带缓冲I/O 和不带缓冲I/O的区别与联系

标签：c linux 缓冲

2014-04-10 15:38 2674人阅读 评论(1) 收藏 举报

分类： C语言（68）

这里搜集从网上看到的一些言论，自认为还是比较靠谱的，有些不懂请的根据自己的理解进行了修正。

首先要明白不带缓冲的概念：所谓不带缓冲，并不是指内核不提供缓冲，而是只是纯的系统调用，不是函数库的调用。系统内核对磁盘的读写都会提供一个块缓冲（在有些地方也被称为内核高速缓存），当用write函数对其写数据时，直接调用系统调用，将数据写入到块缓冲进行排队，当块缓冲达到一定的量时，才会把数据写入磁盘。因此所谓的不带缓冲的I/O是指进程不提供缓冲功能（但内核还是提供缓冲的）。每调用一次write或read函数，直接系统调用。

而带缓冲的I/O是指进程对输入输出流进行了改进，提供了一个流缓冲，当用fwrite函数向磁盘写数据时，先把数据写入流缓冲区中，当达到一定条件，比如流缓冲区满了，或刷新流缓冲，这时候才会把数据一次送往内核提供的块缓冲，再经块缓冲写入磁盘。（双重缓冲）

因此，带缓冲的I/O在往磁盘写入相同的数据量时，会比不带缓冲的I/O调用系统调用的次数要少。

看正常情况下，和磁盘交互的读写文件是怎么个流程！

当应用程序尝试读取某块数据的时候，如果这块数据已经存放在页缓存中，那么这块数据就可以立即返回给应用程序，而不需要经过实际的物理读盘操作。当然，如果数据在应用程序读取之前并未被存放在页缓存中（也就是上面提到的内核高速缓存），那么就需要先将数据从磁盘读到页缓存中去。对于写操作来说，应用程序也会将数据先写到页缓存中去（这里所说的写到页缓存中，如果是调用标准库I/O进行写，那么首先是写到标准库的缓冲区内，如果标准库的缓冲区写满了后，在写到页缓冲中；如果是系统调用，那么直接写到页缓冲中），数据是否被立即写到磁盘上去取决于应用程序所采用的写操作机制：如果用户采用的是同步写机制,那么数据会立即被写回到磁盘上，应用程序会一直等到数据被写完为止；如果用户采用的是延迟写机制，那么应用程序就完全不需要等到数据全部被 写回到磁盘，数据只要被写到页缓存中去就可以了。在延迟写机制的情况下，操作系统会定期地将放在页缓存中的数据刷到磁盘上。与同步写机制不同的是，延迟写机制在数据完全写到磁盘上得时候不会通知应用程序，而异步写机制在数据完全写到磁盘上得时候是会返回给应用程序的。所以延迟写机制本省是存在数据丢失的风险的，而异步写机制则不会有这方面的担心。

下面聊聊不带缓冲的I/O

不带缓存，不是直接对磁盘文件进行读取操作.像read()和write()函数，它们都属于系统调用，只不过在用户层没有缓存，所以叫做无缓存IO.但对于内核来说，还是进行了缓存，只是用户层看不到罢了。

带不带缓存是相对来说的，如果你要写入数据到文件上时（就是写入磁盘上），内核先将数据写入到内核中所设的缓冲储存器，假如这个缓冲储存器的长度是100个字节，你调用系统函：

ssize_t write (int fd,const void * buf,size_t count);

写操作时，设每次写入长度count=10个字节，那么你几要调用10次这个函数才能把这个缓冲区写满，此时数据还是在缓冲区，并没有写入到磁盘，缓冲区满时才进行实际上的IO操作，把数据写入到磁盘上，所以上面说的"不带缓存"不是没有缓存而是没有直写进磁盘就是这个意思（既然没有写入磁盘，调用系统调用为何可以在文件中看到写入的内容呢，因为内核控件是共享的）

那么，既然不带缓存的操作实际在内核是有缓存器的，那带缓存的IO操作又是怎么回事呢？

带缓存IO也叫标准IO，符合ANSI C 的标准IO处理，不依赖系统内核，所以移植性强，我们使用标准IO操作很多时候是为了减少对read()和write()的系统调用次数，带缓存IO其实就是在用户层再建立一个缓存区，这个缓存区的分配和优化长度等细节都是标准IO库代你处理好了，不用去操心，还是用上面那个例子说明这个操作过程：

上面说要写数据到文件上，内核缓存（注意这个不是用户层缓存区）区长度是100字节，我们调用不带缓存的IO函数write()就要调用10次，这样系统效率低，现在我们在用户层建立另一个缓存区（用户层缓存区或者叫流缓存），假设流缓存的长度是50字节，我们用标准C库函数的fwrite()将数据写入到这个流缓存区里面，流缓存区满50字节后在进入内核缓存区，再调用系统函数write()将数据写入到内核缓冲中，如果内核缓冲也接满了，或者内核进行flush操作，那么内核缓冲区内数据就列写入到文件（实质是磁盘）上，看到这里，你用说明白一点，标准IO操作fwrite()最后还是要掉用无缓存IO操作write.这里进行了两次调用fwrite()写100字节也就是进行两次系统调用write()。

如果看到这里还没有一点眉目的话，那就比较麻烦了，希望下面两条总结能够帮上忙：

无缓存IO操作数据流向路径：**数据——内核缓存区——磁盘**

标准IO操作数据流向路径：**数据——流缓存区——内核缓存区——磁盘**

下面是一个网友的见解，以供参考：

不带缓存的IO对文件描述符操作，下面带缓存的IO是针对流的。

标准I/O库就是带缓存的I/O，它由ANSI C标准说明。当然，标准I/O最终都会调用上面的I/O例程。标准I/O库代替用户处理很多细节，比如缓存分配、以优化长度执行I/O等。

标准I/O提供缓存的目的就是减少调用read和write的次数，它对每个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）。

下面的东西是找从网上查到的对这两者的理解，我觉得还是很到位的：

以下主要讨论关于open,write等基本系统IO的带缓冲与不带缓冲的差别

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>

程序中用open和write打开创建并把"hello world"写入文件test.txt，相应用fopen和fwrite操作文件test2.txt。程序执行到open和fopen之后，sleep 15秒，这时用ls查看生成了文件没，这时用open打开的test.txt出现了，用fopen打开的的test2.txt也出现了；当程序执行完write和 fwrite之后，在15秒睡眠期间，用cat查看test.txt，其内容是"hello, world"；但是此时用cat查看test2.txt，其内容为空。睡眠结束后，执行了close（fd），此时再用cat查看test2.txt，发现其内容也有了："hello, world"。该例子证明了open和write是不带缓冲的，即程序一执行其io操作也立即执行，不会停留在系统提供的缓冲里，不需等到close操作完才执行。与之相比的fopen和fwrite则是带缓冲的，（一般）要等到fclose操作完后才会执行。

相关的源码示例如下：

```
#include <unistd.h>
#include <iostream>
#include <fcntl.h>
#include <string>
#include <sys/types.h>
#include <sys/stat.h>


using namespace std;


int main(){
int fd;
FILE *file;
char *s="hello,world\n";
if((fd=open("test.txt",O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR))==-1){
cout<<"Error open file"<<endl;
return -1;
}
if((file=fopen("test2.txt","w"))==NULL){
cout<<"Error Open File."<<endl;
return -1;
}
cout<<"File has been Opened."<<endl;
sleep(15);
if(write(fd,s,strlen(s))<strlen(s)){
cout<<"Write Error"<<endl;


return -1;
}
if(fwrite(s,sizeof(char),strlen(s),file)<strlen(s)){
cout<<"Write Error in 2."<<endl;


return -1;
}
cout<<"After write"<<endl;


sleep(15);
cout<<"After sleep."<<endl;


close(fd);
return 0;
}
```

以 ssize_t write(int filedes, const void *buff, size_t nbytes)和size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp)来讲讲自己对unix系统下带缓存的I/O和不带缓存的I/O的区别。

首先要清楚一个概念，所谓的带缓存并不是指上面两个函数的buf参数。

当将数据写到文件上时，内核先将该数据写到缓存，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输入队列，待其到达队首，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。

现在假设内核所设的缓存是100个字节，如果你使用write，且buff的size为10，当你要把9个同样的buff写到文件时，你需要调用9次write，也就是9次系统调用，此时也并没有写到硬盘，如果想立即写到硬盘，调用sync，可以进行实际的I/O操作。

标准I/O，也就是带缓存的I/O采用 FILE*，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，又称为用户空进程空间的缓存，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，假设流缓存的长度为50字节，把以上的数据写到文件，则只需要2次系统调用（fwrite调用write系统调用），因为先把数据写到流缓存，当其满以后或者调用fflush时才填入内核缓存，所以进行了2次的系统调用write。

fflush将流所有未写的缓冲送入（刷新）到内核（内核缓冲区），fsync将所有内核缓冲区的缓冲写到文件（磁盘）。至于究竟写到了文件中还是内核缓冲区中对于进程来说是没有差别 的.如果进程A和进程B对打开同一文件.进程A写到内核I/O缓冲区中的数据从进程B也能读到.因为内核空间是进程共享的.

而标准库的I/O缓冲区则不具有这一特性.因为进程的用户空间是完全独立的（个人觉得这句话非常重要）

不带缓存的read和write是相对于 fread/fwrite等流函数来说明的，因为fread和fwrite是用户函数（3），所以他们会在用户层进行一次数据的缓存，而read/write是系统调用（2）所以他们在用户层是没有缓存的，所以将read和write是无缓存的IO，其实对于内核来说还是进行了缓存，不过用户层看不到罢了。

上面的内容介绍了库缓冲机制，其中也提到了内核缓冲区这个概念，到底内核缓冲存在的价值是很么呢：

为什么总是需要将数据由内核缓冲区换到用户缓冲区或者相反呢？

答：用户进程是运行在用户空间的，不能直接操作内核缓冲区的数据。 用户进程进行系统调用的时候，会由用户态切换到内核态，待内核处理完之后再返回用户态

应用缓冲技术能很明显的提高系统效率。内核与外围设备的数据交换，内核与用户空间的数据交换都是比较贵时的，使用缓冲区就是为了优化这些贵时的操作。其实核心到用户空间的操作本身是不buffer的，是由I/O库用buffer来优化了这个操作。比如read本来从内核读取数据时是比较贵时的，所以一次取出一块，以避免多次陷入内核。

应用内核缓冲区的 主要思想就是一次读入大量的数据放在缓冲区，需要的时候从缓冲区取得数据。

管理员模式和用户模式之间的切换需要消耗时间，但相比之下，磁盘的I/O操作消耗的时间更多，为了提高效率，内核也使用缓冲区技术来提高对磁盘的访问速度。磁盘是数据块 的集合，内核会对磁盘上的数据块做缓冲。内核将磁盘上的数据块复制到内核缓冲区中，当一个用户空间中的进程要从磁盘上读数据时，内核一般不直接读磁盘，而 是将内核缓冲区中的数据复制到进程的缓冲区中。当进程所要求的数据块不在内核缓冲区时，内核会把相应的数据块加入到请求队列，然后把该进程挂起，接着为其 他进程服务。一段时间之后(其实很短的时间)，内核把相应的数据块从磁盘读到内核缓冲区，然后再把数据复制到进程的缓冲区中，最后唤醒被挂起的进程。

注：理解内核缓冲区技术的原理有助于更好的掌握系统调用read&write，read把数据从内核缓冲区复制到进程缓冲区，write把数据从进程缓冲区复制到内核缓冲区，它们不等价于数据在内核缓冲区和磁盘之间的交换。

从理论上讲，内核可以在任何时候写磁盘，但并不是所有的write操作都会导致内核的写动作。内核会把要写的数据暂时存在缓冲区中，积累到一定数量后再一次写入。有时会导致意外情况，比如断电，内核还来不及把内核缓冲区中的数据写进磁盘上，这些更新的数据就会丢失。

应用内核缓冲技术导致的结果是：提高了磁盘的I/O效率；优化了磁盘的写操作；需要及时的将缓冲数据写到磁盘。