主要3.1.

第三章：低级输入输出

当我们需要用很大的缓冲区来读取二进制文件，或需要对特定设备进行控制操作，或需要传递文件描述字给子进程(子进程可以用继承的文件描述字创建自己的流，但不能直接继承流)，就需要用低级I/O函数

3.1：文件描述字的打开，创建和关闭

Int open(const char\* filename, int flags[, mode\_t mode]);

Int create(const char\* filename, mode\_t mode);

flags给出文件的打开方式，它是一个位串，mode是可选参数，给出文件的访问方式。Open函数返回的描述字一定是当前最小没使用的描述字，比如当一个程序关闭啦它的标准输出，然后在调用这个open函数创建或者打开另一个文件，则标准输出描述字1将会成为这个新打开文件的描述字，从而使得标准输出可以重新定向到不同的文件或者设备。Opne返回的文件描述字是唯一的，他不会有正在运行的其他进程共享fd = create(filename, mode)等价于 fd = open(filename, O\_WRONLY|O\_CREAT|O\_TRUNC, mode);

int close(int filedes)

每一个活跃进程在系统的进程表中有一个登记项，称为进程表项，此表可以看成数组，每一个元素对应一个文件描述字，open函数返回的文件描述字就是该数组元素的索引。数组元素中记录啦文件描述字标签和指向系统打开文件的指针。

每当open成功打开一个文件时，便在系统打开文件表中建立一个打开文件表项。里面有文件状态标签，由open第二个参数指定，有当前文件位置和指向该文件的v-node的指针

vnode是内核中表示文件的数据结构，内核都通过与文件相连的vnode来访问文件。一个文件只有一个vnode结构，vnode包含vnode信息和文件的inode信息，inode中包含文件的属主，文件大小等等

如果两个独立进程打开同一个文件，这时候每个进程都有自己的系统打开文件表项，但它们的描述字都指向同一个vnode。及系统打开文件表的vnode指针相同

注意：1子进程总是继承父进程的所有描述字2：每一次write()完成之后，系统打开文件表项中当前文件位置将会增加所写的字节数3：文件描述字总是从最小可用描述字开始分配4：如果文件使用O\_APPEND标志打开的，该标志将被设置在系统文件打开表项的文件状态标签中5：如果一个文件位置被lseek()定位于当前文件尾，所发生的只是用inode中的当前文件大小设置系统打开文件表的当前文件位置6：一个进程或者多个进程的多个文件描述字可以指向同一个系统打开文件表项，这对应于单次打开文件但复制了多个描述字的情形，多个系统打开文件表项也可以指向同一个vnode结构，这对应于一个进程多次打开或者多个进程同时打开同一个文件的情形。

3.2：read和write

ssize\_t read(int filedes, void \*buffer, size\_t nbytes); ssize\_t write(int filledes, const void \*buffer, size\_t nbytes). 读至多nbytes个字节得数据放到buffer所指缓冲区中。正常返回得是实际读入得字节数，若在实际读入字节之前便遇到文件尾，其返回值为0.read调用成功得返回值大于等于0，遇到文件尾返回0.若在文件结束时继续调用read，他将返回0并无其他动作。write的返回值是实际写入的字节数，正常情况下等于nbytes，但也有可能小于(如被写的物理介质满时)

3.3：设置描述字的文件位置

off\_t lseek(int filedes, off\_t offset, int whence); offset指明相对whence的位移字节数，whence的取值和fseek一样只能是SEEK\_SET等等， n = lseek(fd, 0, SEEK\_CUR);是读当前文件位置值 。lseek只移动文件的当前位置，它并不引起任何的I/O动作，文件的新位置可以大于文件的当前大小，在这种情况下下一次写将会扩展该文件，这种情况也称为在文件中生成空洞。如果在数据尾部之后到lseek设置的偏移量之前这段区域的数据尚未写入，则后继读这段区域将读出0

ls -l file.hole 查看文件大小

od -c file,hole 查看文件的实际内容 od用于查看文件的内容，-c选项指定打印内容为字符

当同一个文件有多个打开的描述字的时候，由不同open得到的文件描述字具有独立的文件位置，lseek只作用于指定的描述字而不会对另外的描述字造成影响

{

int d1,d2;

char buf[4];

d1 = open(“foo”, O\_RDONLY);

d2 = open(“foo”, O\_RDONLY);

lseek(d1, 1024, SEEK\_SET);

read(d2, buf, 4);

}

read将读文件foo的前4个字符，但是dup重复而得的描述字与原描述字共享一个公共的文件位置，改变两个重复描述字中的一个文件位置，包括读或者写数据，同时也会改变另外一个描述字的位置文件

{

int d1,d2;

char buf1[80], buf2[80];

d1 = open(“foo”, O\_RDONLY);

d2 =dup(d1);

lseek(d1, 1024, SEEK\_SET);

read(d2, buf1, 4);

read(d2, buf2, 4);

}

第一个read将从foo的第1024个字符开始读4个字节，第二个read则从第1028个字符处再读4个字节

3.4：dup和dup2

重复一个描述字可以用函数dup和dup2或者函数fcntl int dup(int old); int dup2(int old, int new); dup复制描述字old至一个新描述字，新描述字是当前未打开的最小编号可用描述字。dp2复制描述字old至编号为new的描述字。如果new已经打开，它将首先被关闭，如果new等于old，dup2返回new但不关闭它

重复一个文件描述字的主要用途是实现输入输出的重定向，及改变一个特定文件描述字对应的文件或者管道。例子将标准输出文件重定向至名为myoutput的文件，运行这个程序可以看到printf()输出不在终端而在文件中

int main(void)

{

int fd;

if ((fd = open("myoutput", O\_WRONLY|)\_CREAT, 0644)) == -1)

exit(0);

if (dup2(fd, STDOUT\_FILENO) == -1)

exit(0);

printf("this is test program for redirect \n");

close(fd);

}

3.5：fdopen和fileno函数

文件描述字函数是流函数的初等函数，每一个流都与一个描述字相连，给定一个打开文件描述字，可以用fdopen为它创建一个流，反之，知道一个流，可以用fileno得到它的文件描述字

FILE \*fdopen(int filedes, const char \*opentype); int fileno(file \*stream);

不能创建一个流，则返回空指针，参数opentype的取值和fopen中的opentype取值完全相同，文件是早就打开了的，注意：fdopen的opentype参数必须和打开文件描述字时使用的参数opentype一致，fdopen建立的新流的文件位置与描述字filedes的文件位置相同，且流的错误指示器和文件结束指示器均被清除。fopen的实质是为已打开的文件描述字提供标准I/O缓冲

3.6：文件控制函数fcntl

作用：重复一个文件描述字，查询或者设置文件描述字标签，查询或设置文件描述字标签或者文件状态标签 int fcntl(int filedes, int cmd, …);具体是哪一种操作由cmd决定，由一些操作还要求第三个参数

3.6.1：重复文件描述字

fntl(filedes, F\_DUPFD, grg);它返回一个其值等于或者大于arg的新文件描述字 dup等价于fcntl(filedes, F\_DUPFD, 0);

3.6.2：文件描述字标签

该函数用于获取和设置文件描述字标签

3.6.3：文件状态标签

文件状态标签可以分为三类：访问方式，打开时标志和I/O操作方式

为咯设置描述字filedes的文件状态标签，用命令F\_SETFL调用fcntl，该函数要求int类型的第三个参数以指明新标签，fcntl(filedes, F\_SETFL, new\_flags);它的正常返回值是一个非-1的不确定值，-1指出出错，文件状态标签中，文件的访问方式是在打开文件时设定的，一旦设定就不能改变；打开时标志只能在打开时使用，之后便不再保留，因此应用唯一可以设置的是文件的I/O操作方式，及O\_APPEND,O\_NONBLOCK

函数在设置或者清除标志O\_NONBLOCK时不会改变其他标签

//如果value非0，设置desc的O\_NONBLOCK标志，否则清除该标志。成功返回0，失败返回-1

int set\_nonblock\_flag(int desc, int value)

{

int oldflags = fcntl(desc, F\_GETFL, 0);

if (oldflags == -1)

return -1;

//形成并设置新标签

if (value != 0)

oldflags |= O\_NONBLOCK;

else

oldflags &= ~O\_NONBLOCK;

return fcntl(desc, F\_SETFL, oldflags);

}

3.7：非阻塞I/O

3.8：readv和writev函数

read和write系统调用每次在文件和进程的地址空间之间传一块连续的数据。但是，有时候也需要将分散在内存多处地方的数据连续写到文件中，或者反之。这时候就要用到这两个函数，他们只需要一次系统调用就可以实现在文件和进程的多个缓冲区之间传送数据，免除了多次系统调用或者复制数据的开销。readv称为散布读，即将文件中若干连续的数据块读入内存分散的缓冲区中，writev称为散聚写，及收集内存中分散的若干缓冲区中的数据写至文件的连续区域 ssize\_t readv(int fildes, const struct iovec \*iov, int iovcnt); ssize\_t write(int fildes, const struct iovec \*iov, int iovcnt);，iov是一个结构数组，它的每一个元素指明存储器中的一个缓冲区，结构iovec分别给出缓冲区的起始地址和字节数，参数iovcnt指出数组iov的元素个数，writev依次将iov[0]，iov[1]等等的指定的存储区中的数据写至fildes指定的文件，返回写出的数据总字节数。正常下等于数据块长度之和，readv相反。当想要集中写出某张链表的时候，只需要让iov数组的各个元素包含链表中各个表项的地址和长度，然后将iov和他的元素个数作为参数传递给writev，这些数据便可一次写出

3.9：fsync和fdatasync函数

当我们调用write函数写出数据的时候，数据一旦写到缓冲区中，函数便立即返回，此时写出的数据可以被read函数读，也可以被其他进程读到，但是这不意味着这些数据已经被写到了外部永久存储介质上，即使调用close关闭文件后也可能如此，真正的传输动作由独立于CPU的外设控制器或者外设本身(LINUX称之为DMA引擎)来完成，write写出的数据和外部存储设备并不是完全同步的。

对于十分重要的数据，我们可以用一种方法是对文件设置O\_SYNC标志，保证每次写入的数据直接写到磁盘。如果设置了这个标志，write调用将直到数据已安全写到磁盘后(而不仅仅是系统的I/O缓冲区)才返回，但是这样每次写数据都保持同步的效率比较低。另一种方法：调用fsync的进程将阻塞直到设备报告传送已经完成，强制与描述字fildes相连文件的所有修改过的数据传送到外部永久介质，及刷新fildes给出的文件的所有信息。fdatasync只是强制传送用户已写出的数据至物理存储设备，不包括文件本身的特征数据，这样可以减少文件刷新时的数据传送量，在有些系统上，他们的功能完全一样。

数据库应用通常会在调用write保存关键的数据时也同时调用fsync

内核的I/O缓冲区是由操作系统管理的空间，而流缓冲区是由标准I/O库管理的用户空间，fflush只刷新位于用户空间的流缓冲区，刷新后只能保证数据不在流缓冲区中，并不能保证他们一定写在了磁盘，因为他们还可能呆在内核I/O缓冲区中，要确保写入磁盘，就的调用fflush后调用fsync