Linux内存--零拷贝

原文链接

本系列有如下几篇

[Linux 内存问题汇总](/2020/01/15/Linux 内存问题汇总/)

Linux内存-PageCache

Linux内存-管理和碎片

Linux内存-HugePage

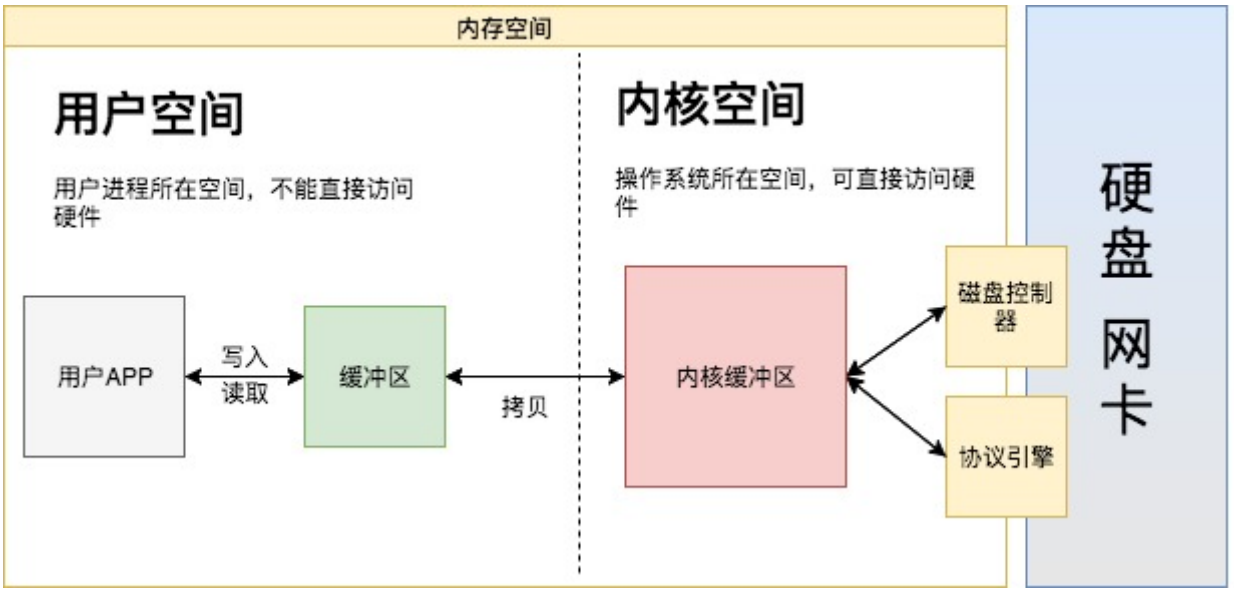
Linux内存-零拷贝

零拷贝

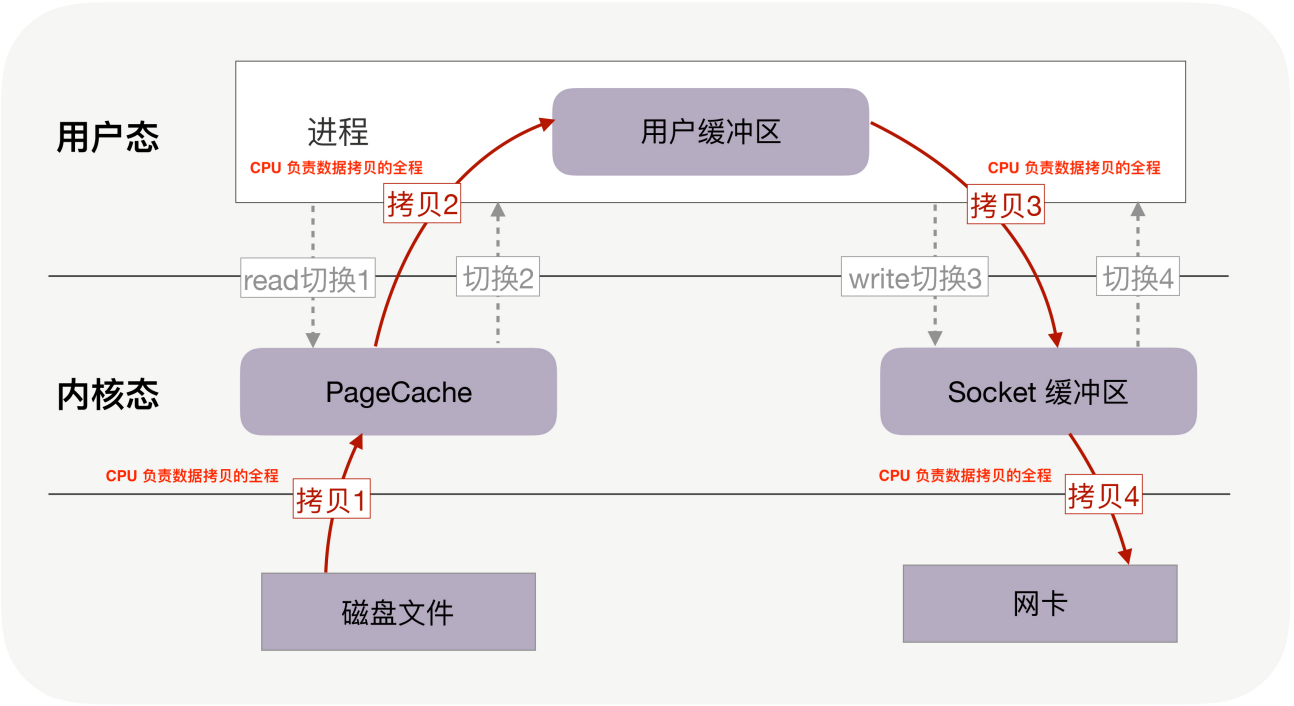
> “**Zero-copy**“ describes computer operations in which the CPU does not perform the task of copying data from one memory area to another. This is frequently used to save CPU cycles and memory bandwidth when transmitting a file over a network.

零拷贝技术是指计算机执行操作时，CPU不需要先将数据从某处内存复制到另一个特定区域。这种技术通常用于通过网络传输文件时节省 CPU 和内存带宽。

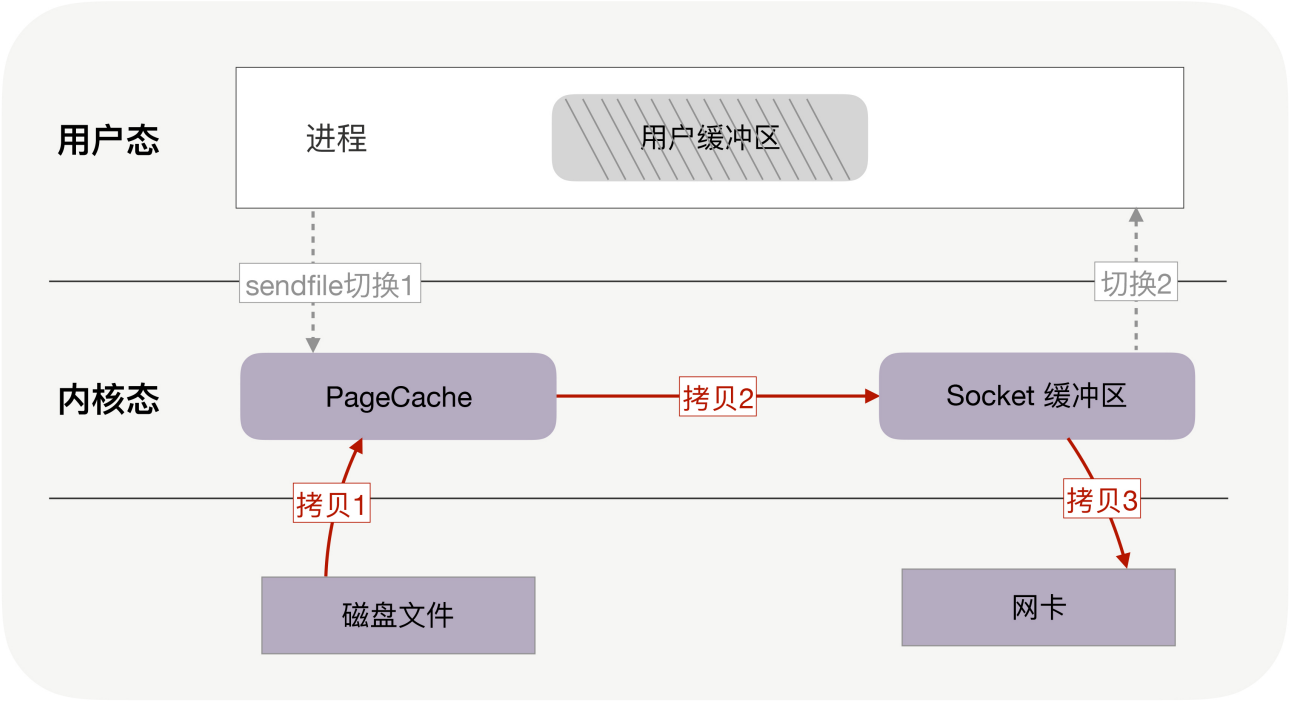
零拷贝可以做到用户空间和内核空间共用同一块内存（Java中的DirectBuffer），这样少做一次拷贝。普通Buffer是在JVM堆上分配的内存，而DirectBuffer是堆外分配的（内核和JVM可以同时读写），这样不需要再多一次内核到用户Buffer的拷贝



比如通过网络下载文件，普通拷贝的流程会复制4次并有4次上下文切换，上下文切换是因为读写慢导致了IO的阻塞，进而线程被内核挂起，所以发生了上下文切换。在极端情况下如果read/write没有导致阻塞是不会发生上下文切换的：



改成零拷贝后，也就是将read和write合并成一次，直接在内核中完成磁盘到网卡的数据复制



零拷贝就是操作系统提供的新函数(sendfile)，同时接收文件描述符和 TCP socket 作为输入参数，这样执行时就可以完全在内核态完成内存拷贝，既减少了内存拷贝次数，也降低了上下文切换次数。

而且，零拷贝取消了用户缓冲区后，不只降低了用户内存的消耗，还通过最大化利用 socket 缓冲区中的内存，间接地再一次减少了系统调用的次数，从而带来了大幅减少上下文切换次数的机会！

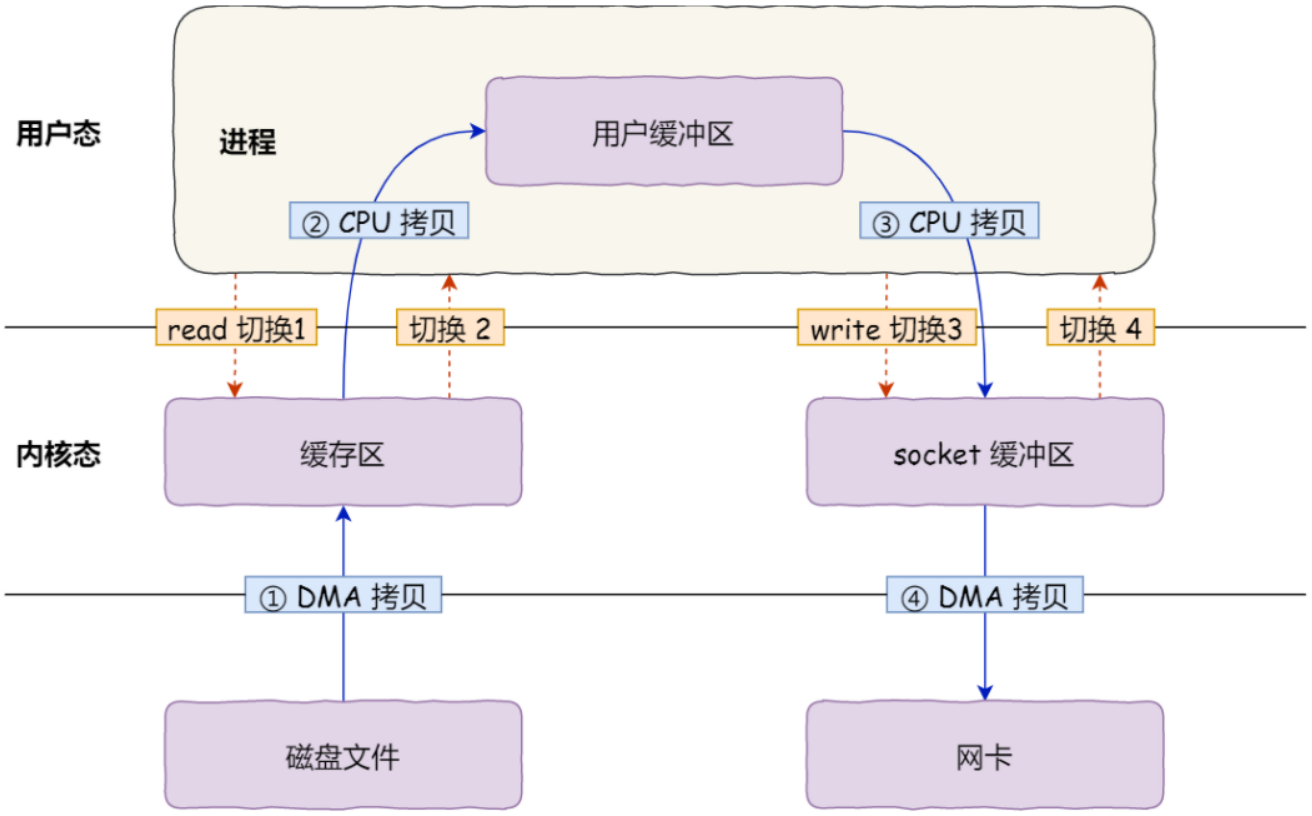
应用读取磁盘写入网络的时候还得考虑缓存的大小，一般会设置的比较小，这样一个大文件导致多次小批量的读取，每次读取伴随着多次上下文切换。

零拷贝使我们不必关心 socket 缓冲区的大小（socket缓冲区大小本身默认就是动态调整、或者应用代码指定大小）。比如，调用零拷贝发送方法时，尽可以把发送字节数设为文件的所有未发送字节数，例如 320MB，也许此时 socket 缓冲区大小为 1.4MB，那么一次性就会发送 1.4MB 到客户端，而不是只有 32KB。这意味着对于 1.4MB 的 1 次零拷贝，仅带来 2 次上下文切换，而不使用零拷贝且用户缓冲区为 32KB 时，经历了 176 次（4 * 1.4MB/32KB）上下文切换。

read+write 和零拷贝优化

```
1 read(file, tmp_buf, len);
2 write(socket, tmp_buf, len);
```

代码很简单，虽然就两行代码，但是这里面发生了不少的事情。

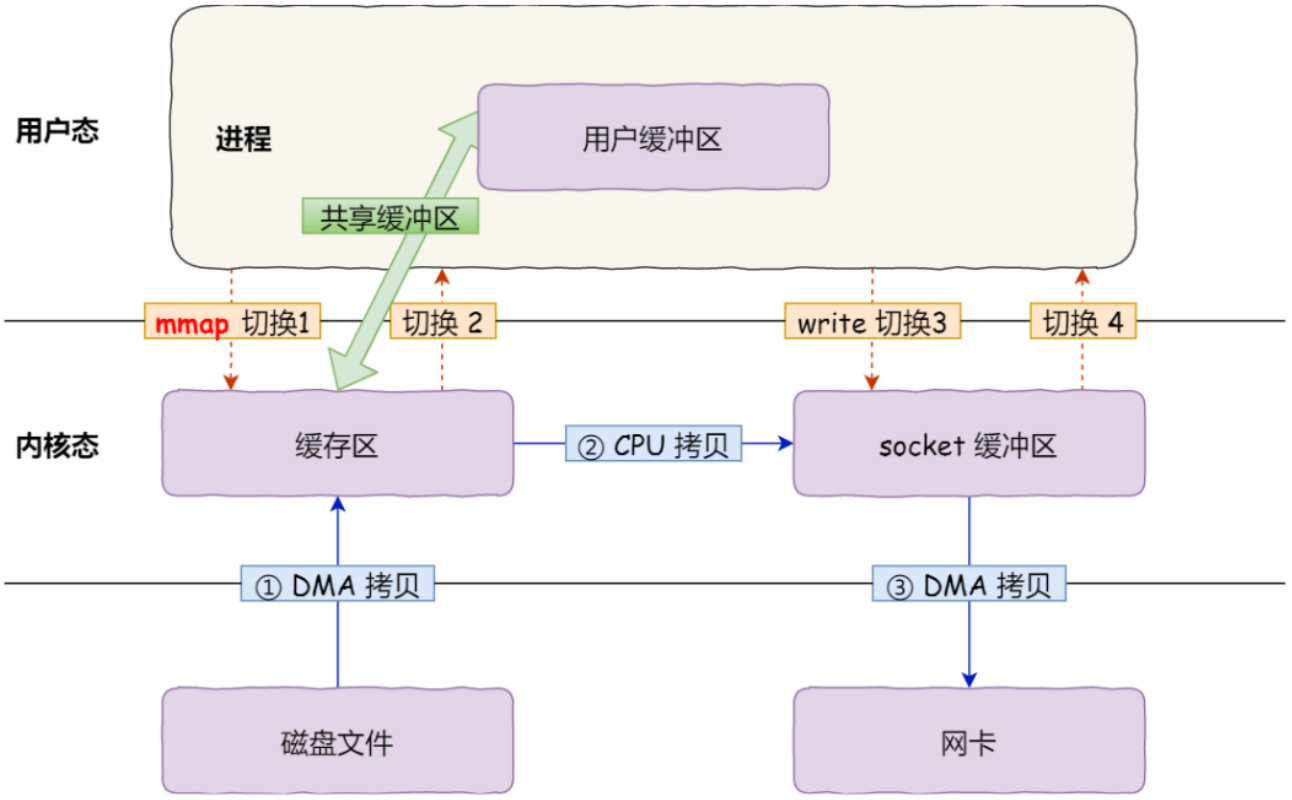


首先，期间共**发生了 4 次用户态与内核态的上下文切换**，因为发生了两次系统调用，一次是 `read()` ，一次是 `write()` ，每次系统调用都得先从用户态切换到内核态，等内核完成任务后，再从内核态切换回用户态。

通过mmap替换read优化一下

用 `mmap()` 替换原先的 `read()`， `mmap()` 也即是内存映射（memory map）：把用户进程空间的一段内存缓冲区（user buffer）映射到文件所在的内核缓冲区（kernel buffer）上。

> `mmap()` 系统调用函数会直接把内核缓冲区里的数据「**映射**」到用户空间，这样，操作系统内核与用户空间就不需要再进行任何的数据拷贝操作。



通过使用 `mmap()` 来代替 `read()`， 可以减少一次数据拷贝的过程。

但这还不是最理想的零拷贝，因为首先仍然需要通过 CPU 把内核缓冲区的数据拷贝到 socket 缓冲区里，而且仍然需要 4 次上下文切换，因为系统调用还是 2 次；另外内存映射技术是一个开销很大的虚拟存储操作：这种操作需要修改页表以及用内核缓冲区里的文件数据汰换掉当前 TLB 里的缓存以维持虚拟内存映射的一致性。

sendfile

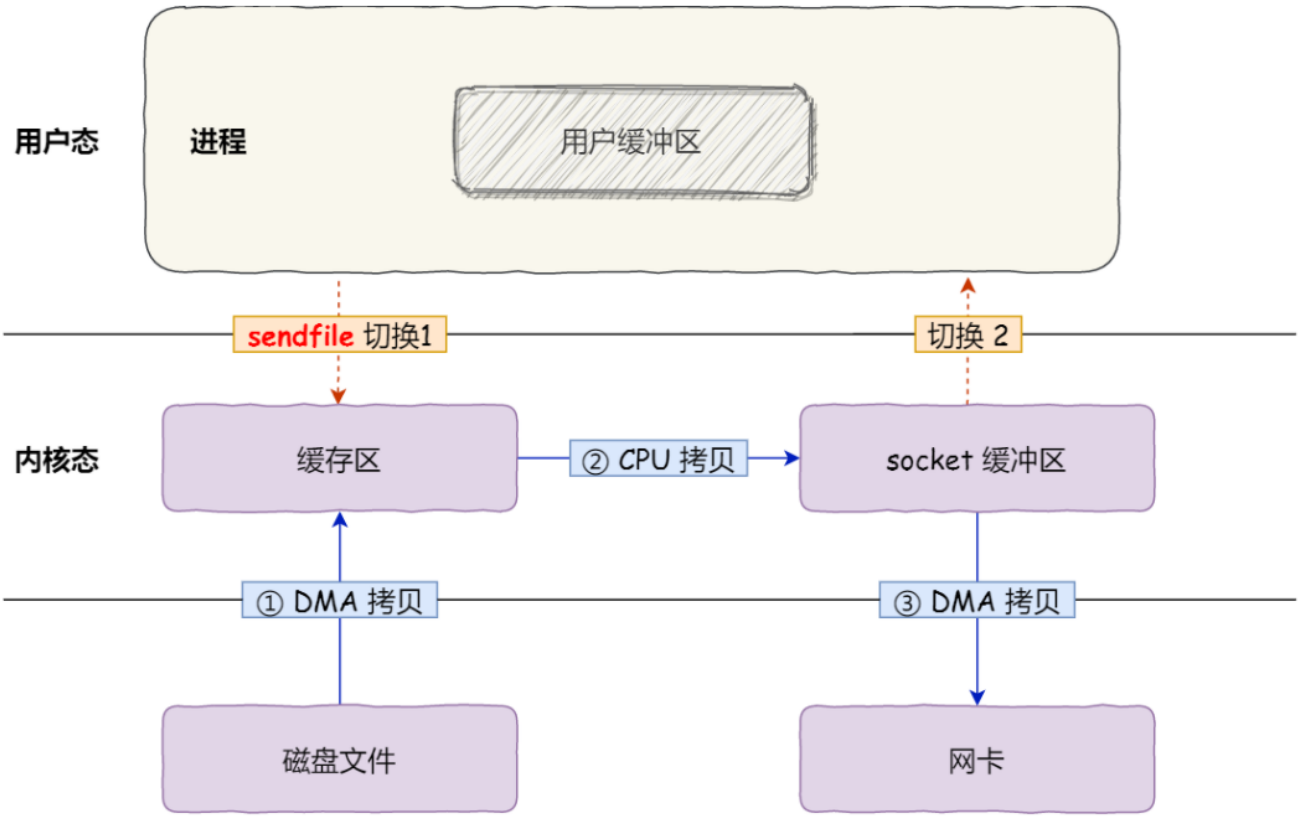
在 Linux 内核版本 2.1 中，提供了一个专门发送文件的系统调用函数 `sendfile()`，函数形式如下：

```
ssize_tsendfile(intout_fd,intin_fd,off_t*offset,size_tcount);
```

它的前两个参数分别是目的端和源端的文件描述符，后面两个参数是源端的偏移量和复制数据的长度，返回值是实际复制数据的长度。

首先，它可以替代前面的 `read()` 和 `write()` 这两个系统调用，这样就可以减少一次系统调用，也就减少了 2 次上下文切换的开销。

其次，该系统调用，可以直接把内核缓冲区里的数据拷贝到 socket 缓冲区里，不再拷贝到用户态，这样就只有 2 次上下文切换，和 3 次数据拷贝。如下图：



当然这里还是有一次CPU来拷贝内存的过程，仍然有文件截断的问题。`sendfile()` 依然是一个适用性很窄的技术，最适合的场景基本也就是一个静态文件服务器了。

然而 `sendfile()` 本身是有很大问题的，从不同的角度来看的话主要是：

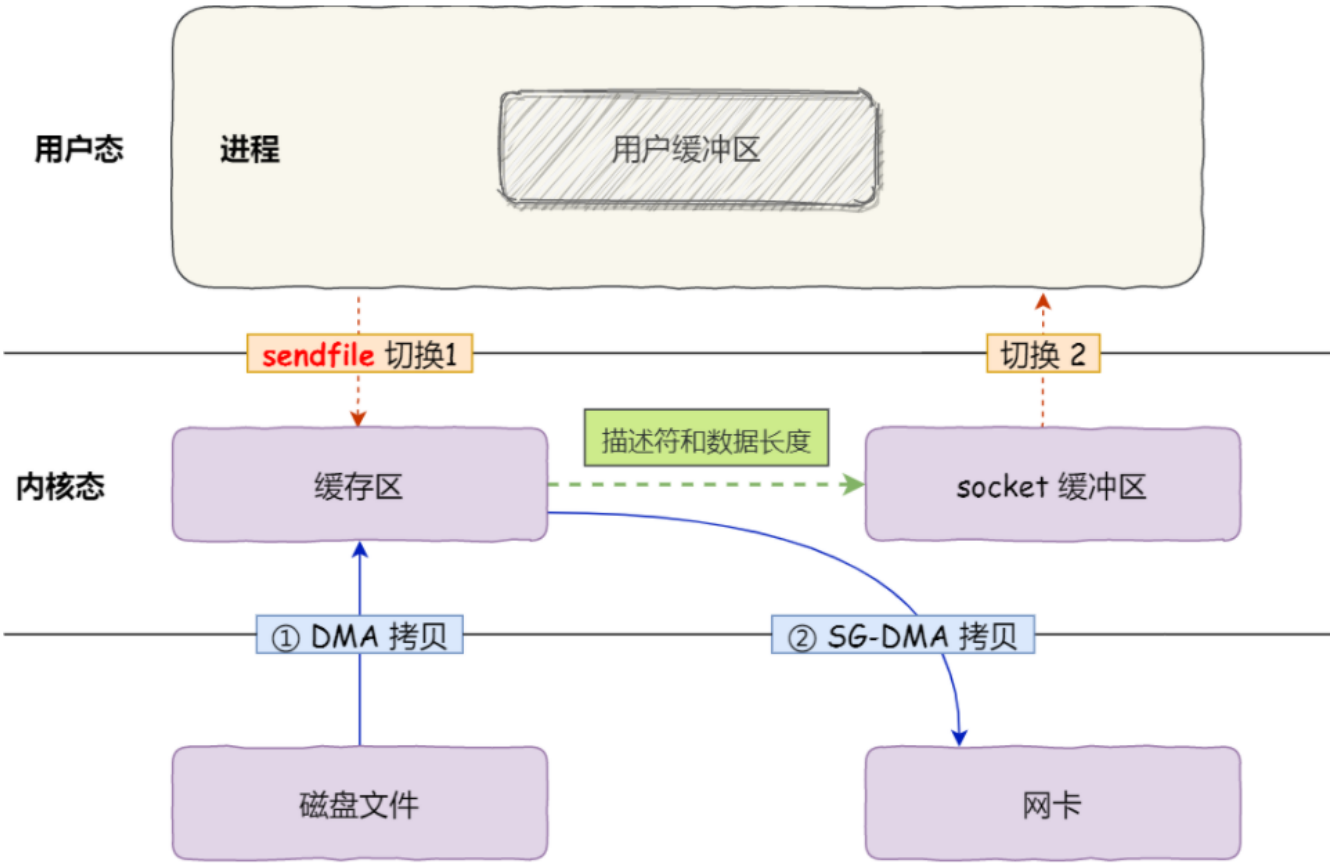
1. 首先一个是这个接口并没有进行标准化，导致 `sendfile()` 在 Linux 上的接口实现和其他类 Unix 系统的实现并不相同；
2. 其次由于网络传输的异步性，很难在接收端实现和 `sendfile()` 对接的技术，因此接收端一直没有实现对应的这种技术；
3. 最后从性能方面考量，因为 `sendfile()` 在把磁盘文件从内核缓冲区（page cache）传输到到套接字缓冲区的过程中依然需要 CPU 参与，这就很难避免 CPU 的高速缓存被传输的数据所污染。

SG-DMA（*The Scatter-Gather Direct Memory Access*）技术

上一小节介绍的 `sendfile()` 技术已经把一次数据读写过程中的 CPU 拷贝的降低至只有 1 次了，但是人永远是贪心和不知足的，现在如果想要把这仅有的一次 CPU 拷贝也去除掉，有没有办法呢？

当然有！通过引入一个新硬件上的支持，我们可以把这个仅剩的一次 CPU 拷贝也给抹掉：Linux 在内核 2.4 版本里引入了 DMA 的 scatter/gather – 分散/收集功能，并修改了 `sendfile()` 的代码使之和 DMA 适配。scatter 使得 DMA 拷贝可以不再需要把数据存储在一片连续的内存空间上，而是允许离散存储，gather 则能够让 DMA 控制器根据少量的元

信息：一个包含了内存地址和数据大小的缓冲区描述符，收集存储在各处的数据，最终还原成一个完整的网络包，直接拷贝到网卡而非套接字缓冲区，避免了最后一次的 CPU 拷贝：



如果网卡支持 SG-DMA（*The Scatter-Gather Direct Memory Access*）技术（和普通的 DMA 有所不同），我们可以进一步减少通过 CPU 把内核缓冲区里的数据拷贝到 socket 缓冲区的过程。

这就是所谓的**零拷贝（Zero-copy）技术，因为我们没有在内存层面去拷贝数据，也就是说全程没有通过 CPU 来搬运数据，所有的数据都是通过 DMA 来进行传输的。**数据传输过程就再也没有 CPU 的参与了，也因此 CPU 的高速缓存再不会被污染了，也不再需要 CPU 来计算数据校验和了，CPU 可以去执行其他的业务计算任务，同时和 DMA 的 I/O 任务并行，此举能极大地提升系统性能。

零拷贝技术的文件传输方式相比传统文件传输的方式，减少了 2 次上下文切换和数据拷贝次数，**只需要 2 次上下文切换和数据拷贝次数，就可以完成文件的传输，而且 2 次的数据拷贝过程，都不需要通过 CPU，2 次都是由 DMA 来搬运。**

所以，总体来看，**零拷贝技术可以把文件传输的性能提高至少一倍以上。**

splice()

`sendfile()` + DMA Scatter/Gather 的零拷贝方案虽然高效，但是也有两个缺点：

1. 这种方案需要引入新的硬件支持；
2. 虽然 `sendfile()` 的输出文件描述符在 Linux kernel 2.6.33 版本之后已经可以支持任意类型的文件描述符，但是输入文件描述符依然只能指向文件。

这两个缺点限制了 `sendfile()` + DMA Scatter/Gather 方案的适用场景。为此，Linux 在 2.6.17 版本引入了一个新的系统调用 `splice()`，它在功能上和 `sendfile()` 非常相似，

但是能够实现在任意类型的两个文件描述符时之间传输数据；而在底层实现上，`splice()`又比`sendfile()`少了一次 CPU 拷贝，也就是等同于`sendfile()`+ DMA Scatter/Gather，完全去除了数据传输过程中的 CPU 拷贝。

`splice()`所谓的写入数据到管道其实并没有真正地拷贝数据，而是玩了个 tricky 的操作：只进行内存地址指针的拷贝而不真正去拷贝数据。所以，数据`splice()`在内核中并没有进行真正的数据拷贝，因此`splice()`系统调用也是零拷贝。

还有一点需要注意，前面说过管道的容量是 16 个内存页，也就是 16 * 4KB = 64 KB，也就是说一次往管道里写数据的时候最好不要超过 64 KB，否则的话会`splice()`会阻塞住，除非在创建管道的时候使用的是`pipe2()`并通过传入`O_NONBLOCK`属性将管道设置为非阻塞。

send() with MSG_ZEROCOPY

Linux 内核在 2017 年的 v4.14 版本接受了来自 Google 工程师 Willem de Bruijn 在 TCP 网络报文的通用发送接口`send()`中实现的 zero-copy 功能（MSG_ZEROCOPY）的 patch，通过这个新功能，用户进程就能够把用户缓冲区的数据通过零拷贝的方式经过内核空间发送到网络套接字中去，这个新技术和前文介绍的几种零拷贝方式相比更加先进，因为前面几种零拷贝技术都是要求用户进程不能处理加工数据而是直接转发到目标文件描述符中去的。Willem de Bruijn 在他的论文里给出的压测数据是：采用 netperf 大包发送测试，性能提升 39%，而线上环境的数据发送性能则提升了 5%~8%，官方文档陈述说这个特性通常只在发送 10KB 左右大包的场景下才会有显著的性能提升。一开始这个特性只支持 TCP，到内核 v5.0 版本之后才支持 UDP。

这个技术是基于 redhat 红帽在 2010 年给 Linux 内核提交的 virtio-net zero-copy 技术之上实现的，至于底层原理，简单来说就是通过`send()`把数据在用户缓冲区中的分段指针发送到 socket 中去，利用 page pinning 页锁定机制锁住用户缓冲区的内存页，然后利用 DMA 直接在用户缓冲区通过内存地址指针进行数据读取，实现零拷贝

目前来说，这种技术的主要缺陷有：

1. 只适用于大文件（10KB 左右）的场景，小文件场景因为 page pinning 页锁定和等待缓冲区释放的通知消息这些机制，甚至可能比直接 CPU 拷贝更耗时；
2. 因为可能异步发送数据，需要额外调用`poll()`和`recvmsg()`系统调用等待 buffer 被释放的通知消息，增加代码复杂度，以及会导致多次用户态和内核态的上下文切换；
3. MSG_ZEROCOPY 目前只支持发送端，接收端暂不支持。

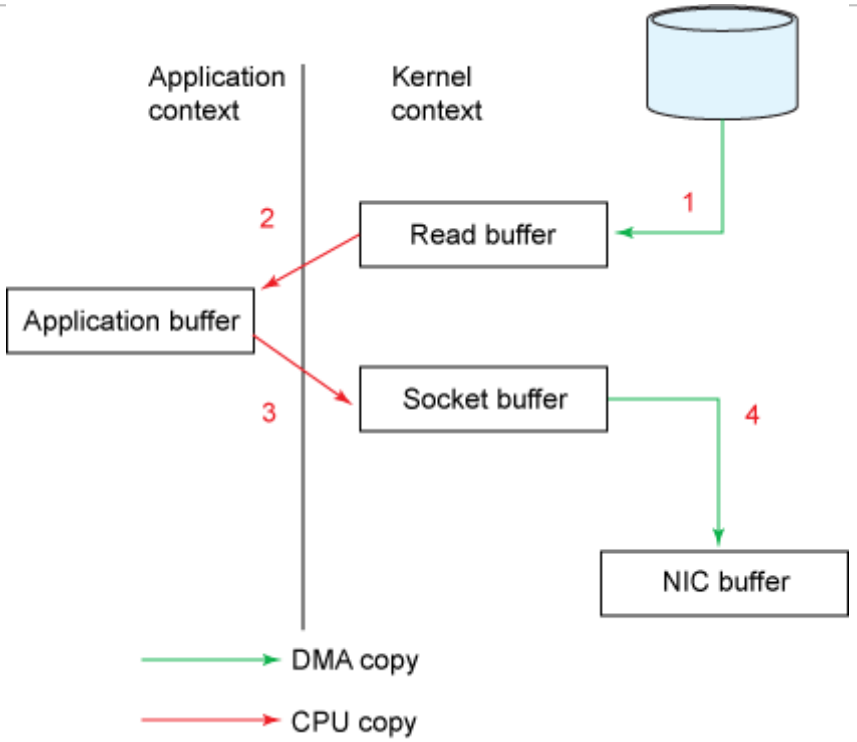
零拷贝应用

kafka就利用了「零拷贝」技术，从而大幅提升了 I/O 的吞吐率，这也是 Kafka 在处理海量数据为什么这么快的原因之一(利用磁盘顺序写；PageCache)。

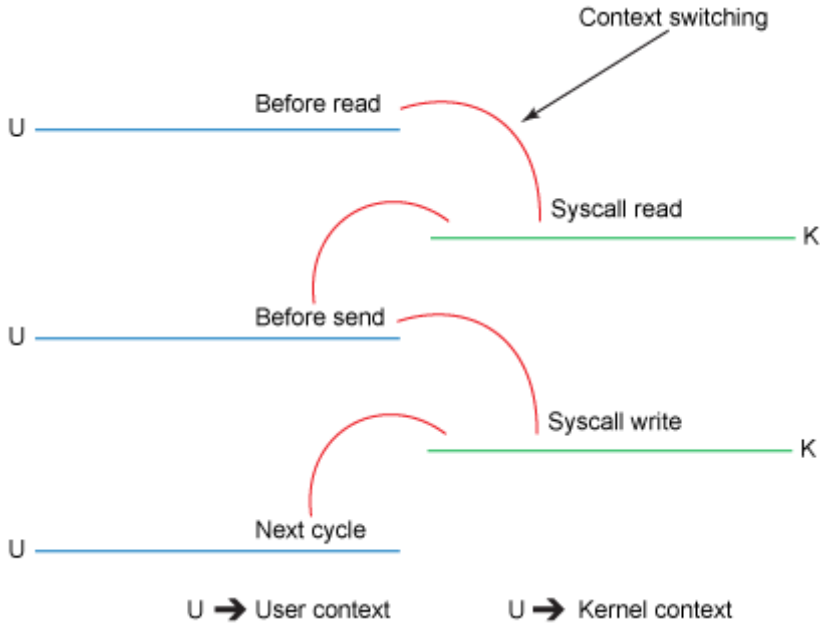
非零拷贝代码：

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

Traditional data copying approach:



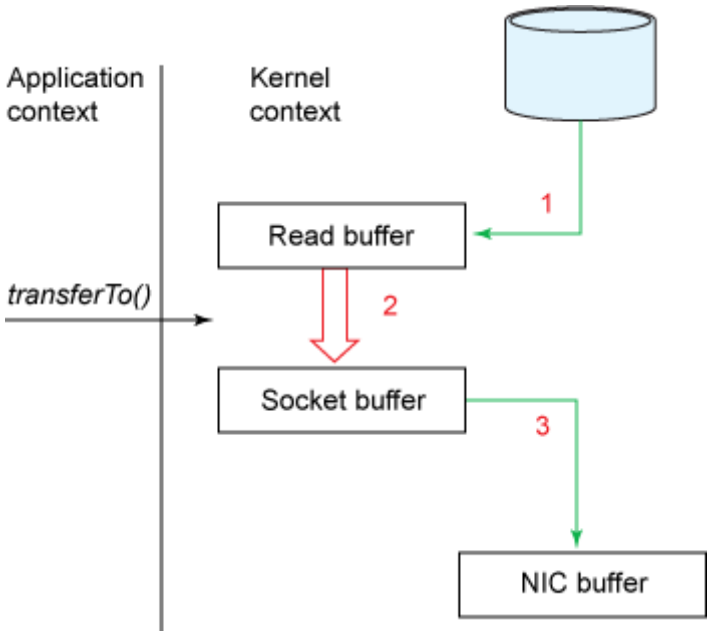
Traditional context switches:



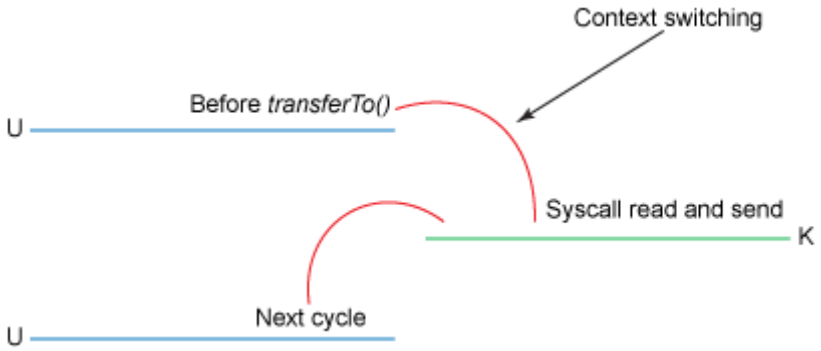
如果你追溯 Kafka 文件传输的代码，你会发现，最终它调用了 Java NIO 库里的`transferTo`方法：

```
longtransferFrom(FileChannel fileChannel,longposition,longcount)throwsIOExcept
returnfileChannel.transferTo(position, count, socketChannel);
}
```

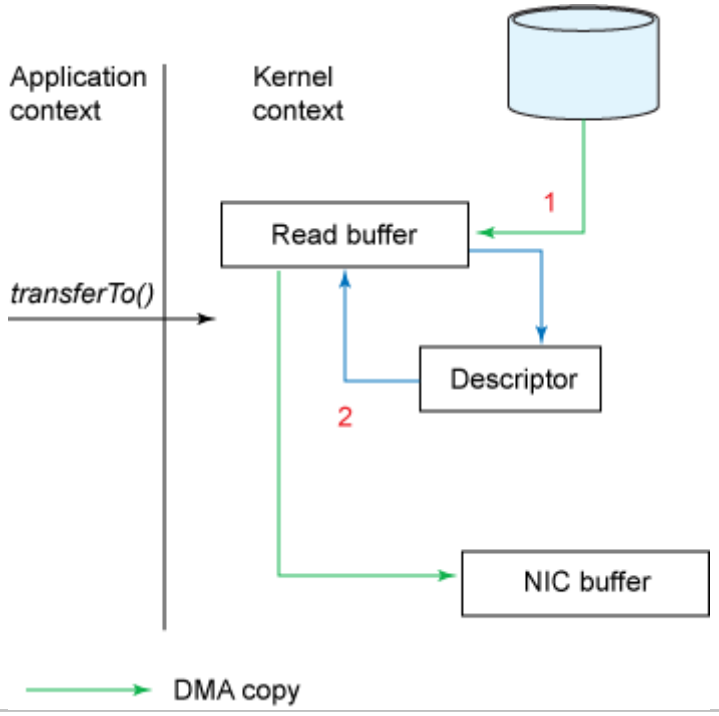
Data copy with transferTo()



Context switching with transferTo():



Data copies when transferTo() and gather operations are used



如果 Linux 系统支持 sendfile() 系统调用，那么 transferTo() 实际上最后就会使用到 sendfile() 系统调用函数。

Nginx 也支持零拷贝技术，一般默认是开启零拷贝技术，这样有利于提高文件传输的效率，是否开启零拷贝技术的配置如下：

```
http {
...
sendfile on
...
}
```

sendfile 配置的具体意思：

- 设置为 on 表示，使用零拷贝技术来传输文件：sendfile ，这样只需要 2 次上下文切换，和 2 次数据拷贝。
- 设置为 off 表示，使用传统的文件传输技术：read + write，这时就需要 4 次上下文切换，和 4 次数据拷贝。

如果是大文件很容易消耗非常多的PageCache，不推荐使用PageCache（或者说零拷贝），建议使用异步IO+直接IO。

在 nginx 中，我们可以用如下配置，来根据文件的大小来使用不同的方式：

```
location /video/ {
sendfile on;
aio on;
directio 1024m;
}
```

当文件大小大于 directio 值后，使用「异步 I/O + 直接 I/O」，否则使用「零拷贝技术」。

零拷贝性能

如果用零拷贝和不用零拷贝来做一个文件服务器，来对比下他们的性能

Performance comparison: Traditional approach vs. zero copy

File size	Normal file transfer (ms)	transferTo (ms)
7MB	156	45
21MB	337	128
63MB	843	387
98MB	1320	617
200MB	2124	1150
350MB	3631	1762
700MB	13498	4422
1GB	18399	8537

DMA

什么是 DMA 技术？简单理解就是，**在进行 I/O 设备和内存的数据传输的时候，数据搬运的工作全部交给 DMA 控制器，而 CPU 不再参与任何与数据搬运相关的事情，这样 CPU 就可以去处理别的事务**

RDMA

remote DMA

参考资料

https://www.atatech.org/articles/66885

https://cloud.tencent.com/developer/article/1087455

https://www.cnblogs.com/xiaolincoding/p/13719610.html

https://mp.weixin.qq.com/s/dZNjq05q9jMFYhJrjae_LA从Linux内存管理到零拷贝

Efficient data transfer through zero copy

CPU：一个故事看懂DMA