## 什么是“零拷贝”技术

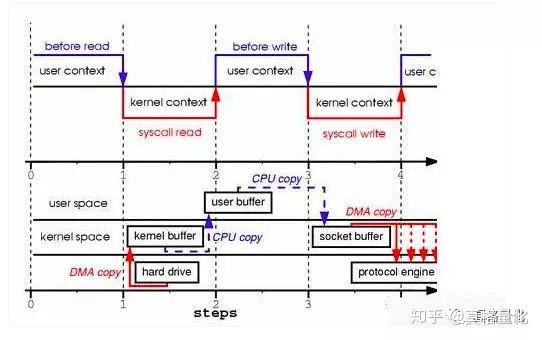
高频交易的研究者有时会遇到“零拷贝”（zero-copy）技术这个名词，那么什么是“零拷贝”呢？

**一般的文件传输过程**

考虑这样一种常用的情形：开发者需要将静态内容（类似图片、数据表、文件）展示给远程的用户。那么这个情形就意味着开发者需要先将静态内容从磁盘中拷贝出来放到一个内存buf中，然后将这个buf通过socket传输给用户，进而用户或者静态内容的展示。这看起来再正常不过了，但是实际上这是很低效的流程，我们把上面的这种情形抽象成下面的过程：



首先调用read将静态内容，这里假设为数据文件A，读取到tmp\_buf, 然后调用write将tmp\_buf写入到socket中，如图：



在这个过程中数据文件A的经历了4次复制的过程：

* 首先，调用read时，数据文件A拷贝到了kernel模式；
* 之后，CPU控制将kernel模式数据复制到user模式下；
* 调用write时，先将user模式下的内容复制到到kernel模式下的socket的buffer中；
* 最后将kernel模式下的socket buffer的数据复制到网卡设备中传送；

从上面的过程可以看出，数据白白从kernel模式到user模式走了一圈，浪费了2次copy(第一次，从kernel模式拷贝到user模式；第二次从user模式再拷贝回kernel模式，即上面4次过程的第2和3步骤)。而且上面的过程中kernel和user模式的上下文的切换也是4次**。**

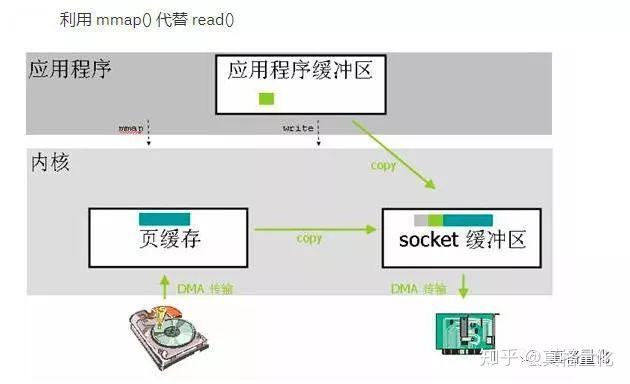
幸运的是，开发者可以用“零拷贝”技术来去掉这些无谓的复制。应用程序用Zero-Copy来请求kernel直接把disk的data传输给socket，而不是通过应用程序传输。Zero-Copy大大提高了应用程序的性能，并且减少了kernel和user模式上下文的切换。

**Linux中的零拷贝**

例如，在 Linux 中，减少拷贝次数的一种方法是调用 mmap() 来代替调用 read，比如：



首先，应用程序调用了 mmap() 之后，数据会先通过 DMA 被复制到操作系统内核的缓冲区中去。接着，应用程序跟操作系统共享这个缓冲区，这样，操作系统内核和应用程序存储空间就不需要再进行任何的数据复制操作。应用程序调用了 write() 之后，操作系统内核将数据从原来的内核缓冲区中复制到与 socket 相关的内核缓冲区中。接下来，数据从内核 socket 缓冲区复制到协议引擎中去，这是第三次数据拷贝操作。



通过使用 mmap() 来代替 read(), 已经可以减半操作系统需要进行数据拷贝的次数。当大量数据需要传输的时候，这样做就会有一个比较好的效率。但是，这种改进也是需要代价的，使用 mmap() 其实是存在潜在的问题的。当对文件进行了内存映射，然后调用 write() 系统调用，如果此时其他的进程截断了这个文件，那么 write() 系统调用将会被总线错误信号 SIGBUS 中断，因为此时正在执行的是一个错误的存储访问。这个信号将会导致进程被杀死，解决这个问题可以通过以下这两种方法：

为 SIGBUS 安装一个新的信号处理器，这样，write() 系统调用在它被中断之前就返回已经写入的字节数目，errno 会被设置成 success。但是这种方法也有其缺点，它不能反映出产生这个问题的根源所在，因为 BIGBUS 信号只是显示某进程发生了一些很严重的错误。

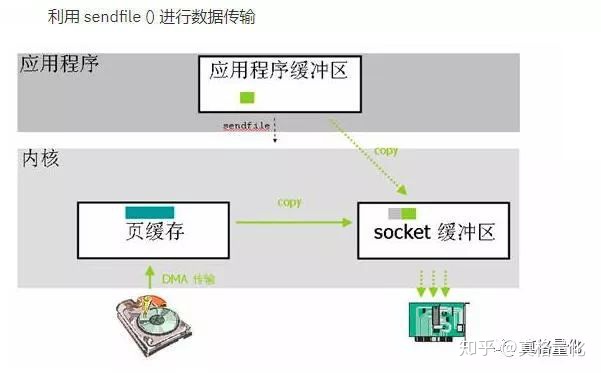
第二种方法是通过文件租借锁来解决这个问题的，这种方法相对来说更好一些。我们可以通过内核对文件加读或者写的租借锁，当另外一个进程尝试对用户正在进行传输的文件进行截断的时候，内核会发送给用户一个实时信号：RT\_SIGNAL\_LEASE 信号，这个信号会告诉用户内核破坏了用户加在那个文件上的写或者读租借锁，那么 write() 系统调用则会被中断，并且进程会被 SIGBUS 信号杀死，返回值则是中断前写的字节数，errno 也会被设置为 success。文件租借锁需要在对文件进行内存映射之前设置。

使用 mmap 是 POSIX 兼容的，但是使用 mmap 并不一定能获得理想的数据传输性能。数据传输的过程中仍然需要一次 CPU 复制操作，而且映射操作也是一个开销很大的虚拟存储操作，这种操作需要通过更改页表以及冲刷 TLB （使得 TLB 的内容无效）来维持存储的一致性。但是，因为映射通常适用于较大范围，所以对于相同长度的数据来说，映射所带来的开销远远低于 CPU 拷贝所带来的开销。

**sendfile()**

为了简化用户接口，同时还要继续保留 mmap()/write() 技术的优点：减少 CPU 的复制次数，Linux 在版本 2.1 中引入了 sendfile() 这个系统调用。

sendfile() 不仅减少了数据复制操作，它也减少了上下文切换。首先：sendfile() 系统调用利用 DMA 引擎将文件中的数据复制到操作系统内核缓冲区中，然后数据被复制到与 socket 相关的内核缓冲区中去。接下来，DMA 引擎将数据从内核 socket 缓冲区中复制到协议引擎中去。如果在用户调用 sendfile () 系统调用进行数据传输的过程中有其他进程截断了该文件，那么 sendfile () 系统调用会简单地返回给用户应用程序中断前所传输的字节数，errno 会被设置为 success。如果在调用 sendfile() 之前操作系统对文件加上了租借锁，那么 sendfile() 的操作和返回状态将会和 mmap()/write () 一样。



sendfile() 系统调用不需要将数据拷贝或者映射到应用程序地址空间中去，所以 sendfile() 只是适用于应用程序地址空间不需要对所访问数据进行处理的情况。相对于 mmap() 方法来说，因为 sendfile 传输的数据没有越过用户应用程序 / 操作系统内核的边界线，所以 sendfile () 也极大地减少了存储管理的开销。但是，sendfile () 也有很多局限性，如下所列：

sendfile() 局限于基于文件服务的网络应用程序，比如 web 服务器。据说，在 Linux 内核中实现 sendfile() 只是为了在其他平台上使用 sendfile() 的 Apache 程序。

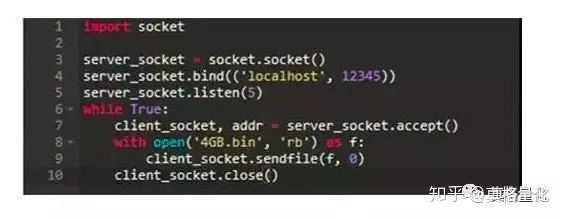
由于网络传输具有异步性，很难在 sendfile () 系统调用的接收端进行配对的实现方式，所以数据传输的接收端一般没有用到这种技术。

基于性能的考虑来说，sendfile () 仍然需要有一次从文件到 socket 缓冲区的 CPU 复制操作，这就导致页缓存有可能会被传输的数据所污染。

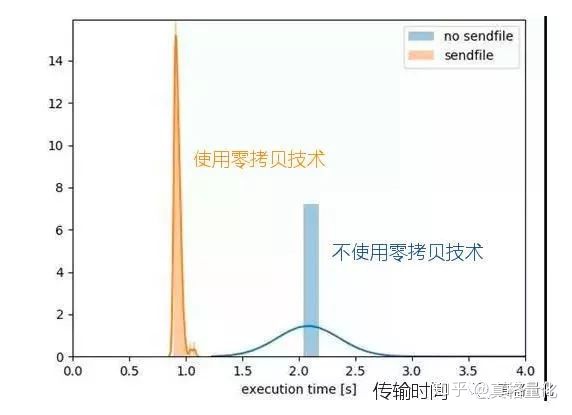
**Python对“零拷贝”的支持**

自从Python 3.3中sendfile系统调用可用作os.sendfile，python 3.5 为基于socket的应用带来了更高级的封装包socket.socket.sendfile。

例如通过socket.socket.sendfile来改进大尺寸文件的传输速度：



在100次大尺寸文件（4GB数据文件）传输测试中，和不使用零拷贝技术的sock.recv相比，使用零拷贝技术的socket.sendfile可以减少一半的时间且显著提高文件传输的稳定性（降低传输时间的标准差）。对于需要大量传输数据的量化交易应用，这一技术也能改善交易系统的性能。



**什么是DMA?**

DMA的全称叫直接内存存取（Direct Memory Access），是一种允许外围设备（硬件子系统）直接访问系统主内存的机制。也就是说，基于 DMA 访问方式，系统主内存于硬盘或网卡之间的数据传输可以绕开 CPU 的全程调度。目前大多数的硬件设备，包括磁盘控制器、网卡、显卡以及声卡等都支持 DMA 技术。