GDB跟踪后分析：

在source中有20字节，buffer提供的是20字节大小，但是打印buffer的时候显示20字节后为什么后面还有别的信息？

buff填充的时候没有字符串结束符，即m字符后面是没有字符串结束符填充的，系统在解析buff的时候，认为buffer是个字符串，只有遇到结束符的时候才会停止，并不理解这段内存到底多大，所以读到buff的m时还会继续往下读，直到结束符停止。

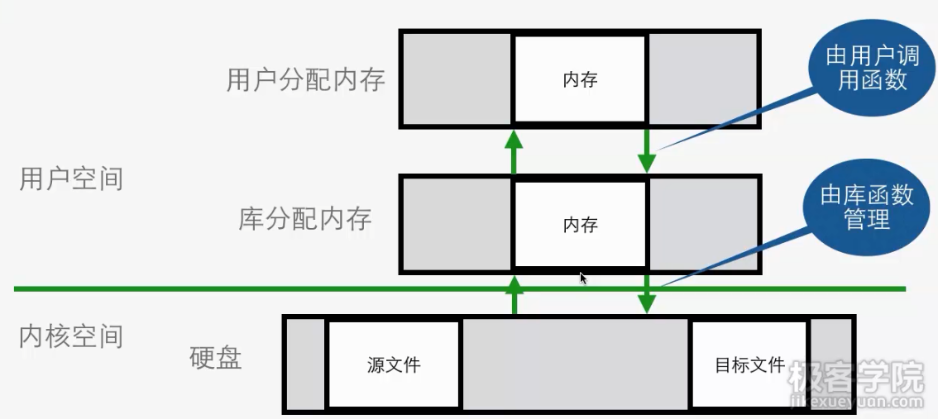
#### fwrite

原型：size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

#### 文件结束函数

#### 错误指示函数

### 缓冲



上述第一个过程用户调用函数是用户来操作的，第二个过程是由系统完成的，这种用内存代替机械硬盘效率高，但是二者存在数据同步不及时（即数据存在于内存中，系统崩溃，没有将数据刷新到磁盘，产生脏数据）。

示例：

#include <stdio.h>

#include <stdlib.h> //man free, man malloc

#include <string.h>

int main(int argc,char \*\*argv)

{

int ret = -1;

//拷贝 源文件（source.c）到目标文件（dst.c)

FILE \*fp\_s = NULL;//源文件流

fp\_s = fopen("source.c","r");//只读 r

if ( NULL == fp\_s )

{

perror("fopen source.c");

return -1;

}

FILE \*fp\_d = NULL;//目标文件流

fp\_d = fopen("dst.c","w");//写w

if ( NULL == fp\_d )

{

perror("fopen dst.c");

return -1;

}

//准备内存

char \*buff = NULL;

buff = malloc(20);//堆 20字节

if( NULL == buff )

{

perror("malloc buff");

return -1;

}

while(1)

{

memset(buff,'\0',20);//内存每次完成后清0

//读

//从源文件的默认光标（文件位置处）读取一个长度信息，存放到用户内存里面

ret = fread(buff,1,20,fp\_s);

if ( 20 > ret )

{

if ( feof(fp\_s))//ferror

{

//文件尾

ret = fwrite(buff,1,ret,fp\_d);

if ( 0 == ret )

{

//bug...

perror("fwrite");

return -1;

}

break;

}else

{

//出错

perror("fread");

return -1;

}

}

//写

//

ret = fwrite(buff,1,ret,fp\_d);

if ( 0 == ret )

{

//bug...

perror("fwrite");

return -1;

}

fflush(fp\_d);//主动将库函数内存与磁盘强制同步，防止脏数据

while(1);//ctrl+c 程序异常退出（查看数据有没有存到磁盘，此时显示未存到磁盘，由于写到库的内存，但是库内存没有完成与磁盘的同步）

}

free(buff);

buff = NULL;

fclose(fp\_s);

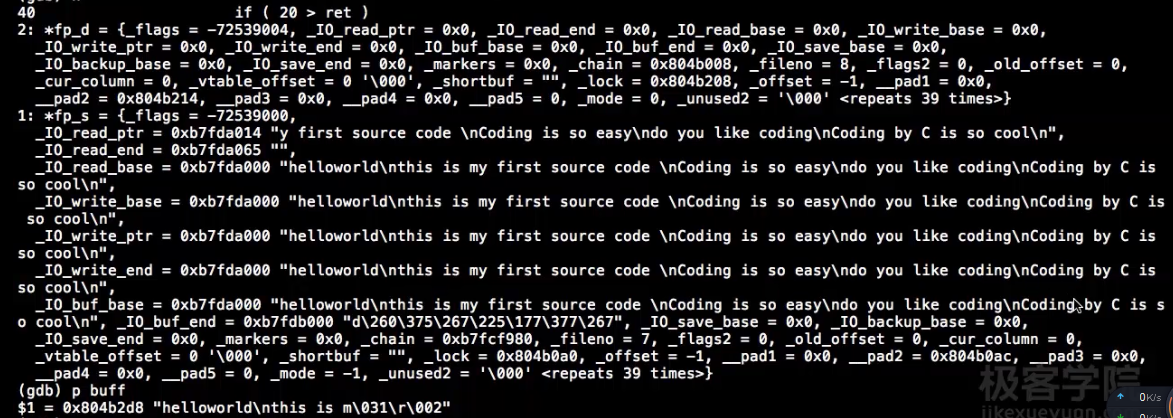
fp\_s = NULL;

fclose(fp\_d);

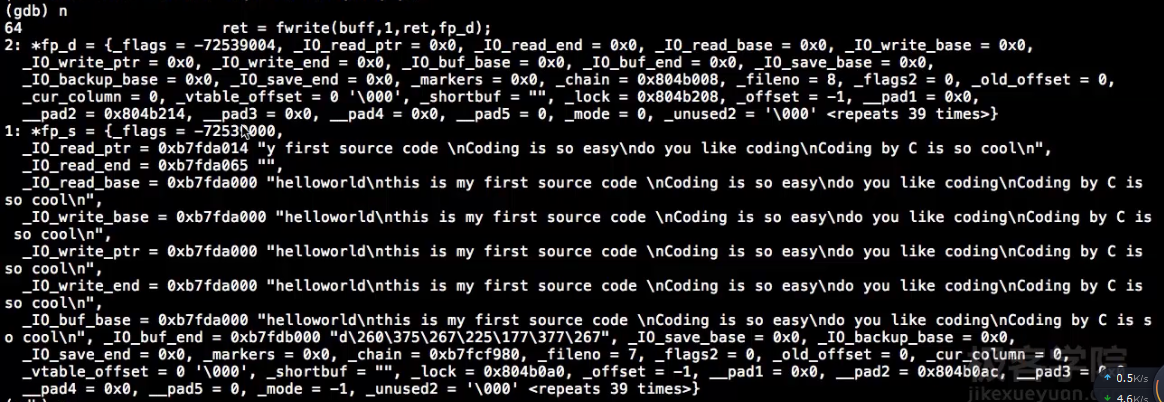
fp\_d = NULL;

}

分析：



文件open之后，打印FILE结构体就已经初始化了（关注IO\_read/IO\_write/IO\_buff的成员变量）。调用fread的时候，观察源文件的流\*fp\_s，可以看到这里缓冲的数据非常多，但是读出来的数据，即buff内容非常少。



当执行完write的时候，目标文件流会变化。

### 定位

在Linux中通过man fseek获取定位函数：

#### fseek

原型：int fseek(FILE \*stream, long offset, int whence);

参数：offset表示偏移量，whence表示具体移动到某个标志性位置（可以是文件头，文件尾或者当前位置）。

作用：针对某个标志性位置做偏移

举例：

#include <stdio.h>

#include <stdlib.h> //man free, man malloc

#include <string.h>

int main(int argc,char \*\*argv)

{

//一次性完成数据的传递

//计算出源文件的大小

//准备一个合适的内存，大小（源文件本身大小）

//一次性读取源文件

//一次性写入源文件

//关闭

int ret = -1;

//拷贝源文件（source.c）到目标文件（dst.c)

FILE \*fp\_s = NULL;//源文件流

fp\_s = fopen("source.c","r");//只读 r

if ( NULL == fp\_s )

{

perror("fopen source.c");

return -1;

}

FILE \*fp\_d = NULL;//目标文件流

fp\_d = fopen("dst.c","w");//写w

if ( NULL == fp\_d )

{

perror("fopen dst.c");

return -1;

}

//准备内存

char \*buff = NULL;

//计算出源文件的大小

ret = fseek(fp\_s,0,SEEK\_END);//定位到文件尾部

if ( -1 == ret )

{

perror("fseek");

return -1;

}

long file\_size;

file\_size = ftell(fp\_s);//计算出大小

if ( -1 == file\_size )

{

perror("ftell");

return -1;

}

//文件位置－－－文件头

ret = fseek(fp\_s,0,SEEK\_SET);

if ( -1 == ret )

{

perror("fseek");

return -1;

}

//准备一个合适的内存，大小（源文件本身大小）

buff = malloc(file\_size);//堆

if( NULL == buff )

{

perror("malloc buff");

return -1;

}

memset(buff,'\0',file\_size);//内存每次完成后清0

//读

//从源文件默认光标（文件位置处）读取一个长度信息，存放到用户内存里面

ret = fread(buff,1,file\_size,fp\_s);

if ( 20 > ret )

{

if ( feof(fp\_s))//ferror

{

//文件尾

ret = fwrite(buff,1,ret,fp\_d);

if ( 0 == ret )

{

//bug...

perror("fwrite");

return -1;

}

}else

{

//出错

perror("fread");

return -1;

}

}

//写

//

ret = fwrite(buff,1,ret,fp\_d);

if ( 0 == ret )

{

//bug...

perror("fwrite");

return -1;

}

free(buff);

buff = NULL;

fclose(fp\_s);

fp\_s = NULL;

fclose(fp\_d);

fp\_d = NULL;

}

#### ftell

原型：long ftell(FILE \*stream);

作用：测试当前光标位置距离文件头的位置（该距离使用long类型标识，单位是按照字节计算）。

可以利用fseek定位到文件头，然后ftell计算偏移量，这样就可以计算出文件的大小。

#### rewind

#### fgetpos

#### fsetpos

# 高级IO

## 非阻塞IO

在读某些文件时，如果文件没有数据，往往会导致读操作阻塞（休眠），比如：

1. 读鼠标、键盘灯字符设备文件；
2. 读管道文件（有名/无名）

读普通文件时，如果读到了数据就成功返回，如果没有读到数据返回0，总之不会阻塞。

在写某些文件时，当文件不能立即接收写入的数据时，也可能会导致写操作阻塞，一直阻塞到写成功为止。一般来说，写文件不会阻塞，因此我们不考虑写文件阻塞的情况。

由于一般情况下，“读鼠标”和“读键盘”都是阻塞的，为了不要让“读鼠标”和“读键盘”因为阻塞而相互干扰，可以采取如下办法读：

1. fork子进程，然后父子进程两线任务

父进程：读键盘

子进程：读鼠标

这种方式肯定是没问题的，这仅是一种方案，实际并不提倡这么做，多线任务时使用多进程实现，开销太大了。

1. 创建次线程，主线程和次线程两线任务

主线程：读键盘

次线程：读鼠标

这种方式是我们经常实现的方式。

1. 将鼠标和键盘设置为“非阻塞”，while轮询读
2. IO多路复用

**阻塞是好还是坏？**

实际上读文件因为没有数据而阻塞，其实是好事，因为这样就进入休眠状态，休眠时就不会占用CPU，节省了CPU的资源。

我们可以将阻塞的读修改为非阻塞的读。非阻塞读意思就是，如果有数据就成功读到，如果没有读到数据就出错返回，而不是阻塞。

尽管我们很少非阻塞的读，但是有些时候还是需要非阻塞读，因此OS还是提供了非阻塞操作方式。

**如何实现非阻塞读？**

1. 打开文件时指定O\_NONBLOCK状态标志；、

在IPC有名管道，如果不希望阻塞的话，就可以在open打开“有名管道”时，指定O\_NONBLOCK，然后读有名管道无数据时就不会阻塞。

1. 通过fcntl函数指定O\_NONBLOCK来实现；

什么情况下使用fcntl来实现：

1. 当文件已经被open打开了，但是open是并没有指定你要的文件状态标志，而你又不想去修改open的参数，此时可以使用fcntl来重设或者补设；
2. 没办法在open指定，你手里只有一个文件描述符fd，此时就使用fcntl来重设或补设：比如无名管道，无名管道连名字都没有，没有办法使用open函数，无名管道是使用pipe函数来返回文件描述符的，如果你想非阻塞的读无名管道的话，是没有办法通过open来指定O\_NONBLOCK的，此时就需要使用fcntl来重设或补设。

当然我们使用fcntl不仅仅只能重设或补设O\_NONBLOCK，也可以重设或补设O\_TEUNC/O\_APPEND等任何你需要的“文件状态”标志。

设置两种方式：

1. 重设

fcntl(0, F\_SETFL, O\_READONLY|O\_NONBLOCK);

1. 补设

flag = fcntl(0, F\_GETFL); //获取原有文件状态标志

flag = O\_NONBLOCK; //通过|操作，在已有标志上增加O\_NONBLOCK

fcntl(0, F\_SETFL, flag);

## 记录锁/文件锁

文件锁也被称为记录锁，是用来保护文件数据的。

当多个进程共享读写同一个文件时，为了不让进程各自读写数据时相互干扰，我们可以使用进程信号量来互斥实现，除了可以使用进程信号量以外，还可以使用“文件锁”来实现，而且功能更加丰富，使用相对容易。

多进程共享读写同一文件时，如果数据很重要的话，为了防止数据相互修改，应该满足如下读写条件：

1. 写与写应该互斥

当某个进程正在写文件，而且在数据没有写完时，其他进程不能写，否则会相互打乱对方写的数据。

1. 读与写应该互斥

分两种情况：

1. 某个进程正在写操作，而且在数据没有写完时，其他进程不能读数据。因为别人在没有写完之前，读到的数据是不完整的，所以读和写是互斥的。
2. 某个进程正在读数据，在数据没有读完之前，其他进程不能写数据。因为可能会扰乱别人读到的数据。
3. 读与读共享

某个进程在读数据时，就算数据没有读完，其他进程也可以共享读数据，并不需要互斥等待别人读完后才能读取。因为读文件时不会修改文件的内容，所以不用担心数据相互干扰的问题。

总结起来就是，多进程读写文件时，如果你想进行资源保护的话，完美的资源保护应该满足以下条件：

1. 写与写之间互斥；
2. 读与写之间互斥；
3. 读与读之间共享。

如何实现以上读写要求？

如果使用信号量实现保护的话，只能是一律互斥，包括读与读都是互斥的，不能够同时实现互斥和共享，但是文件锁可以做到。

## IO多路复用

多线程需要进行上下文切换，故而引入IO多路复用。

## 异步IO

## 存储映射