先来看看不带缓存的I/O和标准(带缓存的)I/O都有那些  
不带缓存的I/O:       **read,write,open......**  
标准(带缓存的)I/O: **fgets,fread,fwrite.....**

这里使用两个对应的函数进行比较：  
ssize\_t write(int filedes, const void \*buff, size\_t nbytes)  
size\_t fwrite(const void \*ptr, size\_t size, size\_t nobj, FILE \*fp)

上面的buff和ptr都是指应用程序自己使用的buffer，实际上当需要对文件进行写操作时，都会先写到内核所设的缓冲存储器。***如果该缓存未满，则并不将其排入输出队列，直到缓存写满或者内核再次需要重新使用此缓存时才将其排入磁盘I/O输入队列，再进行实际的I/O操作，也就是此时才把数据真正写到磁盘，这种技术叫延迟写。***

如果我们直接用非缓存I/O对内核的缓冲区进行读写，会产生许多管理不善而造成的麻烦（如一次性写入过多，或多次系统调用导致的效率低下）。

标准(带缓存的)I/O为我们解决了这些问题，它处理很多细节，如缓冲区分配，以优化长度执行I/O等，更便于我们使用。

由于标准(带缓存的)I/O在系统调用的上一层多加了一个缓冲区，也因此引入了流的概念，在UNIX/Linux下表示为FILE\*（并不限于UNIX/Linux，ANSI C都有FILE的概念），FILE实际上包含了为管理流所需要的所有信息：实际I/O的文件描述符，指向流缓存的指针（标准I/O缓存，由malloc分配，又称为用户态进程空间的缓存，区别于内核所设的缓存），缓存长度，当前在缓存中的字节数，出错标志等。

因此可知，**不带缓存的I/O对文件描述符操作，带缓存的标准I/O是针对流的**。

标准I/O对每个I/O流自动进行缓存管理（标准I/O函数通常调用malloc来分配缓存）。它提供了三种类型的缓存：  
      1) 全缓存。当填满标准I/O缓存后才执行I/O操作。磁盘上的文件通常是全缓存的。  
      2) 行缓存。当输入输出遇到新行符或缓存满时，才由标准I/O库执行实际I/O操作。stdin、stdout通常是行缓存的。  
      3) 无缓存。相当于read、write了。stderr通常是无缓存的，因为它必须尽快输出。

一般而言，由系统选择缓存的长度，并自动分配。标准I/O库在关闭流的时候自动释放缓存。另外，**也可以使用函数fflush()将流所有未写的数据送入（刷新）到内核（内核缓冲区），fsync()将所有内核缓冲区的数据写到文件（磁盘）**。

在标准I/O库中也有引入缓存管理而带来的缺点--效率问题。例如当使用每次一行函数fgets和fputs时，通常需要复制两次数据：一次是在内核和标准I/O缓存之间（当调用read和write时），第二次是在标准I/O缓存（通常系统分配和管理）和用户程序中的行缓存（fgets的参数就需要一个用户行缓存指针）之间。

不管上面讲的到底懂没懂，记住一点：

    使用标准I / O例程的一个优点是无需考虑缓存及最佳I / O长度的选择，并且它并不比直接调用read、write慢多少。

带缓存的文件操作是标准C 库的实现，第一次调用带缓存的文件操作**函数**时标准库会自动分配内存并且读出一段固定大小的内容存储在缓存中。所以以后每次的读写操作并不是针对硬盘上的文件直接进行的，而是针对内存中的缓存的。何时从硬盘中读取文件或者向硬盘中写入文件有标准库的机制控制。不带缓存的文件操作通常都是系统提供的系统调用，更加低级，直接从硬盘中读取和写入文件，由于IO瓶颈的原因，速度并不如意，而且原子操作需要程序员自己保证，但使用得当的话效率并不差。另外标准库中的带缓存文件IO 是调用系统提供的不带缓存IO实现的。

“术语不带缓冲指的是每个read和write都调用嗯内核中的一个系统调用。所有的磁盘I/O都要经过内核的块缓冲（也称内核的缓冲区高速缓存），唯一例外的是对原始磁盘设备的I/O。既然read或write的数据都要被内核缓冲，那么术语“不带缓冲的I/O“指的是在用户的进程中对这两个函数不会自动缓冲，每次read或write就要进行一次系统调用。“--------摘自<unix环境编程>

-----------------------------------------------------------------------------------------

Linux标准IO库缓存策略介绍

标准IO库操作是围绕着流来进行的，当我们通过fopen标准IO库函数打开一个文件，我们就使一个文件和一个IO流相关联。在这里我们把IO流和文件指针FILE\*等同起来，因为所有针对IO流的操作都是通过FILE\*指针来实现的。

    我们知道引入标准IO库的目的是为了提高IO的效率，避免频繁的进行read/write系统调用，而系统调用会消耗较多的资源。因此标准IO库引入了IO缓存，通过累积一定量的IO数据后，然后集中写入到实际的文件中来减少系统调用，从而提高IO效率。标准IO库会自动管理内部的缓存，不需要程序员介入。然而，也正是因为我们看不到标准IO库的缓存，有时候会给我们带来一定的迷惑性。这里介绍下标准IO库的缓存策略。

    一。标准I/O的缓存--标准输出为例：（这里都是指缺省情况下）

    1）当STDOUT连接到终端设备时，那么它就是行缓存的，也就是标准IO库没看到一个新行符 \n时就刷新一次缓存（即执行一次实际的输出操作）。这一特性可以通过如下测试代码来验证

   int main()  
  {  
      printf("This Line Should be Cached...");  
      sleep(3);    //这时候在终端上是看不到任何输出  
      printf("\nThis Line Should be Cached Again");  //这时候可以看到第一个printf的输出，因为被换行符刷新了  
      sleep(3);  //这时候也只能看到一行输出，而看不到第二个printf输出的  
      printf("This Line Should Not be Cached Again\n"); //这时候可以看到第二个和第三个printf的输出，因为被结尾的\n刷新  
      sleep(3);  
      getchar();  
  }

  2）当STDOUT被重定向到一个具体文件时，那么标准输出是全缓存的，也就是说只有当输出缓存被塞满或者调用fflush或fclose时才会执行实际的写入操作，这里就不给出具体例子，可以通过freopen将STDOUT重定向到一个具体文件来进行测试。

   二。标准出错STDERR：为了尽快的看到出错信息，标准出错是不带任何缓存的