DMA

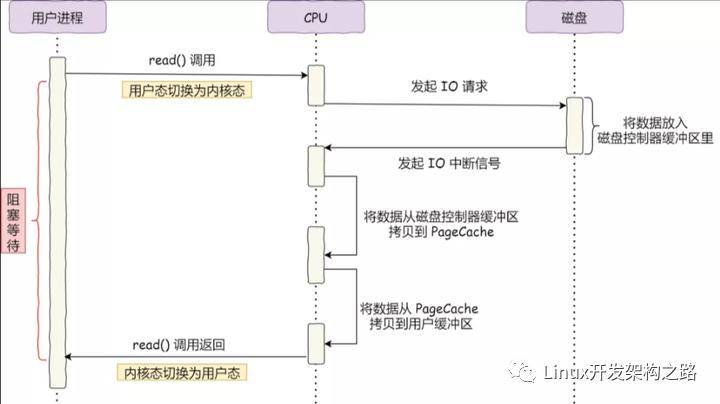
直接内存访问（Direct Memory Access）

什么是DMA？

在进行数据传输的时候，数据搬运的工作全部交给 DMA 控制器，而 CPU 不再参与，可以去干别的事情。

传统I/O

在没有 DMA 技术前，全程数据拷贝都需要CPU来做，严重消耗CPU。

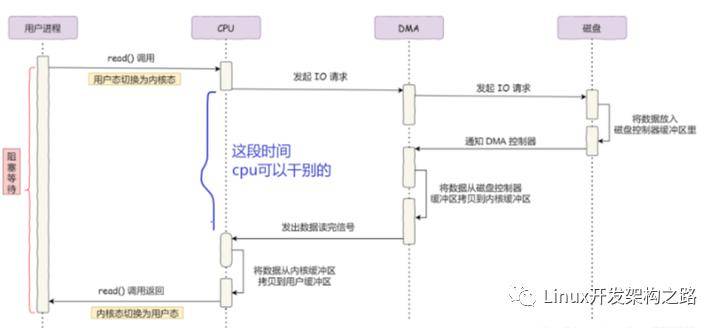


利用DMA的IO

利用DMA之后：

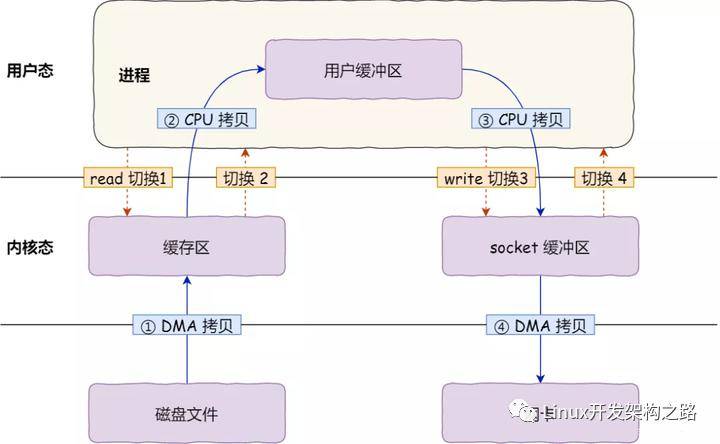
* 4次数据拷贝，其中DMA和CPU分别拷贝2次(CPU的时间多宝贵啊)
* 2次系统调用导致的4 次用户态与内核态的上下文切换

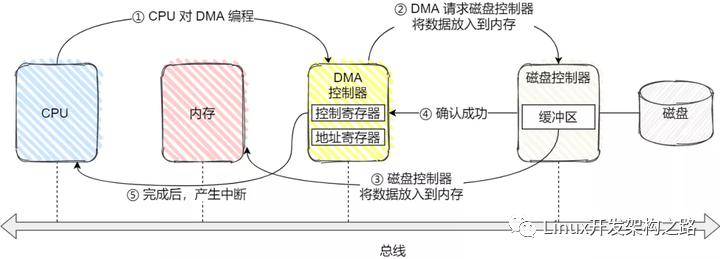
DMA 控制器进行数据传输的过程：



* 用户进程调用 read 方法，向操作系统发出 I/O 请求，请求读取数据到自己的用户缓冲区中，进程进入阻塞状态，用户态切换至内核态；
* 操作系统收到请求后，进一步将 I/O 请求发送 DMA，然后让 CPU 可以执行其他任务；
* DMA 进一步将 I/O 请求发送给磁盘；
* 磁盘收到 DMA 的 I/O 请求，把数据从磁盘读取到磁盘控制器的缓冲区中，当磁盘控制器的缓冲区被读满后，向 DMA 发起中断信号，告知自己缓冲区已满；
* DMA 收到磁盘的信号，将磁盘控制器缓冲区中的数据拷贝到内核缓冲区中，此时不占用 CPU，CPU 可以执行其他任务；
* 当 DMA 读取了足够多的数据，就会发送中断信号给 CPU；
* CPU 收到 DMA 的信号，知道数据已经准备好，于是将数据从内核拷贝到用户空间，系统调用返回，内核态切换至用户态；

利用DMA的IO完整流程图：





1、CPU 需对 DMA 控制器下发指令，告诉它想读取多少数据，读完的数据放在内存；

2、接下来，DMA 控制器会向磁盘控制器发出指令，通知它从磁盘读数据到其内部的缓冲区中，

3、接着磁盘控制器将缓冲区的数据传输到内存；

4、数据拷贝成功之后，磁盘控制器在总线上发出一个确认成功的信号到 DMA 控制器；

5、DMA 控制器收到信号后，DMA 控制器通过中断通知 CPU 指令完成，CPU 就可以直接取内存里面现成的数据了；

可以看到，仅仅在传送开始和结束时需要 CPU 干预，其他任务交由DMA处理。

因为发生了read+write两次系统调用，所以一共发生了 4 次用户态与内核态的上下文切换

上下文切换的成本并不小，一次切换需要耗时几十纳秒到几微秒

还发生了 4 次数据拷贝，其中两次是 CPU参与的拷贝。

如何优化？

减少「用户态与内核态的上下文切换」和「数据拷贝」的次数。

1、如何减少「用户态与内核态的上下文切换」的次数呢？

读取磁盘数据的时候，之所以要发生上下文切换，是因为用户空间没有权限操作磁盘或网卡，这些操作设备的过程只能交由OS内核来完成。所以需要系统调用进行上下文切换，切换到内核态。

所以，减少上下文切换到次数的办法就是：

减少系统调用的次数

2、如何减少「数据拷贝」的次数？

从内核的读缓冲区-----用户的缓冲区里----- socket 的缓冲区里，这个过程是没有必要的。

因为文件传输的应用场景中，在用户空间我们并不会对数据「再加工」，所以数据实际上可以不用搬运到用户空间，因此用户的缓冲区是没有必要存在的。

零拷贝

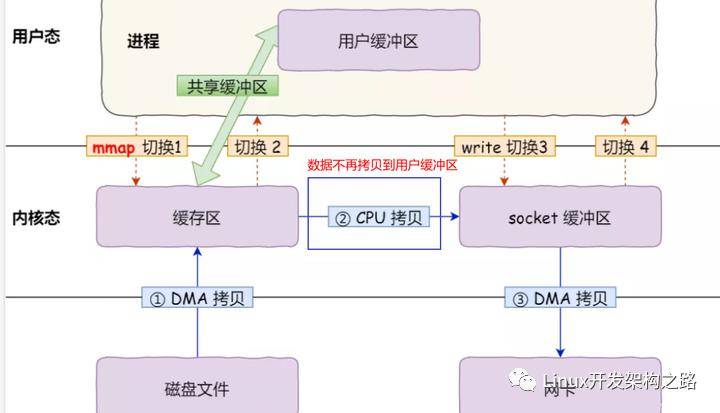
零拷贝技术实现的方式通常有 2 种：

* mmap(内存映射) + write
* sendfile

mmap + write

在前面我们知道，read 系统调用的过程中会把内核缓冲区的数据拷贝到用户的缓冲区里，于是为了减少这一步开销，我们可以用 mmap 替换 read 系统调用函数。

mmap系统调用函数会直接把内核缓冲区里的数据共享到用户空间，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。



具体过程如下：

* 应用进程调用了 mmap后，DMA 会把磁盘的数据拷贝到内核的缓冲区里。接着，应用进程跟操作系统内核「共享」这个缓冲区；
* 应用进程再调用 write，操作系统直接将内核缓冲区的数据拷贝到 socket 缓冲区中，这一切都发生在内核态，由 CPU 来搬运数据；
* 最后，把内核的 socket 缓冲区里的数据，拷贝到网卡的缓冲区里，这个过程是由 DMA 搬运的。

性能如何？

* 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 把内核缓冲区的数据拷贝到 socket 缓冲区里的工作仍然需要通过CPU完成，
* 而且仍然需要 4 次上下文切换，因为系统调用还是 2 次。

mmap详解

是什么？

mmap是一种实现内存映射文件的方法。

即：将一个文件映射到用户进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对映关系。

实现这样的映射关系后，进程就可以采用指针的方式读写操作这一段内存，而系统会自动回写脏页面到对应的文件磁盘上，即完成了对文件的操作，又不必再调用read,write等系统调用函数。

相应地，内核空间对这段区域的修改也直接反映到用户空间，从而可以实现不同用户进程间的文件共享。

mmap内存映射的实现过程，总的来说可以分为三个阶段：

1、进程启动映射过程，并在虚拟地址空间中为映射创建虚拟映射区域

2、调用mmap实现文件的物理地址和进程虚拟地址的一一映射关系

注：前两个阶段仅在于创建虚拟区间并完成地址映射，还没有将任何文件数据拷贝至主存。真正的文件读取是当进程发起读或写操作时开始。

3、进程发起对这片映射空间的访问，引发缺页中断，实现文件到内核缓冲区的拷贝

* 进程的读或写操作访问虚拟地址空间这一段映射地址，通过查询页表，发现这一段地址并不在物理页面上。因为目前只建立了地址映射，真正的硬盘数据还没有拷贝到内存中，因此引发缺页异常。
* 缺页异常进行一系列判断，确定无非法操作后，内核发起请求调页过程。
* 调页完成后。进程即可对这片内核缓冲区进行读写操作，如果写操作改变了其内容，一定时间后系统会自动回写脏页面到对应磁盘地址，也即完成了写入到文件的过程。

mmap的功能：

1、上面已经分析了，mmap最大的功能就是减少了数据的拷贝次数

2、提供了进程间共享内存及相互通信的方式。

不管是父子进程还是无亲缘关系的进程，都可以将自身用户空间映射到同一个文件。从而通过各自对映射区域的改动，达到进程间通信和进程间共享的目的。

同时，如果进程A和进程B都映射了区域C，当A第一次读取C时通过缺页从磁盘复制文件页到内存中；但当B再读C的相同页面时，虽然也会产生缺页异常，但是不再需要从磁盘中复制文件过来，而可直接使用已经保存在内存中的文件数据。

3、可用于实现高效的大规模数据传输。

内存空间不足，是制约大数据操作的一个方面，解决方案往往是借助硬盘空间协助操作，补充内存的不足。但是进一步会造成大量的文件I/O操作，极大影响效率。这个问题可以通过mmap映射很好的解决。换句话说，但凡是需要用磁盘空间代替内存的时候，mmap都可以发挥其功效。

sendfile

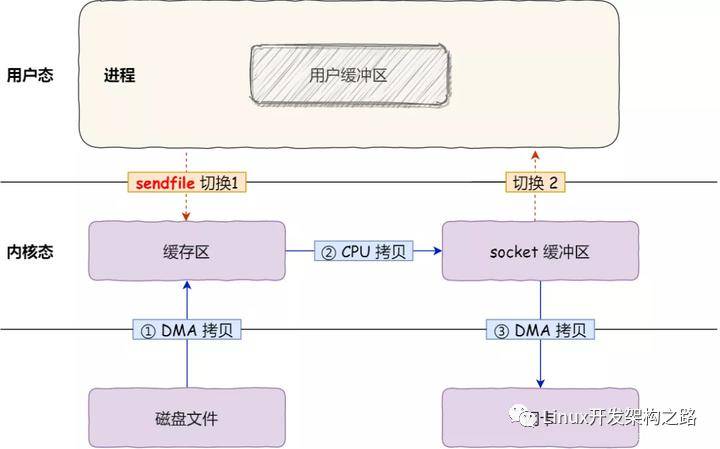
3次数据拷贝，其中CPU拷贝一次 1次系统调用 2 次用户态与内核态的上下文切换

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。

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，这样就减少了一次数据拷贝，

现在一共只有 2 次上下文切换，和 3 次数据拷贝。如下图：



但是这还不是真正的零拷贝技术。

真正的零拷贝

2次数据拷贝，无CPU参与拷贝

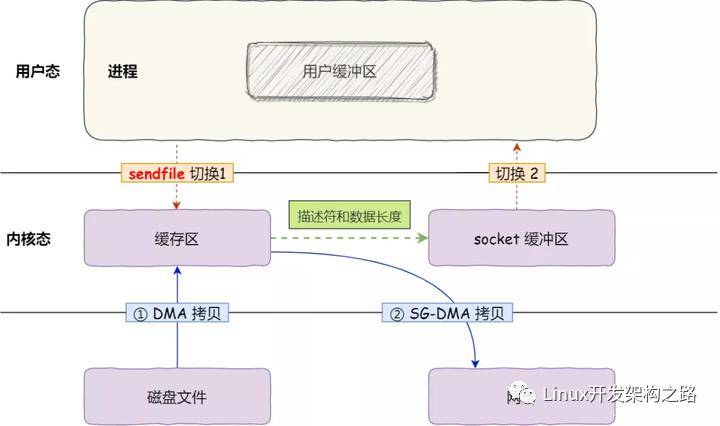
1次系统调用

2 次用户态与内核态的上下文切换

从 Linux 内核 2.4 版本开始起，sendfile 系统调用的过程发生了点变化，具体过程如下：

* 通过 DMA 将磁盘上的数据拷贝到内核缓冲区里；
* 只将缓冲区描述符和数据长度传到 socket 缓冲区，而内核缓存中的数据则通过网卡的 SG-DMA 控制器直接拷贝到网卡的缓冲区里，这样就减少了一次数据拷贝；

所以，这个过程之中，只进行了 2 次数据拷贝，如下图：



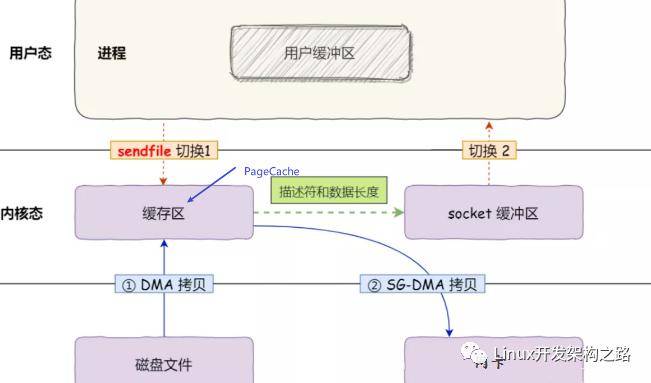
性能如何？

* 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。
* 只需要 2 次上下文切换和2次数据拷贝，就可以完成文件的传输，

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。

kafka和Nginx都使用了零拷贝技术

为什么需要内核缓存区？



现在回过头再来看，为什么不直接将磁盘数据拷贝到网卡，而要在中间加一个内核缓存区呢？——核心原因是磁盘读写太慢了

内核缓存区做了什么？

* 缓存最近被访问的数据；
* 预读功能；
* 内核的 I/O 调度算法会缓存尽可能多的 I/O 请求在 内核缓存区中，最后「合并」成一个更大的 I/O 请求再发给磁盘，这样做是为了减少磁盘的寻址操作；

1、缓存最近被访问的数据

最近访问过的数据接下来很可能还会被访问，所以利用PageCache 缓存最近被访问的数据，读磁盘数据的时候，优先在 PageCache 找，如果数据存在则可以直接返回；如果没有，则从磁盘中读取，然后缓存在 PageCache 中。当PageCache的空间不足时，淘汰最久未被访问的缓存。

2、预读功能

利用空间局部性原理，假设 read 方法每次只会读32 KB的字节，虽然 read 刚开始只会读0 ～ 32 KB的字节，但内核会把其后面的 32～64 KB 也读取到 PageCache，这样后面读取 32～64 KB 的成本就很低，如果在 32～64 KB 淘汰出 PageCache 前，进程读取到它了，收益就非常大。

这两个做法都在于解决读写磁盘相比读写内存的速度慢太多了这一痛点，大大提高了读写磁盘的性能。

所以零拷贝使用 内核缓存区技术进一步提升性能。

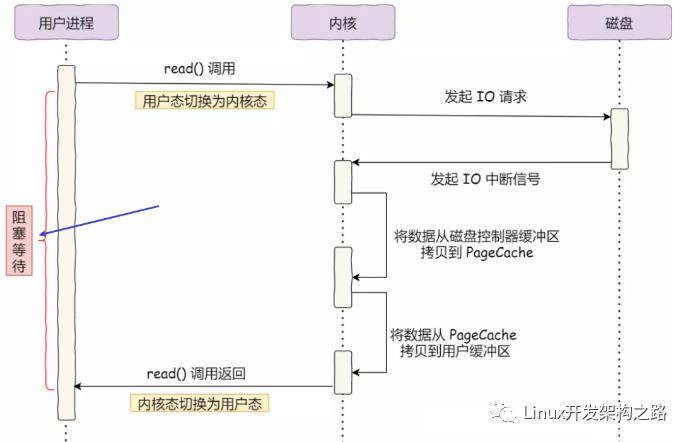
但是由于内核缓存区不适合传输大文件，所以零拷贝不适合传输大文件 因为每当用户访问这些大文件的时候，内核就会把它们载入 内核缓存区中，于是 内核缓存区空间很快被这些大文件占满。其他「热点」的小文件可能就无法充分使用到 内核缓存区，于是这样磁盘读写的性能就会下降了；

所以，内核缓存区中的大文件数据，不但没有享受到缓存带来的好处，却还耗费 DMA 多拷贝到 内核缓存区一次；

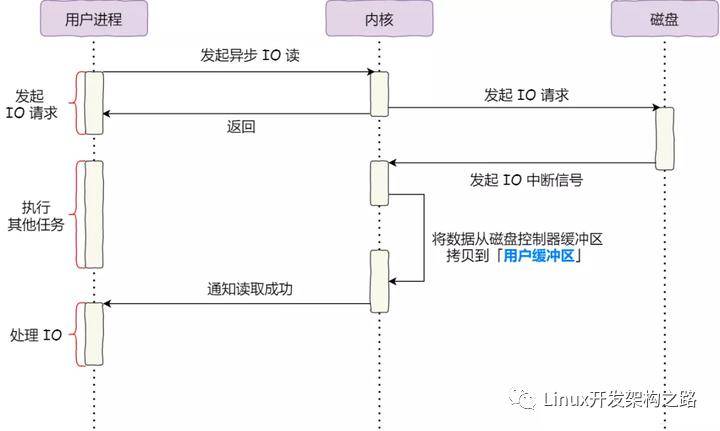
那针对大文件的传输，我们应该使用什么方式呢？

大文件传输：异步IO+直接IO

回顾最初的例子，当调用 read 方法读取文件时，进程实际上会阻塞在 read 方法调用，因为要等待磁盘数据的返回，如下图：



对于阻塞的问题，可以用异步 I/O来解决，它的工作方式如下图：

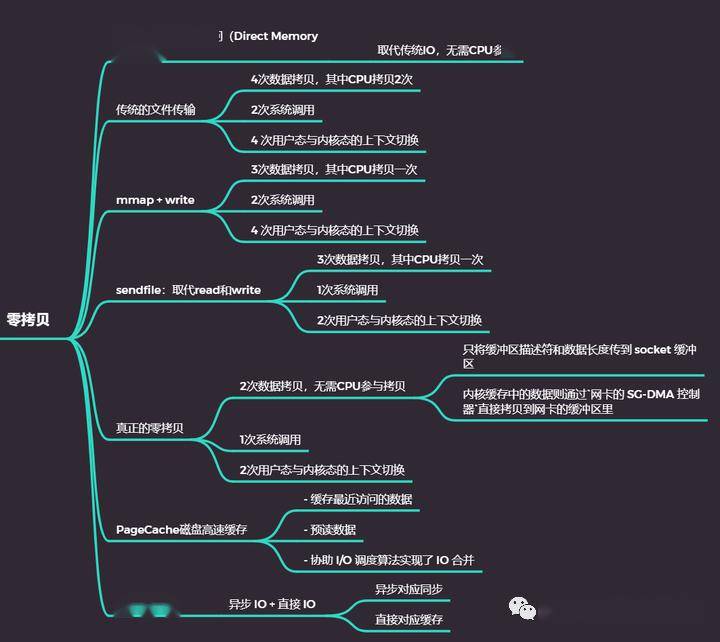


可以发现，异步 I/O 并没有涉及到 内核缓存区。

绕开 内核缓存区的 I/O 叫直接 I/O，使用 内核缓存区的 I/O 则叫缓存 I/O。通常，对于磁盘，异步 I/O 只支持直接 I/O。

所以，针对大文件的传输的方式，应该使用异步 I/O + 直接 I/O来替代零拷贝技术。

总结



DMA和传统IO

早期 I/O 操作，内存与磁盘的数据传输的工作都是由 CPU 完成的，而此时 CPU 不能执行其他任务，会特别浪费 CPU 资源。

于是，为了解决这一问题，DMA 技术就出现了，实际数据传输工作由 DMA控制器来完成，CPU 不需要参与数据传输的工作。

零拷贝

传统 IO的工作方式，从硬盘读取数据，然后再通过网卡向外发送，需要进行 4次上下文切换，和 4 次数据拷贝，更糟糕的是其中两次都是CPU完成的。

为了提高文件传输的性能，于是就出现了零拷贝技术，只有一个sendfile系统调用导致的2 次用户态与内核态的上下文切换，只进行了2 次数据拷贝(磁盘——pageCache——网卡)，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。

需要注意的是，零拷贝技术中，数据没有进入用户缓冲区，所以用户进程无法对文件内容作进一步的加工的，比如压缩数据再发送。

内核缓存区

零拷贝技术是基于 内核缓存区的，内核缓存区具有

* 缓存最近访问的数据
* 预读数据
* 协助 I/O 调度算法实现了 IO 合并

提升了访问缓存数据的性能，解决了磁盘IO慢的问题，进一步提升了零拷贝的性能。

大文件传输

当传输大文件时，不能使用零拷贝，因为可能由于 内核缓存区 被大文件占据，而导致其他的「热点」小文件无法利用到 内核缓存区，并且大文件的缓存命中率不高，这时就需要使用「异步 IO + 直接 IO 」的方式。