

## 4.1 一切皆文件

### 4.1.1 文件的概念

在 Linux 中，有一句经典的话叫做：一切皆文件。这句话是站在内核的角度说的，因为在内核中所有的设备（除了网络接口）都一律使用 Linux 独有的虚拟文件系统（VFS）来管理。这样做的最终目的，是将各种不同的设备用“文件”这个概念加以封装和屏蔽，简化应用层编程的难度。文件，是 Linux 系统最重要的两个抽象概念之一（另一个是进程）。

另外，VFS 中有个非常重要的结构体叫 `file{}`，这个结构体中包含一个非常重要的成员叫做 `file_operation`，他通过提供一个统一的、包罗万象的操作函数集合，来统一规范将来文件所有可能的操作。某一种文件或设备所支持的操作都是这个结构体的子集。做 Linux 底层开发的人对该结构体都应该非常熟悉。

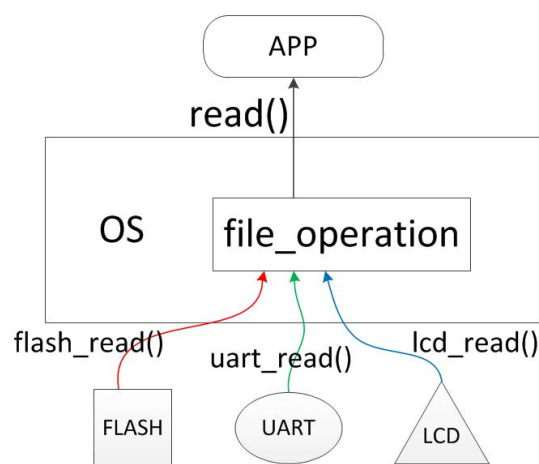


图 4-1 从应用层的 `read()` 到底层的 `xxx_read()`

图 4-1 以 `read()` 为例子，说明了为什么在上层应用中可以对千差万别的设备进行读操作，头号功臣就是 `file_operation` 提供了统一的接口，实际上，VFS 不仅包括 `file` 结构体，还有 `inode` 结构体和 `super_block` 结构体，正是他们的存在，应用层程序才得以摆脱底层设备的差异细节，独立于设备之外。

可以看到，内核做了掐头去尾的事情，提供了一个沟通上下的框架，如果你是软件工程师，就站在用户空间使用下面内核提供的接口，来为你的应用程序服务。如果你是底层驱动工程师，就站在操作硬件设备的角度，结合具体设备的操作方式，实现上面内核规定好的各个该设备可以支持的接口函数。

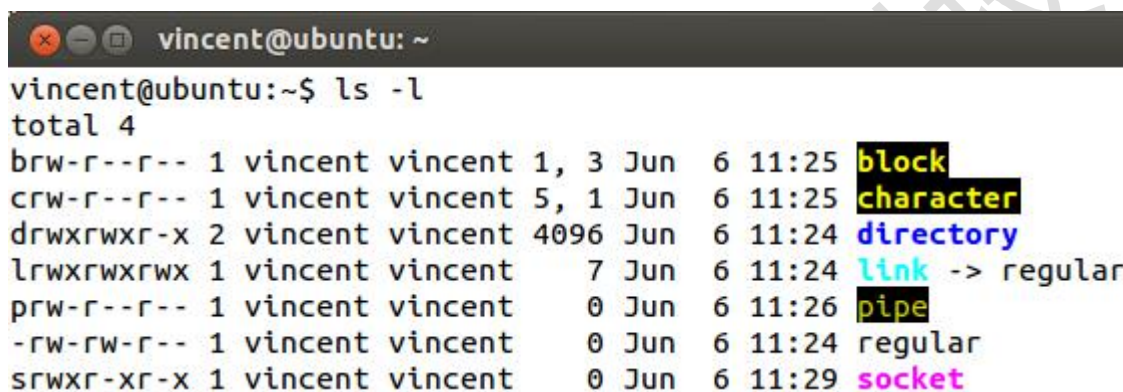
有了内核提供的中间层，我们在操作很多不同类型的文件的时候就方便多了，比方说读取文件 `a.txt` 的内容，跟读取触摸屏的坐标数据，读取鼠标的坐标信息等等，用的都是函数 `read()`，虽然底层的实现代码也许不一样，但是用户空间的进程并不关心也无需操心，Linux 的系统 IO 函数屏蔽了各类文件的差异，使得我们站在应用编程开发者的角度看下去，产生好像各类文件都一样的感觉。这就是 Linux 应用编程中一切皆文件的说法的由来。

### 4.1.2 各类文件

在 Linux 中，文件总共被分成了 7 种，他们分别是：

- 1，普通文件（**regular**）：存在于外部存储器中，用于存储普通数据。
- 2，目录文件（**directory**）：用于存放目录项，是文件系统管理的重要文件类型。
- 3，管道文件（**pipe**）：一种用于进程间通信的特殊文件，也称为命名管道 **FIFO**。
- 4，套接字文件（**socket**）：一种用于网络间通信的特殊文件。
- 5，链接文件（**link**）：用于间接访问另外一个目标文件，相当于 **Windows** 快捷方式。
- 6，字符设备文件（**character**）：字符设备在应用层的访问接口。
- 7，块设备文件（**block**）：块设备在应用层的访问接口。

下面是一张全家福：



```
vincent@ubuntu: ~  
vincent@ubuntu:~$ ls -l  
total 4  
brw-r--r-- 1 vincent vincent 1, 3 Jun 6 11:25 block  
crw-r--r-- 1 vincent vincent 5, 1 Jun 6 11:25 character  
drwxrwxr-x 2 vincent vincent 4096 Jun 6 11:24 directory  
lrwxrwxrwx 1 vincent vincent 7 Jun 6 11:24 link -> regular  
prw-r--r-- 1 vincent vincent 0 Jun 6 11:26 pipe  
-rw-rw-r-- 1 vincent vincent 0 Jun 6 11:24 regular  
srwxr-xr-x 1 vincent vincent 0 Jun 6 11:29 socket
```

图 4-2 Linux 的 7 种文件类型

注意到，每个文件信息的最左边一栏，是各种文件类型的缩写，从上到下依次是：

- b (block) 块设备文件
- c (character) 字符设备文件
- d (directory) 目录文件
- l (link) 链接文件（软链接）
- p (pipe) 管道文件（命名管道）
- (regular) 普通文件
- s (socket) 套接字文件（Unix 域/本地域套接字）

其中，块设备文件和字符设备文件，是 Linux 系统中块设备和字符设备的访问节点，在内核中注册了某一个设备文件之后，还必须在 **/dev/** 下为这个设备创建一个对应的节点文件（网络接口设备除外），作为访问这个设备的入口。目录文件用来存放目录项，是实现文件系统管理的最重要的手段。链接文件指的是软链接，是一种用来指向别的文件的特殊文件，其作用类似于 **Windows** 中的快捷方式，但他有更加有用的功能，比如库文件的版本管理。普通文件指的是外部存储器中的文件，比如二进制文件和文本文件。套接字文件指的是本机内进程间通信用的 **Unix** 域套接字，或称本地域套接字。

各种文件在后续的章节中都会一一涉及。

## 4.2 文件操作

对一个文件的操作有两种不同的方式，既可以使用由操作系统直接提供的编程接口（API），即系统调用，也可以使用由标准 C 库提供的标准 IO 函数，他们的关系如图 4-3 所示。

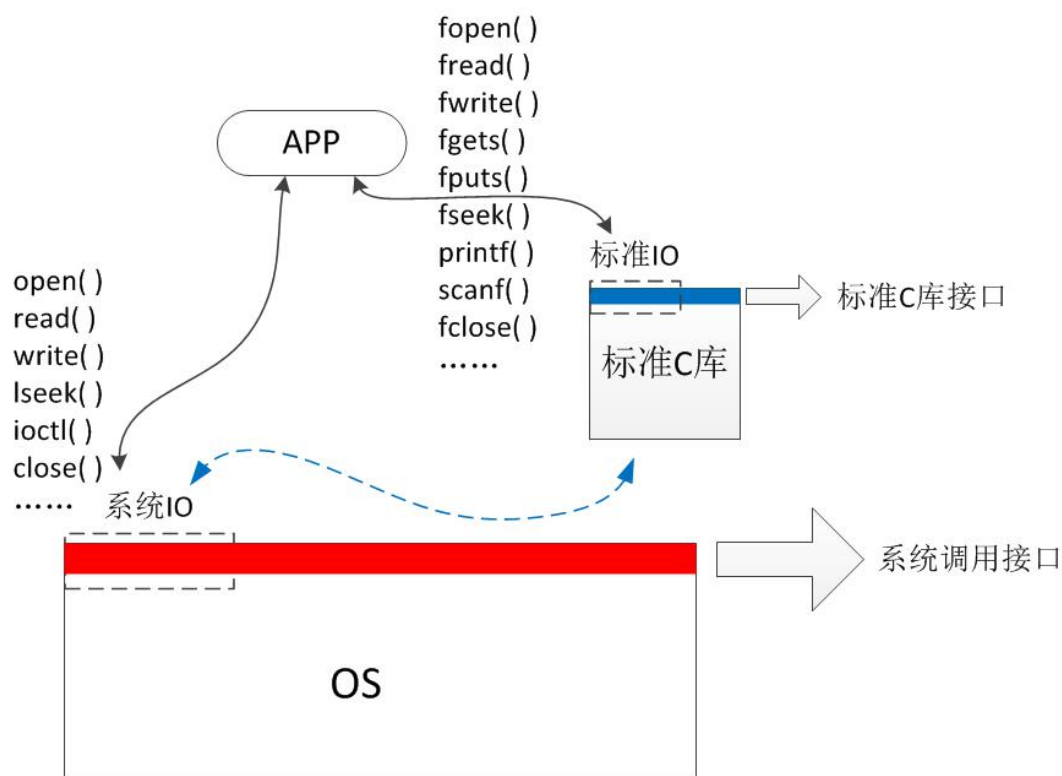


图 4-3 标准 IO 和系统 IO 的位置

在 Linux 操作系统中，应用程序的一切行为都依赖于这个操作系统，但是操作系统的内部函数应用层的程序是不能直接访问的，因此操作系统 OS 提供了大约四五百个接口函数，叫做“系统调用接口”，好让应用程序通过他们使用内核提供的各种服务，上图中用红色标注的那一层，就是这所谓的系统调用接口，这几百个函数是非常精炼的（Windows 系统的接口函数有数千个），他们以功能的简洁单一为美，以健壮稳定为美，但考虑到用户可能需要用到更加丰富的功能，因此就开发了很多库，其中最重要的也是应用程序必备的库就是标准 C 库，库里面的很多函数实际上都是对系统调用函数的进一步封装而已，用个比喻来讲就是：OS 的系统调用接口类似于菜市场，只提供最原始的肉菜，而库的函数接口相当于饭馆，对肉菜进行了加工，提供风味各异品种丰富的更方便食用的佳肴。

在几百个 Linux 系统调用中，有一组函数是专门针对文件操作的，比如打开文件、关闭文件、读写文件等，这些系统调用接口就被称为“系统 IO”，相应地，在几千个标准 C 库函数中，有一组函数也是专门针对文件操作的，被称为“标准 IO”，他们是工作在不同层次，但都是为应用程序服务的函数接口。

下面我们来逐一对系统 IO 函数和标准 IO 函数中最重要最常用的接口进行详细剖析，理解他们的异同，以便于在程序中恰当地使用他们。

4.2.1 系统 IO

要对一个文件进行操作，首先必须“打开”他，打开两个字之所以加上冒号，是因为这是代码级别的含义，并非图形界面上所理解的“双击打开”一个文件，代码中打开一个文件意味着获得了这个文件的访问句柄（即 **file descriptor**，文件描述符 **fd**），同时规定了之后访问这个文件的限制条件。

我们使用以下系统 IO 函数来打开一个文件：

功能	打开一个指定的文件并获得文件描述符，或者创建一个新文件	
头文件	#include <sys/types.h> #include <sys/stat.h> #include <fcntl.h>	
原型	int <b>open</b> (const char *pathname, int flags); int <b>open</b> (const char *pathname, int flags, mode_t mode);	
参数	pathname:	即将要打开的文件
	flags	O_RDONLY: 只读方式打开文件
		O_WRONLY: 只写方式打开文件
		O_RDWR: 读写方式打开文件
		这三个参数互斥
		O_CREAT: 如果文件不存在，则创建该文件。
		O_EXCL: 如果使用 O_CREAT 选项且文件存在，则返回错误消息。
		O_NOCTTY: 如果文件为终端，那么终端不可以作为调用 open()系统调用的那个进程的控制终端。
		O_TRUNC: 如文件已经存在，则删除文件中原有数据。
返回值	mode	O_APPEND: 以追加方式打开文件。
	成功	如果文件被新建，指定其权限为 mode（八进制表示法）
	失败	大于等于 0 的整数（即文件描述符）
		-1
备注	无	

表 4-1 函数 open( )的接口规范

使用系统调用 open( )需要注意的问题有：

- 1， flags 的各种取值可以用位或的方式叠加起来，比如创建文件的时候需要满足这样的选项：读写方式打开，不存在要新建，如果存在了则清空他。那么此时指定的 flags 的取值应该是：O\_RDWR | O\_CREAT | O\_TRUNC。
- 2， mode 是八进制权限，比如 0644，或者 0755 等。
- 3，它可以用来打开普通文件、块设备文件、字符设备文件、链接文件和管道文件，但只能用来创建普通文件，每一种特殊文件的创建都有其特定的其他函数。
- 4，其返回值就是一个代表这个文件的描述符，是一个非负整数。这个整数将作为以后任何系统 IO 函数对其操作的句柄，或称入口。

以下的系统 IO 函数用来关闭一个文件：

功能	关闭文件并释放相应资源	
头文件	#include <unistd.h>	
原型	int <b>close</b> (int fd);	
参数	fd: 即将要关闭的文件的描述符	
返回值	成功	0
	失败	-1
备注	重复关闭一个已经关闭了的文件或者尚未打开的文件是安全的。	

表 4-2 函数 close( ) 的接口规范

系统调用 **close( )** 相对来讲简单得多，只需要提供已打开的文件描述即可。一般来讲，当我们使用完一个文件之后，需要及时对其进行关闭，以防止内核为继续维护它而付出不必要的代价。

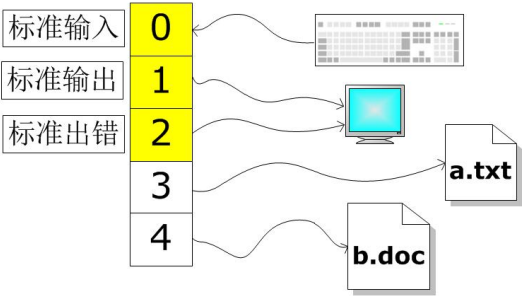


图 4-4 文件描述符与文件

上面的示意图表示，每一个被打开的文件（键盘、显示器都是文件）都会有一个非负的描述符来对应他们，一个文件还可以被重复打开多次，每打开一次也都会有一个描述符对应，并且可以有不同的模式。

那么，这个所谓的文件描述符究竟是什么玩意儿呢？其实他是一个数组的下标值，在 4.1 节中提到过，在内核中打开的文件是用 **file** 结构体来表示的，每一个结构体都会有一个指针来指向他们，这些指针被统一存放在一个叫做 **fd\_array** 的数组当中，而这个数组被存放在一个叫做 **files\_struct** 的结构体中，该结构体是进程控制块 **task\_struct** 的重要组成部分。他们的关系如图 4-5 所示。

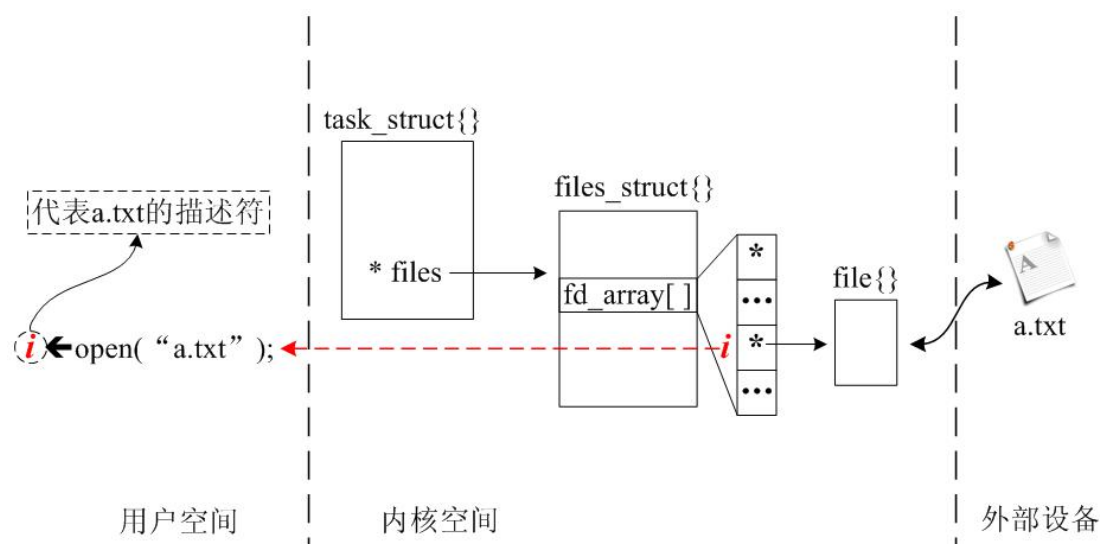


图 4-5 文件描述符的本质

上图中 **task\_struct** 被称为进程控制块（Process Control Block），是程序运行时在内核中的实现形式，在下一章有详细剖析。它里面包含了一个进程在运行时的所有信息，当然就包括了进程在运行过程中所打开文件的信息，这些信息被一个 **files** 指针加以统一的管理，**files** 指针所指向的结构体 **files\_struct{}** 里面的数组 **fd\_array[ ]** 是一个指针数组，用户空间每一次调用 **open( )** 都会使得内核实例化一个 **file{}** 结构体并将一个指向该结构体的指针依次存放在 **fd\_array[ ]** 中，并且该指针所占据的数组下标将作为所谓的“文件描述符（file descriptor）”返回给用户空间的调用者，这就是为什么文件描述符是非负整数的原因。

在用户空间使用系统 **IO** 编写应用程序的时候，也许无需了解这些内核数据和原理，但是知道一些细节，明白打开一个文件的本质内涵，无疑是分析复杂代码的有力保障，比如在多进程多线程操作文件的时候，比如编写设备驱动程序的测试代码的时候。因此推荐读者可以根据自己的情况，经常阅读内核源码，加深一些概念的代码级别的理解，接口是拳脚，原理是内功，最好内外兼修。

接下来是文件的读写接口：

功能	从指定文件中读取数据	
头文件	#include <unistd.h>	
原型	ssize_t read(int fd, void *buf, size_t count);	
参数	fd: 从文件 fd 中读数据	
	buf: 指向存放读到的数据的缓冲区	
	count: 想要从文件 fd 中读取的字节数	
返回值	成功	实际读到的字节数
	失败	-1
备注	实际读到的字节数小于等于 count	

功能	将数据写入指定的文件	
头文件	#include <unistd.h>	
原型	ssize_t <b>write</b> (int fd, const void *buf, size_t count);	
参数	fd:	将数据写入到文件 fd 中
	buf:	指向即将要写入的数据
	count:	要写入的字节数
返回值	成功	实际写入的字节数
	失败	-1
备注	实际写入的字节数小于等于 count	

图 4-6 函数 read( )和 write( )的接口规范

这两个函数都非常容易理解，需要特别注意的是：

1，实际的读写字节数要通过返回值来判断，参数 count 只是一个“愿望值”。

2，当实际的读写字节数小于 count 时，有以下几种情形：

A) 读操作时，文件剩余可读字节数不足 count

B) 读写操作期间，进程收到异步信号。

3，读写操作同时对 f\_pos 起作用。也就是说，不管是读还是写，文件的位置偏移量（即内核中的 f\_pos）都会加上实际读写的字节数，不断地往后偏移。

最后再介绍一个非常有用的系统 IO 接口：mmap( )。该函数在进程的虚拟内存空间中映射出一块内存区域，用以对应指定的一个文件，该内存区域上的数据跟对应的文件的数据是一一对应的，并在一开始的时候用文件的内容来初始化这片内存。

功能	内存映射	
头文件	#include <sys/mman.h>	
原型	void * <b>mmap</b> (void *addr, size_t length, int prot, int flags, int fd, off_t offset);	
参数	addr:	映射内存的起始地址。 如果该参数为 NULL，则系统将会自动寻找一个合适的起始地址，一般都使用这个值。 如果该参数不为 NULL，则系统会以此为依据来找到一个合适的起始地址。在 Linux 中，映射后的内存起始地址必须是页地址的整数倍。
	length:	映射内存大小。
	prot:	映射内存的保护权限。 PROT_EXEC：可执行。 PROT_READ：可读。 PROT_WRITE：可写。 PROT_NONE：不可访问。
	flags:	当有多个进程同时映射了这块内存时，该参数可以决定在某一个进程内使映射内存的数据发生变更是否影响其他进程，也可以决定是否影



		<p>响其对应的文件数据。</p> <p>以下两个选项互斥：</p> <p><b>MAP_SHARED</b>：所有的同时映射了这块内存的进程对数据的变更均可见，而且数据的变更会直接同步到对应的文件（有时可能还需要调用 <code>msync()</code> 或者 <code>munmap()</code> 才会真正起作用）。</p> <p><b>MAP_PRIVATE</b>：与 <b>MAP_SHARED</b> 相反，映射了这块内存的进程对数据的变更对别的进程不可见，也不会影响其对应的文件数据。</p> <p>以下选项可以位或累加：</p> <p><b>MAP_32BIT</b>：在早期的 64 位 x86 处理器上，设置这个选项可以将线程的栈空间设置在最低的 2GB 空间附近，以便于上下文切换时得到更好的表现性能，但现代的 64 位 x86 处理器本身已经解决了这个问题，因此这个选项已经被弃用了。</p> <p><b>MAP_ANON</b>：等同于 <b>MAP_ANONYMOUS</b>，已弃用。</p> <p><b>MAP_ANONYMOUS</b>：匿名映射。该选项使得该映射内存不与任何文件关联，一般来讲参数 <code>fd</code> 和 <code>offset</code> 会被忽略（但是可移植性程序需要将 <code>fd</code> 设置为 -1）。另外，这个选项必须跟 <b>MAP_SHARED</b> 一起使用。</p> <p><b>MAP_DENYWRITE</b>：很久以前，这个选项可以使得试图写文件 <code>fd</code> 的进程收到一个 <code>ETXTBUSY</code> 的错误，但是这很快成为所谓“拒绝服务”攻击的来源，因此现在这个选项也被弃用了。</p> <p><b>MAP_FIXED</b>：该选项使得映射内存的起始地址严格等于参数 <code>addr</code> 而不仅仅将 <code>addr</code> 当做参考值，这必须要求 <code>addr</code> 是页内存大小的整数倍，由于可移植性的关系，这个选项一般不建议设置。</p> <p><b>MAP_GROWSDOWN</b>：使得映射内存向下增长，即返回的是内存的最高地址，一般用于栈。</p> <p><b>MAP_HUGETLB</b>：使用“大页”来分配映射内存。关于“大页”请参考内核源代码中的 <code>Documentation/vm/hugetlbpage.txt</code>。</p> <p><b>MAP_NONBLOCK</b>：该选项必须与 <b>MAP_POPULATE</b> 一起使用，表示不进行“预读”操作。这使得选项 <b>MAP_POPULATE</b> 变得毫无意义，相信未来的某一天这两个选项会被修改。</p> <p><b>MAP_NORESERVE</b>：该选项旨在不为这块映射内存使用“交换分区”，也就是说当物理内存不足时，操作映射内存将会收到 <code>SIGSEGV</code>，而如果允许使用交换分区则可以保证不会因为物理内存不足而出现这个错误。</p> <p><b>MAP_POPULATE</b>：将页表映射至内存中，如果用于文件映射，该选项会导致“预读”的操作，因而在遇到页错误的时候也不会被阻塞。</p> <p><b>MAP_STACK</b>：在进程或线程的栈中映射内存。</p> <p><b>MAP_UNINITIALIZED</b>：不初始化匿名映射内存。</p>
	fd:	要映射的文件的描述符
	offset:	文件映射的开始区域偏移量，该值必须是页内存大小的整数倍，即必须是函数 <code>sysconf(_SC_PAGE_SIZE)</code> 返回值的整数倍。
返回值	成功	映射内存的起始地址
	失败	(void *) -1



备注	无
----	---

表 4-7 函数 `mmap()` 的接口规范

下面使用 `mmap()` 实现类似 Shell 命令 `cat` 的功能：将一个普通文件的内容显示到屏幕上。代码 `mmap.c` 演示了这个功能。

```
vincent@ubuntu:~/ch04/4.2$ cat mmap.c -n
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <errno.h>
5
6  #include <sys/stat.h>
7  #include <sys/mman.h>
8  #include <sys/types.h>
9  #include <fcntl.h>
10
11 int main(int argc, char **argv)
12 {
13     if(argc != 2)
14     {
15         printf("Usage: %s <filename>\n", argv[0]);
16         exit(1);
17     }
18
19     // 以只读方式打开一个普通文件
20     int fd = open(argv[1], O_RDONLY);
21
22     // 申请一块大小为 1024 字节的映射内存，并将之与文件 fd 相关联
23     char *p = mmap(NULL, 1024, PROT_READ,
24                     MAP_PRIVATE, fd, 0);
25
26     // 将该映射内存的内容打印出来（即其相关联文件 fd 的内容）
27     printf("%s\n", p);
28
29     return 0;
30 }
```

事实上，上面的示例没有什么实际的作用，甚至显得有点做作，因为我们极少使用 `mmap` 来读写普通文件的数据，更安全可靠且易懂的方式是 `read()`/`write()`，但是有些特殊文件只能用映射内存来读写，比如 6.3 节的 LCD 液晶屏。

4.2.2 标准 IO

系统 IO 的最大特点一个是更具通用性，不管是普通文件、管道文件、设备节点文件、套接字文件等等都可以使用，另一个是他的简约性，对文件内数据的读写在任何情况下都是不带任何格式的，而且数据的读写也都没有经过任何缓冲处理，这样做的理由是尽量精简内核 API，而更加丰富的功能应该交给第三方库去进一步完善。

标准 C 库是最常用的第三方库，而标准 IO 就是标准 C 库中的一部分接口，这一部分接口实际上是系统 IO 的封装，他提供了更加丰富的读写方式，比如可以按格式读写、按 ASCII 码字符读写、按二进制读写、按行读写、按数据块读写等等，还可以提供数据读写缓冲功能，极大提高程序读写效率。

在 4.2.1 中，所有的系统 IO 函数都是围绕所谓的“文件描述符”进行的，这个文件描述符由函数 open( )获取，而在这一节中，所有的标准 IO 都是围绕所谓的“文件指针”进行的，这个文件指针则是由 fopen( )获取的，他是第一个需要掌握的标准 IO 函数：

功能	获取指定文件的文件指针	
头文件	#include <stdio.h>	
原型	FILE *fopen(const char *path, const char *mode);	
参数	path:	即将要打开的文件
	mode	"r" : 以只读方式打开文件，要求文件必须存在。
		"r+" : 以读写方式打开文件，要求文件必须存在。
		"w" : 以只写方式打开文件，文件如果不存在将会创建新文件，如果存在将会将其内容清空。
		"w+" : 以读写方式打开文件，文件如果不存在将会创建新文件，如果存在将会将其内容清空。
		"a" : 以只写方式打开文件，文件如果不存在将会创建新文件，且文件位置偏移量被自动定位到文件末尾（即以追加方式写数据）。
		"a+" : 以读写方式打开文件，文件如果不存在将会创建新文件，且文件位置偏移量被自动定位到文件末尾（即以追加方式写数据）。
返回值	成功	文件指针
	失败	NULL
备注	无	

表 4-8 函数 fopen( )的接口规范

返回的文件指针是一种指向结构体 FILE{ }的指针，该结构体在标准 IO 中被定义：

```
vincent@ubuntu:/usr/include$ cat stdio.h -n
.....
46 __BEGIN_NAMESPACE_STD
47 /* The opaque type of streams. */
48 typedef struct _IO_FILE FILE; // 定义 FILE 等价于 _IO_FILE
49 __END_NAMESPACE_STD
.....
```

vincent@ubuntu:/usr/include\$ **cat libio.h -n**

```
.....
244
245 struct _IO_FILE {
246     int _flags; /* High-order word is _IO_MAGIC; rest is flags. */
247     #define _IO_file_flags _flags
248
249     // The following pointers correspond to the C++ streambuf protocol.
250     // Note: Tk uses the _IO_read_ptr and _IO_read_end fields directly.
251     char* _IO_read_ptr;
252     char* _IO_read_end;
253     char* _IO_read_base;
254     char* _IO_write_base;
255     char* _IO_write_ptr;
256     char* _IO_write_end;
257     char* _IO_buf_base;
258     char* _IO_buf_end;
259     /* The following fields are used to support backing up and undo. */
260     char *_IO_save_base;
261     char *_IO_backup_base;
262     char *_IO_save_end;
263
264     struct _IO_marker *_markers;
265
266     struct _IO_FILE *_chain;
267
268     int _fileno; // 文件描述符
269     #if 0
270     int _blksize;
271     #else
272     int _flags2;
273     #endif
274     _IO_off_t _old_offset;
275
276     #define __HAVE_COLUMN /* temporary */
277     /* 1+column number of pbase(); 0 is unknown. */
278     unsigned short _cur_column;
279     signed char _vtable_offset;
280     char _shortbuf[1];
281
282     /* char* _save_gptr; char* _save_egptr; */
283
284     _IO_lock_t *_lock;
```

```
285 #ifdef _IO_USE_OLD_IO_FILE
286 };
.....
```

观察上述代码，第 268 行中的 `_fileno` 就是打开的文件的描述符，被封装在了 `FILE{}` 里面，`FILE{}` 里面除了封装了由系统 IO 函数 `open()` 得来的 `_fileno` 之外，还提供了一组指针（第 251 行到第 262 行），用来管理数据缓冲区。

文件指针和文件描述符的关系如下图所示：

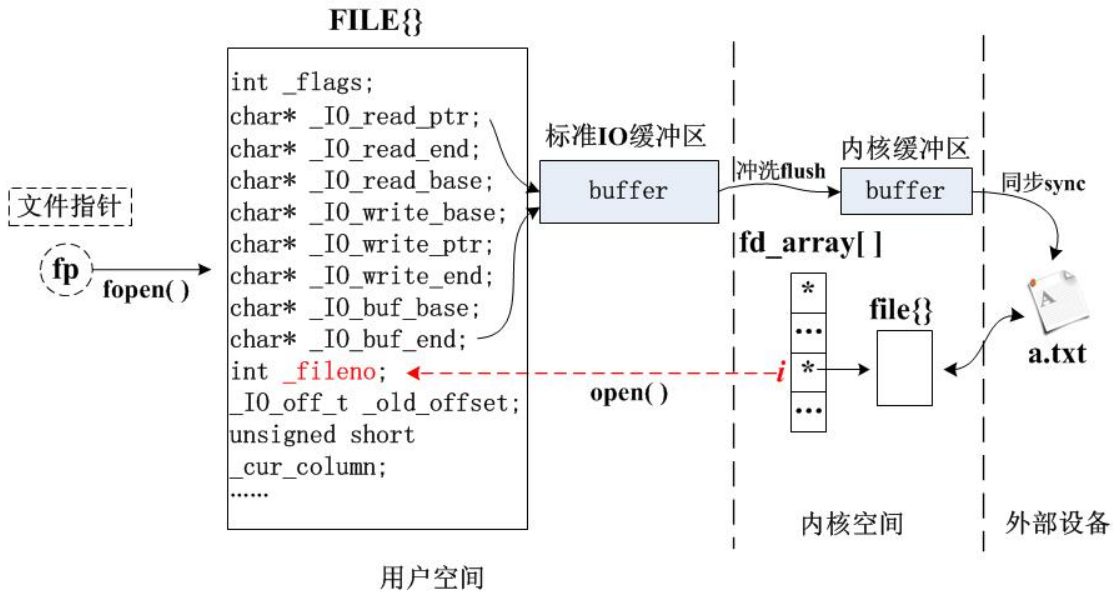


图 4-8 文件指针与文件描述符的关系

可以看到，使用标准 IO 函数处理文件的最大特点是，数据将会先存储在一个标准 IO 缓冲区中，而后在一定条件下才被一并 `flush`（冲洗，或称刷新）至内核缓冲区，而不是像系统 IO 那样，数据直接被 `flush` 至内核。

注意到，标准 IO 函数 `fopen()` 实质上是系统 IO 函数 `open()` 的封装，他们是一一对应的，每一次 `fopen()` 都会导致系统分配一个 `file{ }` 结构体和一个 `FILE{}` 来保存维护该文件的读写信息，每一次的打开和操作都可以不一样，是相对独立的，因此可以在多线程或者多进程中多次打开同一个文件，再利用文件空洞技术进行多点读写。

另外由上一节得知，标准输入输出设备是默认被打开的，在标准 IO 中也是一样，他们在程序的一开始就已经拥有相应的文件指针了：

设备	文件描述符 (int)	文件指针 (FILE *)
标准输入设备 (键盘)	0 STDIN_FILENO	stdin
标准输出设备 (屏幕)	1 STDOUT_FILENO	stdout
标准出错设备 (屏幕)	2 STDERR_FILENO	stderr

表 4-9 缺省打开的三个标准文件

跟 `fopen( )` 一起配套使用的是 `fclose( )`：

功能	关闭指定的文件并释放其资源	
头文件	#include <stdio.h>	
原型	int <b>fclose</b> (FILE *fp);	
参数	fp: 即将要关闭的文件	
返回值	成功	0
	失败	EOF
备注	无	

表 4-10 函数 `fclose( )` 的接口规范

该函数用于释放由 `fopen( )` 申请的系统资源，包括释放标准 IO 缓冲区内存，因此 `fclose( )` 不能对一个文件重复关闭。

下面是他们的应用示例代码：

```
vincent@ubuntu:~/ch04/4.2$ cat fopen_fclose.c -n
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <sys/types.h>
5  #include <fcntl.h>
6
7  int main(int argc, char **argv)
8  {
9      FILE *fp = fopen("a.txt", "r+"); // 以读写方式打开已存在文件
10
11      // 如果打开文件"a.txt"失败，fopen( )将会返回 NULL
12      if(fp == NULL)
13      {
14          perror("fopen()");
15          exit(1);
16      }
17
18      // 如果关闭 fp 失败，fclose( )将会返回 EOF
19      if(fclose(fp) == EOF)
20      {
21          perror("fclose()");
22          exit(1);
23      }
24
25      return 0;
26 }
```

标准 IO 函数的读写接口非常多，下面逐一列出最常用的各个函数集合：

第一组：每次一个字符的读写标准 IO 函数接口。

功能	获取指定文件的一个字符	
头文件	#include <stdio.h>	
原型	int <b>fgetc</b> (FILE *stream); int <b>getc</b> (FILE *stream); int <b>getchar</b> (void);	
参数	stream: 文件指针	
返回值	成功	读取到的字符
	失败	EOF
备注	当返回 EOF 时，文件 stream 可能已达末尾，或者遇到错误	

功能	讲一个字符写入一个指定的文件	
头文件	#include <stdio.h>	
原型	int <b>fputc</b> (int c, FILE *stream); int <b>putc</b> (int c, FILE *stream); int <b>putchar</b> (int c);	
参数	c: 要写入的字符	
	stream: 写入的文件指针	
返回值	成功	写入到的字符
	失败	EOF
备注	无	

表 4-11 每次读写一个字符的函数接口规范

需要注意的几点：

1, fgetc( )、getc( )和 getchar( )返回值是 int，而不是 char，原因是因为他们在出错或者读到文件末尾的时候需要返回一个值为 -1 的 EOF 标记，而 char 型数据有可能因为系统的差异而无法表示负整数。

2, 当 fgetc( )、getc( )和 getchar( )返回 EOF 时，有可能是发生了错误，也有可能是读到了文件末尾，这是要用以下两个函数接口来进一步加以判断：

功能	feof(): 判断一个文件是否到达文件末尾 ferror(): 判断一个文件是否遇到了某种错误	
头文件	#include <sys/ioctl.h>	
原型	int <b>feof</b> (FILE *stream); int <b>ferror</b> (FILE *stream);	
参数	stream: 进行判断的文件指针	
返回值	feof	如果文件已达末尾则返回真，否则返回假
	ferror	如果文件遇到错误则返回真，否则返回假
备注	无	

图 4-12 函数 feof( )和 ferror( )的接口规范

- 3, getchar( )缺省从标准输入设备读取一个字符。
- 4, putchar( )缺省从标准输出设备输出一个字符。
- 5, fgetc( )和 fputc( )是函数, getc( )和 putc( )是宏定义。
- 6, 两组输入输出函数一般成对地使用, fgetc( )和 fputc( ), getc( )和 putc( ), getchar( )和 putchar( )。

第二组：每次一行的读写标准 IO 函数接口。

功能	从指定文件读取最多一行数据	
头文件	#include <sys/ioctl.h>	
原型	char *fgets(char *s, int size, FILE *stream); char *gets(char *s);	
参数	s: 自定义缓冲区指针	
	size: 自定义缓冲区大小	
	stream: 即将被读取数据的文件指针	
返回值	成功	自定义缓冲区指针 s
	失败	NULL
备注	1, gets( )缺省从文件 stdin 读入数据 2, 当返回 NULL 时, 文件 stream 可能已达末尾, 或者遇到错误	

功能	将数据写入指定的文件	
头文件	#include <sys/ioctl.h>	
原型	int fputs(const char *s, FILE *stream); int puts(const char *s);	
参数	s: 自定义缓冲区指针	
	stream: 即将被写入数据的文件指针	
返回值	成功	非负整数
	失败	EOF
备注	puts( )缺省将数据写入文件 stdout	

表 4-13 每次读写一行的函数接口规范

- 值得注意的有以下几点：
- 1, fgets( )跟 fgetc( )一样, 当其返回 NULL 并不能确定究竟是达到文件末尾还是碰到错误, 需要用 feof( )/ferror( )来进一步判断。
  - 2, fgets( )每次读取至多不超过 size 个字节的一行, 所谓“一行”即数据至多包含一个换行符'\n'。
  - 3, gets( )是一个已经过时的接口, 因为他没有指定自定义缓冲区 s 的大小, 这样很容易造成缓冲区溢出, 导致程序段访问错误。
  - 4, fgets( )和 fputs( ), gets( )和 puts( )一般成对使用, 鉴于 gets( )的不安全性,



一般建议使用前者。

第三组：每次读写若干数据块的标准 IO 函数接口。

功能	从指定文件读取若干个数据块	
头文件	#include <sys/ioctl.h>	
原型	size_t <b>fread</b> (void *ptr, size_t size, size_t nmemb, FILE *stream);	
参数	ptr:	自定义缓冲区指针
	size:	数据块大小
	nmemb:	数据块个数
	stream:	即将被读取数据的文件指针
返回值	成功	读取的数据块个数，等于 nmemb
	失败	读取的数据块个数，小于 nmemb 或等于 0
备注	当返回小与 nmemb 时，文件 stream 可能已达末尾，或者遇到错误	

功能	将若干块数据写入指定的文件	
头文件	#include <sys/ioctl.h>	
原型	size_t <b>fwrite</b> (const void *ptr, size_t size, size_t nmemb, FILE *stream);	
参数	ptr:	自定义缓冲区指针
	size:	数据块大小
	nmemb:	数据块个数
	stream:	即将被写入数据的文件指针
返回值	成功	写入的数据块个数，等于 nmemb
	失败	写入的数据块个数，小于 nmemb 或等于 0
备注	无	

表 4-14 每次读写若干数据块的函数接口规范

这一组标准 IO 函数被称为“直接 IO 函数”或者“二进制 IO 函数”，因为他们对数据的读写严格按照规定的数据块数和数据块的大小来处理，而不会对数据格式做任何处理，而且当数据块中出现特殊字符，比如换行符'\n'、字符串结束标记'\0'等是不会受到影响。

需要注意的几点：

- 1，如果 **fread()** 返回值小于 **nmemb** 时，则可能已达末尾，或者遇到错误，需要借助于 **feof()**/**ferror()** 来加以进一步判断。
- 2，当发生上述第 1 种情况时，其返回值并不能真正反映其读取或者写入的数据块数，而只是一个所谓的“截短值”，比如正常读取 5 个数据块，每个数据块 100 个字节，在执行成功的情况下返回值是 5，表示读到 5 个数据块总共 500 个字节，但是如果只读到 499 个数据块，那么返回值就变成 4，而如果读到 99 个字节，那么 **fread()** 会返回 0。因此当发生返回值小于 **nmemb** 时，需要仔细确定究竟读取了几个字节，而不能直接从返回值确定。

第四组：获取或设置文件当前位置偏移量。

功能	设置指定文件的当前位置偏移量	
头文件	#include <sys/ioctl.h>	
原型	int <b>fseek</b> (FILE *stream, long offset, int whence);	
参数	stream: 需要设置位置偏移量的文件指针	
	offset: 新位置偏移量相对基准点的偏移	
	whence: 基准点	SEEK_SET: 文件开头处
		SEEK_CUR: 当前位置
		SEEK_END: 文件末尾处
返回值	成功	0
	失败	-1
备注	无	

功能	获取指定文件的当前位置偏移量	
头文件	#include <sys/ioctl.h>	
原型	long <b>ftell</b> (FILE *stream);	
参数	stream: 需要返回当前文件位置偏移量的文件指针	
返回值	成功	当前文件位置偏移量
	失败	-1
备注	无	

功能	将指定文件的当前位置偏移量设置到文件开头处	
头文件	#include <sys/ioctl.h>	
原型	void <b>rewind</b> (FILE *stream);	
参数	stream: 需要设置位置偏移量的文件指针	
返回值	无	
备注	该函数的功能是将文件 stream 的位置偏移量置位到文件开头处。	

表 4-15 调整文件位置偏移量的函数接口规范

这一组函数需要注意的几点是：

- 1, fseek( ) 的用法基本上跟系统 IO 的 lseek( ) 是一致的。
- 2, rewind(fp) 相等于 fseek(fp, 0L, SEEK\_SE);