

昵称：hapjin
园龄：3年2个月
粉丝：155
关注：19
+加关注

<	2018年4月						>
日	一	二	三	四	五	六	
25	26	27	28	29	30	31	
1	2	3	4	5	6	7	
8	9	10	11	12	13	14	
15	16	17	18	19	20	21	
22	23	24	25	26	27	28	
29	30	1	2	3	4	5	

搜索

找找看

谷歌搜索

随笔分类
Big Data(11)
Design Pattern(4)
JAVA(33)
JMS(9)
leetcode(46)
Linux(4)
machine learning(23)
Python(4)
分布式理论(7)
工具&技巧(7)
计算机基础(2)
数据结构(48)
算法(48)

积分与排名
积分 - 318443
排名 - 560

最新评论

1. Re:JAVA多线程之中断机制(如何处理中断?)

@夜无痕星interrupted() 和 isInterrupted()这两个方法 是有区别的：[参考这篇文章]()...

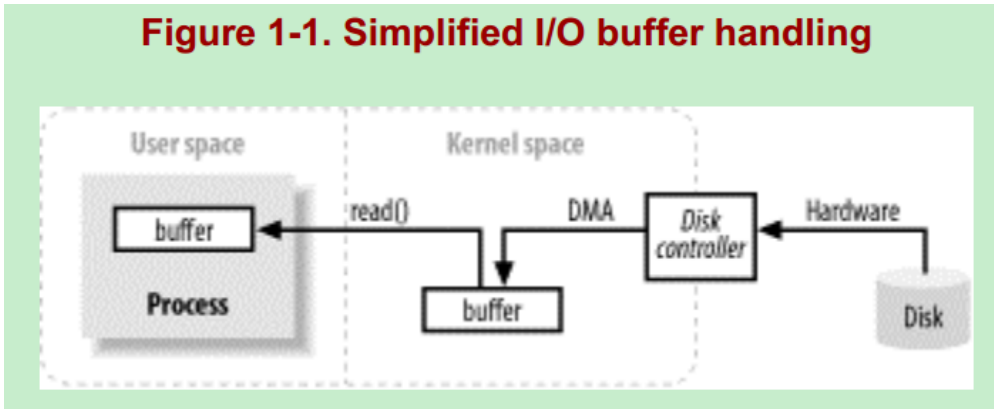
JAVA IO 以及 NIO 理解

由于Netty，了解了一些异步IO的知识，JAVA里面NIO就是原来的IO的一个补充，本文主要记录下在JAVA中IO的底层实现原理，以及对ZeroCopy技术介绍。

IO，其实意味着：数据不停地搬入搬出缓冲区而已（使用了缓冲区）。比如，用户程序发起读操作，导致“syscall read”系统调用，就会把数据搬入到一个buffer中；用户发起写操作，导致“syscall write”系统调用，将会把一个buffer中的数据搬出去(发送到网络中 or 写入到磁盘文件)

上面的过程看似简单，但是底层操作系统具体如何实现以及实现的细节就非常复杂了。正是因为实现方式不同，有针对普通情况下的文件传输(暂且称普通IO吧)，也有针对大文件传输或者批量大数据传输的实现方式，比如zerocopy技术。

先来看一张普通的IO处理的流程图：



整个IO过程的流程如下：

1) 程序员写代码创建一个**缓冲区（这个缓冲区是用户缓冲区）**：哈哈。然后在一个while循环里面调用read()方法读数据(触发"syscall read"系统调用)

```
byte[] b = new byte[4096];

while((read = inputStream.read(b))>=0) {
    total = total + read;
    // other code....
}
```

2)当执行到read()方法时，其实底层是发生了很多操作的：

①内核给磁盘控制器发命令说：我要读磁盘上的某某块磁盘块上的数据。--kernel issuing a command to the disk controller hardware to fetch the data from disk.

②在DMA的控制下，把磁盘上的数据读入到**内核缓冲区**。--The disk controller writes the data directly into a kernel memory buffer by DMA

③内核把数据从**内核缓冲区**复制到**用户缓冲区**。--kernel copies the data from the temporary buffer in kernel space

这里的用户缓冲区应该就是我们写的代码中 new 的 byte[] 数组。

从上面的步骤中可以分析出什么？

④对于操作系统而言，JVM只是一个用户进程，处于用户态空间中。而处于用户态空间的进程是不能直接操作底层的硬件的。而IO操作就需要操作底层的硬件，比如磁盘。因此，IO操作必须得借助内核的帮助才能完成(中断，trap)，即：会有用户态到内核态的切换。

⑤我们写代码 new byte[] 数组时，一般都是都是“随意” 创建一个“任意大小”的数组。比如，new byte[128]、new byte[1024]、new byte[4096]....

但是，对于磁盘块的读取而言，每次访问磁盘读数据时，并不是读任意大小的数据的，而是：每次读一个磁盘块或者若干个磁盘块(这是因为访问磁盘操作代价是很大的，而且我们也相信局部性原理) 因此，就需要有一个“中间缓冲区”--即内核缓冲区。先把数据从磁盘读到内核缓冲区中，然后再把数据从内核缓冲区搬到用户缓冲区。

这也是为什么我们总感觉到第一次read操作很慢，而后续的read操作却很快的原因吧。因为，对于**后续**的read操作而言，它所需要读的数据很可能已经在内核缓冲区了，此时只需将内核缓冲区中的数据拷贝到用户缓冲区即可，并未涉及到底层的读取磁盘操作，当然就快了。

```
The kernel tries to cache and/or prefetch data, so the data being requested by the process may already be available in kernel space.
If so, the data requested by the process is copied out.
```

	--hapjin
2. Re:JAVA多线程之中断机制(如何处理中断?)	
@琅琊天 我的理解是：第7行代码 Thread.sleep(20);//modify 2000 to 20的目的应该是：让main线程睡眠一下，好让Mythread线程 获得cpu运行MyThrea.....	
	--hapjin
3. Re:Netty 实现HTTP文件服务器	
这个好像是修改官方的Sample, 我找到了,Netty自己实现了HttpUtil	
	--呆呆的哲学
4. Re:JAVA多线程之中断机制(如何处理中断?)	
if里面为什么不用isInterrupted()去检测中断状态？	
	--夜无痕星
5. Re:JAVA多线程之中断机制(如何处理中断?)	
深度好文	
	--夜无痕星

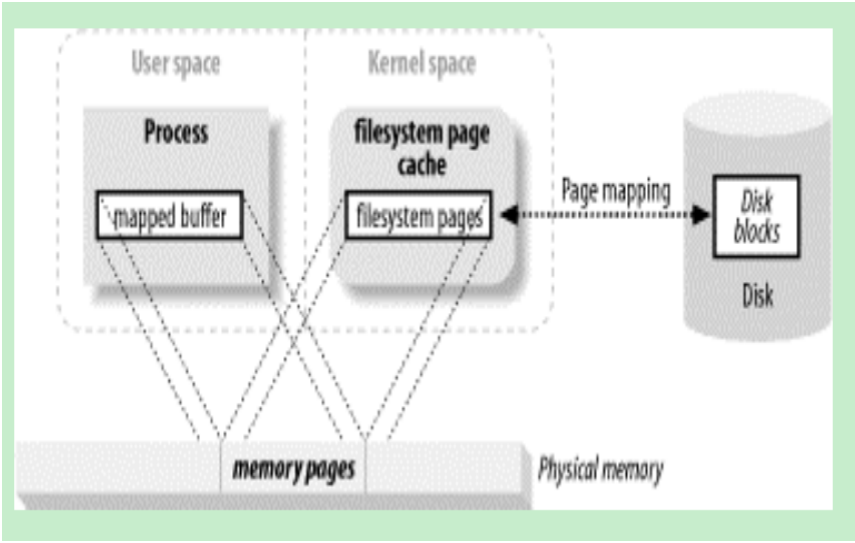
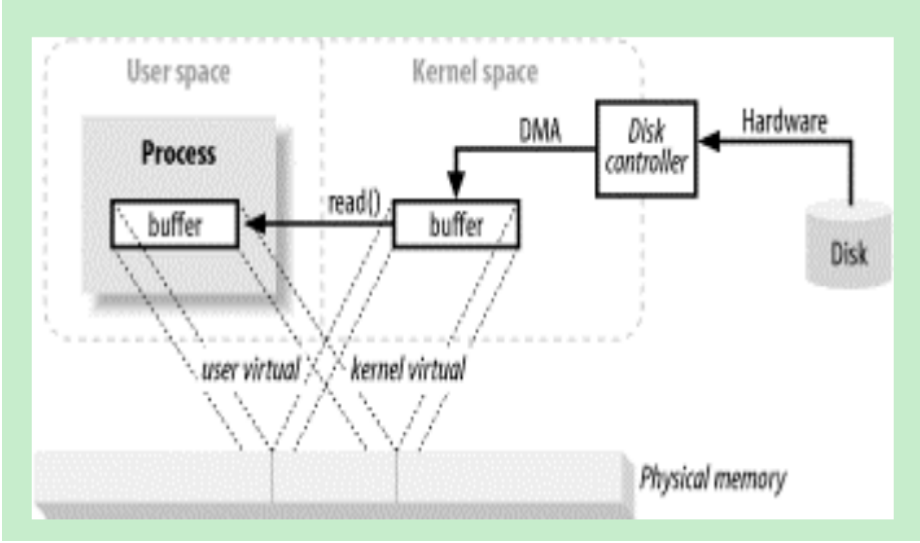
If the data isn't available, the process is **suspended** while the kernel goes about bringing the data into memory.

如果数据不可用，process将会被挂起，并需要等待内核从磁盘上把数据取到内核缓冲区中。

那我们可能会说：DMA为什么不直接将磁盘上的数据读入到用户缓冲区呢？一方面是 ⑥中提到的内核缓冲区作为一个中间缓冲区。用来“**适配**”用户缓冲区的“**任意大小**”和每次读磁盘块的**固定大小**。另一方面则是，用户缓冲区位于用户态空间，而DMA读取数据这种操作涉及到底层的硬件，硬件一般是不能直接访问用户态空间的（OS的原因吧）

综上，由于DMA不能直接访问用户空间(用户缓冲区)，普通IO操作需要将数据来回地在 用户缓冲区 和 内核缓冲区移动，这在一定程度上影响了IO的速度。那有没有相应的解决方案呢？

那就是直接内存映射IO，也即JAVA NIO中提到的内存映射文件，或者说 直接内存....总之，它们表达的意思都差不多。示例图如下：



从上图可以看出：内核空间的 buffer 与 用户空间的 buffer 都映射到同一块 物理内存区域。

它的主要特点如下：

- ①对文件的操作不需要再发read 或者 write 系统调用了---The user process sees the file data as memory, so there is **no need to issue read() or write() system calls**.
- ②当用户进程访问“内存映射文件”地址时，自动产生缺页错误，然后由底层的OS负责将磁盘上的数据送到内存。关于页式存储管理，可参考：[内存分配与内存管理的一些理解](#)

As the user process touches the mapped memory space, page faults will be generated automatically to bring in the file data from disk.
If the user modifies the mapped memory space, the affected page is automatically marked as dirty and will be subsequently flushed to disk to update the file.

这就是是JAVA NIO中提到的内存映射缓冲区（Memory-Mapped-Buffer）它类似于JAVA NIO中的**直接缓冲区(Direct Buffer)**。MappedByteBuffer可以通过java.nio.channels.FileChannel.java(通道)的 map方法创建。

使用内存映射缓冲区来操作文件，它比普通的IO操作读文件要快得多。甚至比使用文件通道 (FileChannel)操作文件 还要快。因为，使用内存映射缓冲区操作文件时，没有显示的系统调用 (read,write)，而且OS还会自动缓存一些文件页(memory page)

zerocopy技术介绍

看完了上面的IO操作的底层实现过程，再来了解zerocopy技术就很easy了。IBM有一篇名为《Efficient data transfer through zero copy》的论文对zerocopy做了完整的介绍。感觉非常好，下面就基于这篇文来记录下自己的一些理解。

zerocopy技术的目标就是提高IO密集型JAVA应用程序的性能。在本文的前面部分介绍了：IO操作需要数据频繁地在内核缓冲区和用户缓冲区之间拷贝，而zerocopy技术可以减少这种拷贝的次数，同时也降低了

上下文切换(用户态与内核态之间的切换)的次数。

比如，大多数WEB应用程序执行的一项操作就是：接受用户请求--->从本地磁盘读数据--->数据进入内核缓冲区--->用户缓冲区--->内核缓冲区--->用户缓冲区--->socket发送

数据每次在内核缓冲区与用户缓冲区之间的拷贝会消耗CPU以及内存的带宽。而zerocopy有效减少了这种拷贝次数。

```
Each time data traverses the user-kernel boundary, it must be copied, which consumes CPU cycles and memory bandwidth.
Fortunately, you can eliminate these copies through a technique called—appropriately enough —zero copy
```

那它是怎么做到的呢？

我们知道，JVM(JAVA虚拟机)为JAVA语言提供了跨平台的一致性，屏蔽了底层操作系统的具体实现细节，因此，JAVA语言也很难直接使用底层操作系统提供的一些“奇技淫巧”。

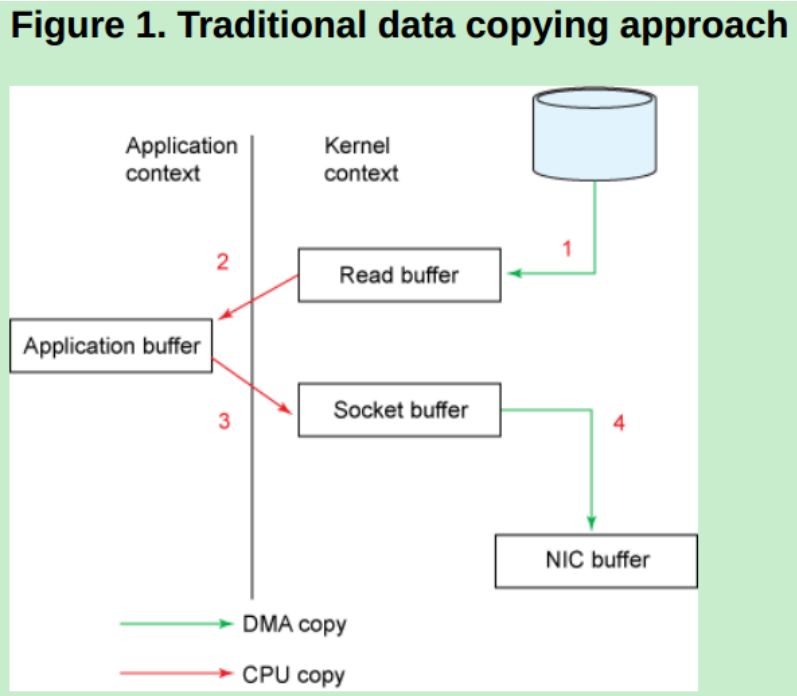
而要实现zerocopy，首先得有操作系统的支持。其次，JDK类库也要提供相应的接口支持。幸运的是，自JDK1.4以来，JDK提供了对NIO的支持，通过java.nio.channels.FileChannel类的transferTo()方法可以直接将字节传送到可写的通道中(Writable Channel)，并不需要将字节送入用户程序空间(用户缓冲区)

```
You can use the transferTo()method to transfer bytes directly from the channel on which it is invoked to another writable byte channel, without requiring data to flow through the application
```

下面就来详细分析一下经典的web服务器(比如文件服务器)干的活：从磁盘中中读文件，并把文件通过网络(socket)发送给Client。

```
File.read(fileDesc, buf, len);
Socket.send(socket, buf, len);
```

从代码上看，就是两步操作。第一步：将文件读入buf；第二步：将 buf 中的数据通过socket发送出去。但是，这两步操作需要**四次上下文切换**(用户态与内核态之间的切换) 和 **四次拷贝操作**才能完成。



①第一次上下文切换发生在 read()方法执行，表示服务器要去磁盘上读文件了，这会导致一个 sys_read()的系统调用。此时由用户态切换到内核态，完成的动作是：DMA把磁盘上的数据读入到内核缓冲区中（**这也是第一次拷贝**）。


②第二次上下文切换发生在**read()方法的返回(这也说明read()是一个阻塞调用)**，表示数据已经成功从磁盘上读到内核缓冲区了。此时，由内核态返回到用户态，完成的动作是：将内核缓冲区中的数据拷贝到用户缓冲区（**这是第二次拷贝**）。

③第三次上下文切换发生在 send()方法执行，表示服务器准备把数据发送出去了。此时，由用户态切换到内核态，完成的动作是：将用户缓冲区中的数据拷贝到内核缓冲区(**这是第三次拷贝**)

④第四次上下文切换发生在 send()方法的返回【这里的send()方法可以异步返回，所谓异步返回就是：线程执行了send()之后立即从send()返回，剩下的数据拷贝及发送就交给底层操作系统实现了】。此时，由内核态返回到用户态，完成的动作是：将内核缓冲区中的数据送到 **protocol engine**.（**这是第四次拷贝**）


这里对 protocol engine不是太了解，但是从上面的示例图来看：它是NIC(NetWork Interface Card) buffer。网卡的buffer???

下面这段话，非常值得一读：**这里再一次提到了为什么需要内核缓冲区。**

```

Use of the intermediate kernel buffer (rather than a direct transfer of the data into the user buffer)might seem inefficient. But intermediate kernel buffers were introduced into the process to improve performance. Using the intermediate
```



```
buffer on the read side allows the kernel buffer to act as a "readahead cache"
when the application hasn't asked for as much data as the kernel buffer holds.
This significantly improves performance when the requested data amount is less
than the kernel buffer size. The intermediate buffer on the write side allows the write
to complete asynchronously.
```



一个核心观点就是：内核缓冲区提高了性能。咦？是不是很奇怪？因为前面一直说正是因为引入了内核缓冲区(中间缓冲区)，使得数据来回地拷贝，降低了效率。

那先来看看，它为什么说内核缓冲区提高了性能。

对于读操作而言，内核缓冲区就相当于一个“readahead cache”，当用户程序**一次**只需要读一小部分数据时，首先操作系统从磁盘上读一大块数据到内核缓冲区，用户程序只取走了一小部分(*我可以只 new 了一个 128B的byte数组啊! new byte[128]*)。当用户程序下一次再读数据，就可以直接从内核缓冲区中取了，操作系统就不需要再次访问磁盘啦！因为用户要读的数据已经在内核缓冲区啦！这也是前面提到的：为什么后续的读操作(read()方法调用)要明显地比第一次快的原因。**从这个角度而言，内核缓冲区确实提高了读操作的性能。**

再来看写操作：可以做到“异步写”(write asynchronously)。也即：wirte(dest[]) 时，用户程序告诉操作系统，把dest[]数组中的内容写到XX文件中去，于是write方法就返回了。操作系统则在后台默默地把用户缓冲区中的内容(dest[])拷贝到内核缓冲区，再把内核缓冲区中的数据写入磁盘。那么，只要内核缓冲区未满，用户的write操作就可以很快地返回。这应该就是异步刷盘策略吧。

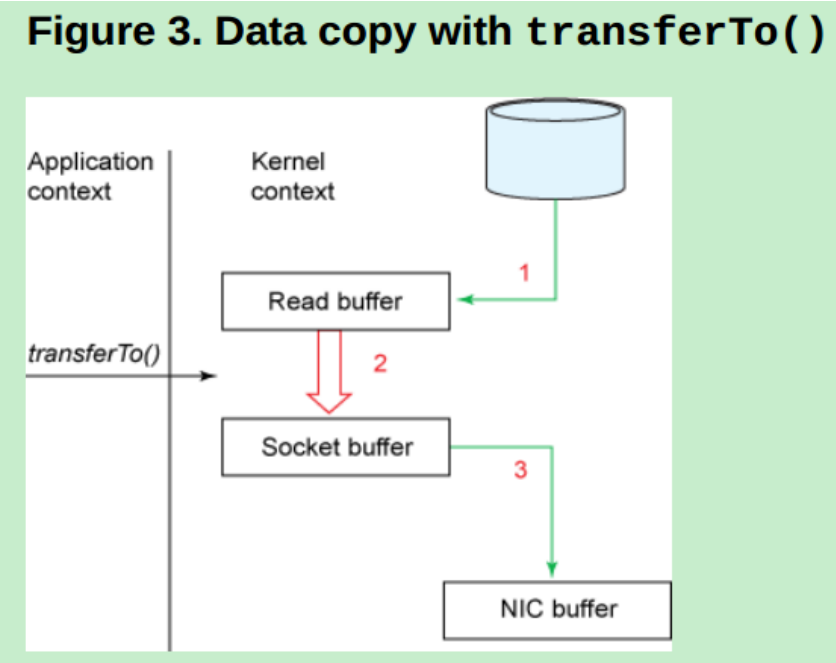
(其实，到这里。以前一个纠结的问题就是同步IO，异步IO，阻塞IO，非阻塞IO之间的区别已经没有太大的意义了。这些概念，只是针对的看问题的角度不一样而已。阻塞、非阻塞是针对线程自身而言；同步、异步是针对线程以及影响它的外部事件而言....)【更加完美、精辟的解释可以参考这个系列的文章：[系统间通信 \(3\) ——IO通信模型和AVA实践 上篇](#)】

既然，你把内核缓冲区说得这么强大和完美，那还要 zerocopy干嘛啊？？？

```
Unfortunately, this approach itself can become a performance bottleneck if the size of
the data requested
is considerably larger than the kernel buffer size. The data gets copied multiple times
among the disk, kernel buffer,
and user buffer before it is finally delivered to the application.
Zero copy improves performance by eliminating these redundant data copies.
```

终于轮到zerocopy粉墨登场了。**当需要传输的数据远远大于内核缓冲区的大小时，内核缓冲区就会成为瓶颈。**这也是为什么zerocopy技术合适大文件传输的原因。内核缓冲区为啥成为了瓶颈？---我想，很大的一个原因是它已经起不到“缓冲”的功能了，毕竟传输的数据量太大了。

下面来看看zerocopy技术是如何来处理文件传输的。



当 transferTo()方法 被调用时，由**用户态切换到内核态**。完成的动作是：DMA将数据从磁盘读入 Read buffer中(第一次数据拷贝)。然后，还是在内核空间中，将数据从Read buffer 拷贝到 Socket buffer(第二次数据拷贝)，最终再将数据从 Socket buffer 拷贝到 NIC buffer(第三次数据拷贝)。然后，再**从内核态返回到用户态**。

上面整个过程就只涉及到了：三次数据拷贝和二次上下文切换。感觉也才减少了一次数据拷贝嘛。但这里已经不涉及用户空间的缓冲区了。

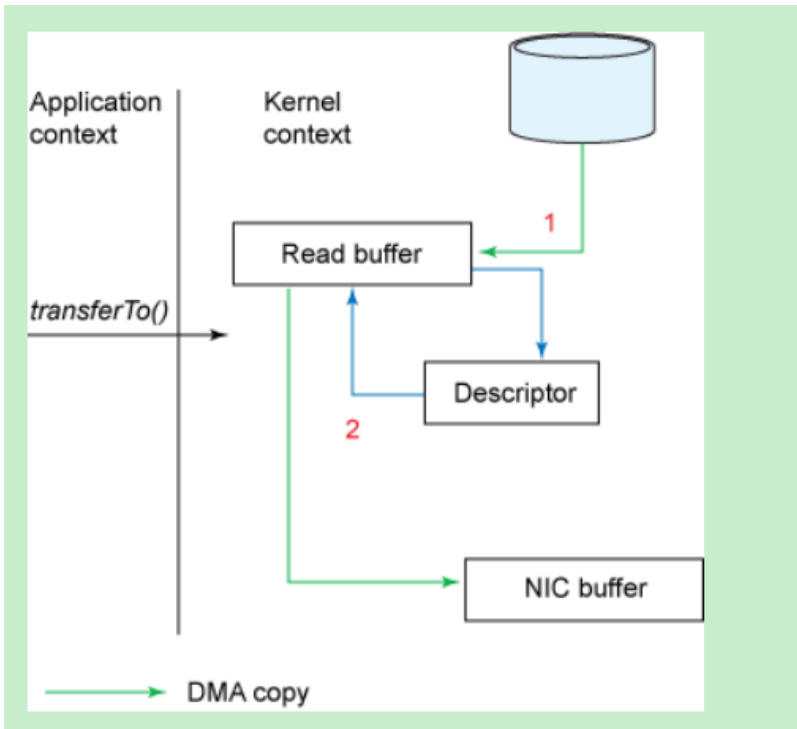
三次数据拷贝中，也只有一次拷贝需要到CPU的干预。（第2次拷贝），而前面的传统数据拷贝需要四次且有三次拷贝需要CPU的干预。

```
This is an improvement: we've reduced the number of context switches from four to two and
reduced the number of data copies
from four to three (only one of which involves the CPU)
```

如果说zerocopy技术只能完成到这步，那也就 just so so 了。

We can further reduce the data duplication done by the kernel **if the underlying network interface card supports gather operations**. In Linux kernels 2.4 and later, the socket buffer descriptor was modified to accommodate this requirement. This approach not only reduces multiple context switches but also eliminates the duplicated data copies that require CPU involvement.

也就是说，**如果底层的网络硬件以及操作系统支持**，还可以进一步减少数据拷贝次数 以及 CPU干预次数。



从上图看出：这里一共只有两次拷贝 和 两次上下文切换。而且这两次拷贝都是DMA copy，并不需要CPU干预(严谨一点的话就是不完全需要吧.)。

整个过程如下：

用户程序执行 transferTo()方法，导致一次系统调用，从用户态切换到内核态。完成的动作是：DMA将数据从磁盘中拷贝到Read buffer

用一个描述符标记此次待传输数据的地址以及长度，DMA直接把数据从Read buffer 传输到 NIC buffer。数据拷贝过程都不用CPU干预了。

总结：

这篇文章从IO底层实现原理开始讲解，分析了IO底层实现细节的一些优缺点，以及为什么引入zerocopy技术和zerocopy技术的实现原理。个人的学习记录，转载请注明出处。


参考文献：

- 1) 《JAVA NIO》 O'Reilly出版社
- 2) 《Efficient data transfer through zero copy》 IBM出版
- 3) Zero Copy I: User-Mode Perspective

分类: [JAVA](#),[计算机基础](#)

标签: [NIO](#), [zerocopy](#)

[好文要顶](#)[关注我](#)[收藏该文](#)



[hapijin](#)
关注 - 19
粉丝 - 155
[+加关注](#)

5

0

« 上一篇：[演示数字黑洞现象](#)
» 下一篇：[二叉树的构造](#)

posted @ 2016-08-04 16:28 hapijin 阅读(7807) 评论(3) 编辑 收藏

评论列表

#1楼 2016-08-04 16:42 独孤求败呢

什么鬼

支持(0) 反对(0)

版主讲的用户态和内核态很清晰，受教了！！！

支持(0) 反对(0)

#3楼[楼主] 2018-03-01 21:02 hapjin

@ HACKXIYU
哈哈，谢谢。

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。



最新IT新闻：

- 比特币一季度价格腰斩 全球投机者中国人或超过一半
 - 苹果周一发布红色版iPhone 8 为慈善筹款
 - Facebook又封杀了一数据公司 该公司将用户信息共享给广告主
 - 打车服务和无人驾驶汽车让城市交通变得越来越拥堵
 - 因为欧洲税务问题 抗议组织到苹果店“装死”
- » 更多新闻...



最新知识库文章：

- 写给自学者的入门指南
 - 和程序员谈恋爱
 - 学会学习
 - 优秀技术人的管理陷阱
 - 作为一个程序员，数学对你到底有多重要
- » 更多知识库文章...