Linux I/O 原理和 Zero-copy 技术全面揭秘

腾讯技术...
已认证帐号

作者：allanpan，腾讯 IEG 后台开发工程师

> 两万字长文从虚拟内存、I/O 缓冲区，用户态&内核态以及 I/O 模式等等知识点全面而又详尽地剖析 Linux 系统的 I/O 底层原理，分析了 Linux 传统的 I/O 模式的弊端，进而引入 Linux Zero-copy 零拷贝技术的介绍和原理解析，将零拷贝技术和传统的 I/O 模式进行区分和对比，帮助读者理解 Linux 内核对 I/O 模块的优化改进思路。 全网最深度和详尽的 Linux I/O 及零拷贝技术的解析文章

导言

如今的网络应用早已从 CPU 密集型转向了 I/O 密集型，网络服务器大多是基于 `C-S` 模型，也即 `客户端 - 服务端` 模型，客户端需要和服务端进行大量的网络通信，这也决定了现代网络应用的性能瓶颈：I/O。

传统的 Linux 操作系统的标准 I/O 接口是基于数据拷贝操作的，即 I/O 操作会导致数据在操作系统内核地址空间的缓冲区和用户进程地址空间定义的缓冲区之间进行传输。设置缓冲区最大的好处是可以减少磁盘 I/O 的操作，如果所请求的数据已经存放在操作系统的高速缓冲存储器中，那么就不需要再进行实际的物理磁盘 I/O 操作；然而传统的 Linux I/O 在数据传输过程中的数据拷贝操作深度依赖 CPU，也就是说 I/O 过程需要 CPU 去执行数据拷贝的操作，因此导致了极大的系统开销，限制了操作系统有效进行数据传输操作的能力。

I/O 是决定网络服务器性能瓶颈的关键，而传统的 Linux I/O 机制又会导致大量的数据拷贝操作，损耗性能，所以我们亟需一种新的技术来解决数据大量拷贝的问题，这个答案就是零拷贝(Zero-copy)。

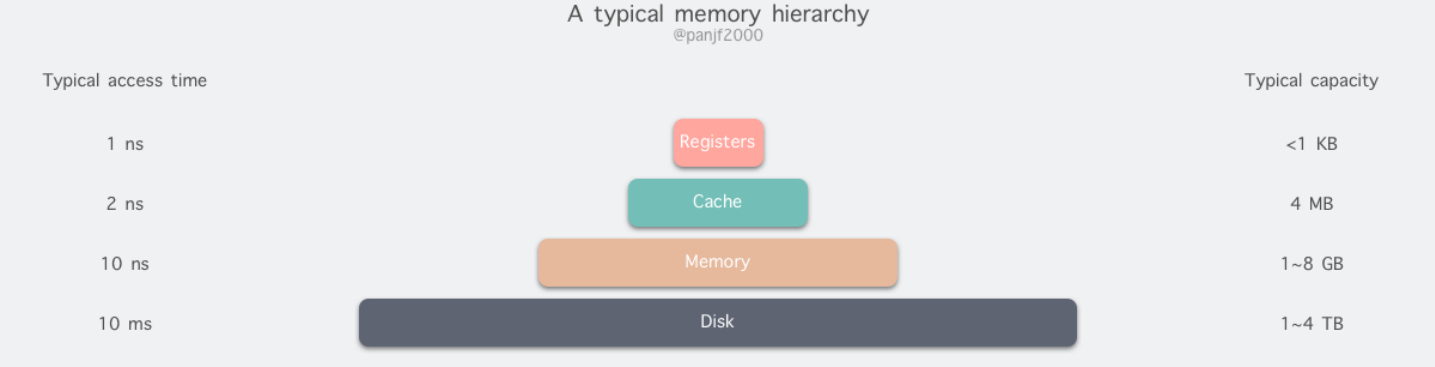
计算机存储器

既然要分析 Linux I/O，就不能不了解计算机的各类存储器。

存储器是计算机的核心部件之一，在完全理想的状态下，存储器应该要同时具备以下三种特性：

1. 速度足够快：存储器的存取速度应当快于 CPU 执行一条指令，这样 CPU 的效率才不会受限于存储器
2. 容量足够大：容量能够存储计算机所需的全部数据
3. 价格足够便宜：价格低廉，所有类型的计算机都能配备

但是现实往往是残酷的，我们目前的计算机技术无法同时满足上述的三个条件，于是现代计算机的存储器设计采用了一种分层次的结构：



从顶至底，现代计算机里的存储器类型分别有：寄存器、高速缓存、主存和磁盘，这些存储器的速度逐级递减而容量逐级递增 。存取速度最快的是寄存器，因为寄存器的制作材料和 CPU 是相同的，所以速度和 CPU 一样快，CPU 访问寄存器是没有时延的，然而因为价格昂贵，因此容量也极小，一般 32 位的 CPU 配备的寄存器容量是 32⨉32 Bit，64 位的 CPU 则是 64⨉64 Bit，不管是 32 位还是 64 位，寄存器容量都小于 1 KB，且寄存器也必须通过软件自行管理。

第二层是高速缓存，也即我们平时了解的 CPU 高速缓存 L1、L2、L3，一般 L1 是每个 CPU 独享，L3 是全部 CPU 共享，而 L2 则根据不同的架构设计会被设计成独享或者共享两种模式之一，比如 Intel 的多核芯片采用的是共享 L2 模式而 AMD 的多核芯片则采用的是独享 L2 模式。

第三层则是主存，也即主内存，通常称作随机访问存储器（Random Access Memory, RAM）。是与 CPU 直接交换数据的内部存储器。它可以随时读写（刷新时除外），而且速度很快，通常作为操作系统或其他正在运行中的程序的临时资料存储介质。

最后则是磁盘，磁盘和主存相比，每个二进制位的成本低了两个数量级，因此容量比之会大得多，动辄上 GB、TB，而问题是访问速度则比主存慢了大概三个数量级。机械硬盘速度慢主要是因为机械臂需要不断在金属盘片之间移动，等待磁盘扇区旋转至磁头之下，然后才能进行读写操作，因此效率很低。

主内存是操作系统进行 I/O 操作的重中之重，绝大部分的工作都是在用户进程和内核的内存缓冲区里完成的，因此我们接下来需要提前学习一些主存的相关原理。

物理内存

我们平时一直提及的物理内存就是上文中对应的第三种计算机存储器，RAM 主存，它在计算机中以内存条的形式存在，嵌在主板的内存槽上，用来加载各式各样的程序与数据以供 CPU 直接运行和使用。

虚拟内存

在计算机领域有一句如同摩西十诫般神圣的哲言："**计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决**"，从内存管理、网络模型、并发调度甚至是硬件架构，都能看到这句哲言在闪烁着光芒，而虚拟内存则是这一哲言的完美实践之一。

虚拟内存是现代计算机中的一个非常重要的存储器抽象，主要是用来解决应用程序日益增长的内存使用需求：现代物理内存的容量增长已经非常快速了，然而还是跟不上应用程序对主存需求的增长速度，对于应用程序来说内存还是不够用，因此便需要一种方法来解决这两者之间的容量差矛盾。

计算机对多程序内存访问的管理经历了 `静态重定位` --> `动态重定位` --> `交换(swapping)技术` --> `虚拟内存`，最原始的多程序内存访问是直接访问绝对内存地址，这种方式几乎是完全不可用的方案，因为如果每一个程序都直接访问物理内存地址的话，比如两个程序并发执行以下指令的时候：

```
mov cx, 2
mov bx, 1000H
mov ds, bx
mov [0], cx

...

mov ax, [0]
add ax, ax
```

这一段汇编表示在地址 1000:0 处存入数值 2，然后在后面的逻辑中把该地址的值取出来乘以 2，最终存入 ax 寄存器的值就是 4，如果第二个程序存入 cx 寄存器里的值是 3，那么并发执行的时候，第一个程序最终从 ax 寄存器里得到的值就可能是 6，这就完全错误了，得到脏数据还顶多算程序结果错误，要是其他程序往特定的地址里写入一些危险的指令而被另一个程序取出来执行，还可能会导致整个系统的崩溃。所以，为了确保进程间互不干扰，每一个用户进程都需要实时知晓当前其他进程在使用哪些内存地址，这对于写程序的人来说无疑是一场噩梦。

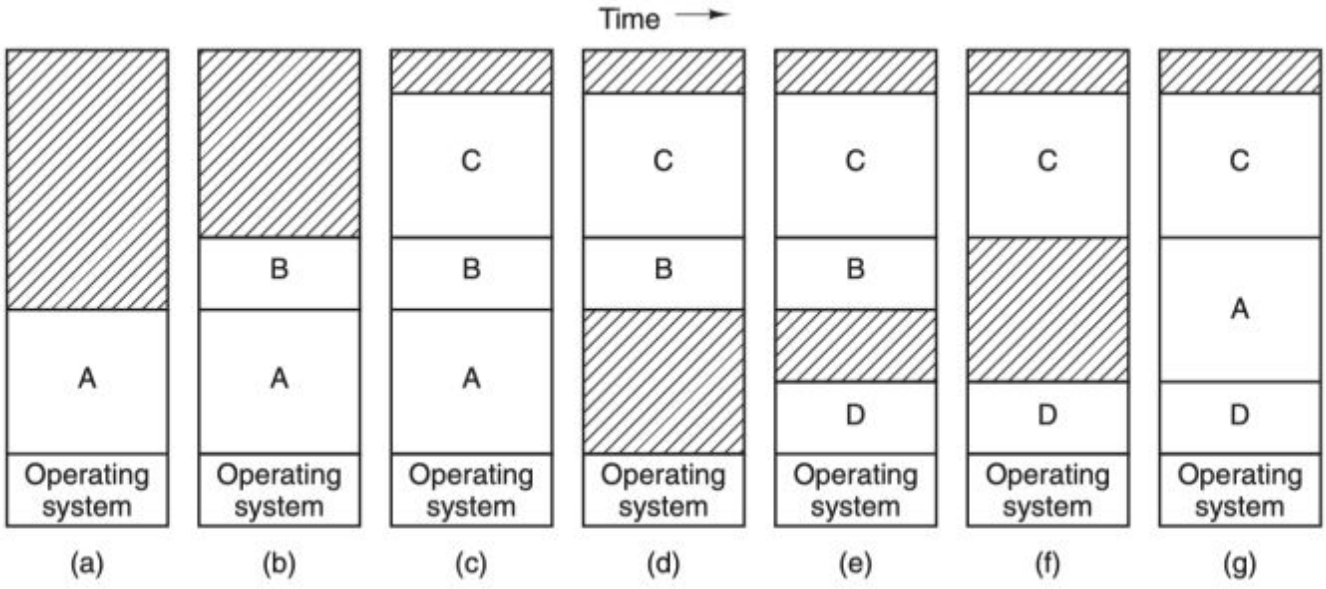
因此，操作绝对内存地址是完全不可行的方案，那就只能用操作相对内存地址，我们知道每个进程都会有自己的进程地址，从 0 开始，可以通过相对地址来访问内存，但是这同样有问题，还是前面类似的问题，比如有两个大小为 16KB 的程序 A 和 B，现在它们都被加载进了内存，内存地址段分别是 0 ~ 16384，16384 ~ 32768。A 的第一条指令是 `jmp 1024`，而在地址 1024 处是一条 `mov` 指令，下一条指令是 `add`，基于前面的 `mov` 指令做加法运算，与此同时，B 的第一条指令是 `jmp 1028`，本来在 B 的相对地址 1028 处应该也是一条 `mov` 去操作自己的内存地址上的值，但是由于这两个程序共享了段寄存器，因此虽然他们使用了各自的相对地址，但是依然操作的还是绝对内存地址，于是 B 就会跳去执行 `add` 指令，这时候就会因为非法的内存操作而 crash。

有一种 `静态重定位` 的技术可以解决这个问题，它的工作原理非常简单粗暴：当 B 程序被加载到地址 16384 处之后，把 B 的所有相对内存地址都加上 16384，这样的话当 B 执行 `jmp 1028` 之时，其实执行的是 `jmp 1028+16384`，就可以跳转到正确的内存地址处去执行正确的指令了，但是这种技术并不通用，而且还会对程序装载进内存的性能有影响。

再往后，就发展出来了存储器抽象：地址空间，就好像进程是 CPU 的抽象，地址空间则是存储器的抽象，每个进程都会分配独享的地址空间，但是独享的地址空间又带来了新的问题：如何实现不同进程的相同相对地址指向不同的物理地址？最开始是使用 `动态重定位` 技术来实现，这是用一种相对简单的地址空间到物理内存的映射方法。基本原理就是为每一个 CPU 配备两个特殊的硬件寄存器：基址寄存器和界限寄存器，用来动态保存每一个程序的起始物理内存地址和长度，比如前文中的 A，B 两个程序，当 A 运行时基址寄存器和界限寄存器就会分别存入 0 和 16384，而当 B 运行时则两个寄存器又会分别存入 16384 和 32768。然后每次访问指定的内存地址时，CPU 会在把地址发往内存总线之前自动把基址寄存器里的值加到该内存地址上，得到一个真正的物理内存地址，同时还会根据界限寄存器里的值检查该地址是否溢出，若是，则产生错误中止程序，`动`

态重定位 技术解决了 静态重定位 技术造成的程序装载速度慢的问题，但是也有新问题：每次访问内存都需要进行加法和比较运算，比较运算本身可以很快，但是加法运算由于进位传递时间的问题，除非使用特殊的电路，否则会比较慢。

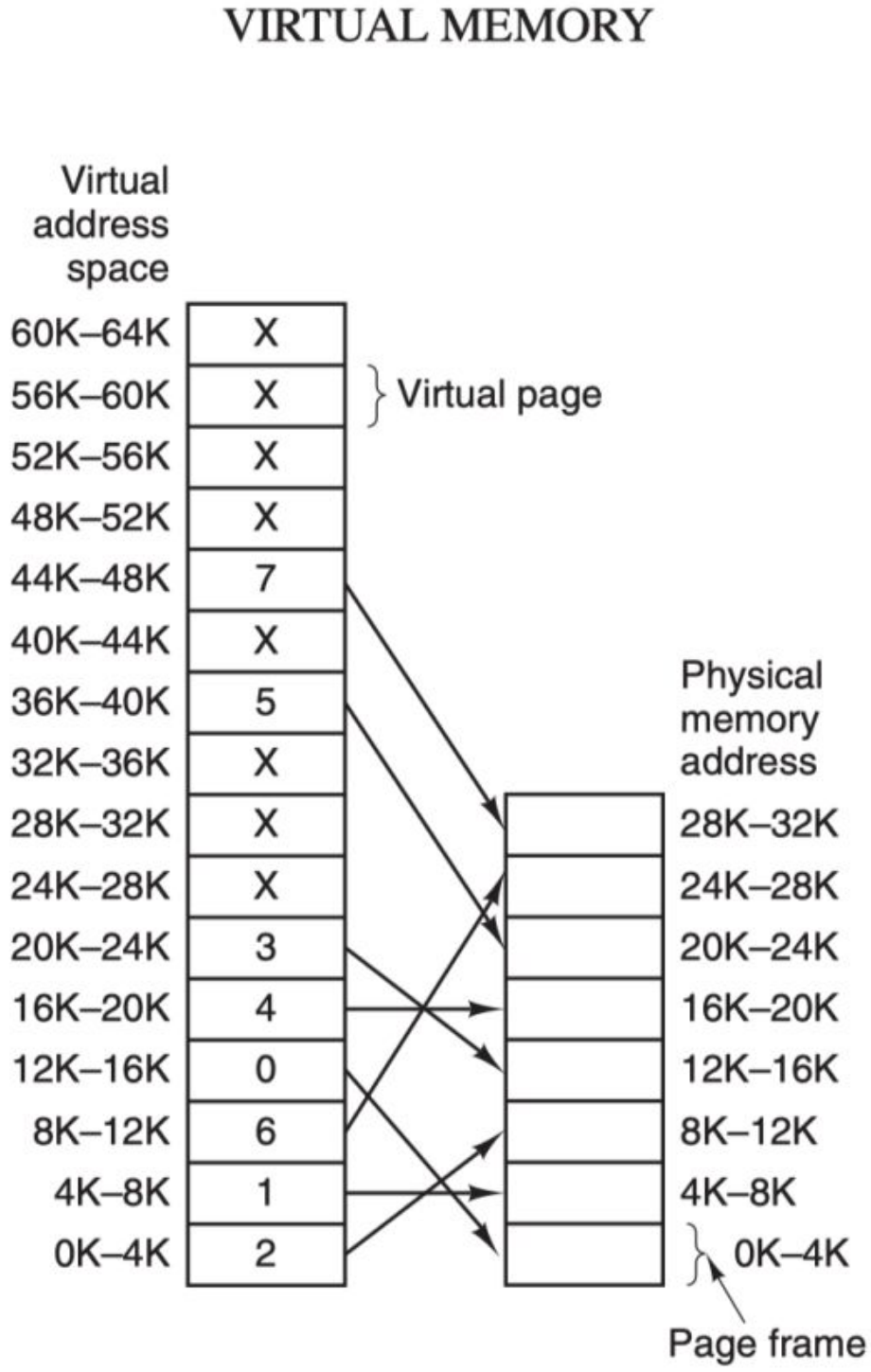
然后就是 交换（swapping） 技术，这种技术简单来说就是动态地把程序在内存和磁盘之间进行交换保存，要运行一个进程的时候就把程序的代码段和数据段调入内存，然后再把程序封存，存入磁盘，如此反复。为什么要这么麻烦？因为前面那两种重定位技术的前提条件是计算机内存足够大，能够把所有要运行的进程地址空间都加载进主存，才能够并发运行这些进程，但是现实往往不是如此，内存的大小总是有限的，所有就需要另一类方法来处理内存超载的情况，第一种便是简单的交换技术：



先把进程 A 换入内存，然后启动进程 B 和 C，也换入内存，接着 A 被从内存交换到磁盘，然后又有新的进程 D 调入内存，用了 A 退出之后空出来的内存空间，最后 A 又被重新换入内存，由于内存布局已经发生了变化，所以 A 在换入内存之时会通过软件或者在运行期间通过硬件（基址寄存器和界限寄存器）对其内存地址进行重定位，多数情况下都是通过硬件。

另一种处理内存超载的技术就是 虚拟内存 技术了，它比 交换（swapping） 技术更复杂而又更高效，是目前最新应用最广泛的存储器抽象技术：

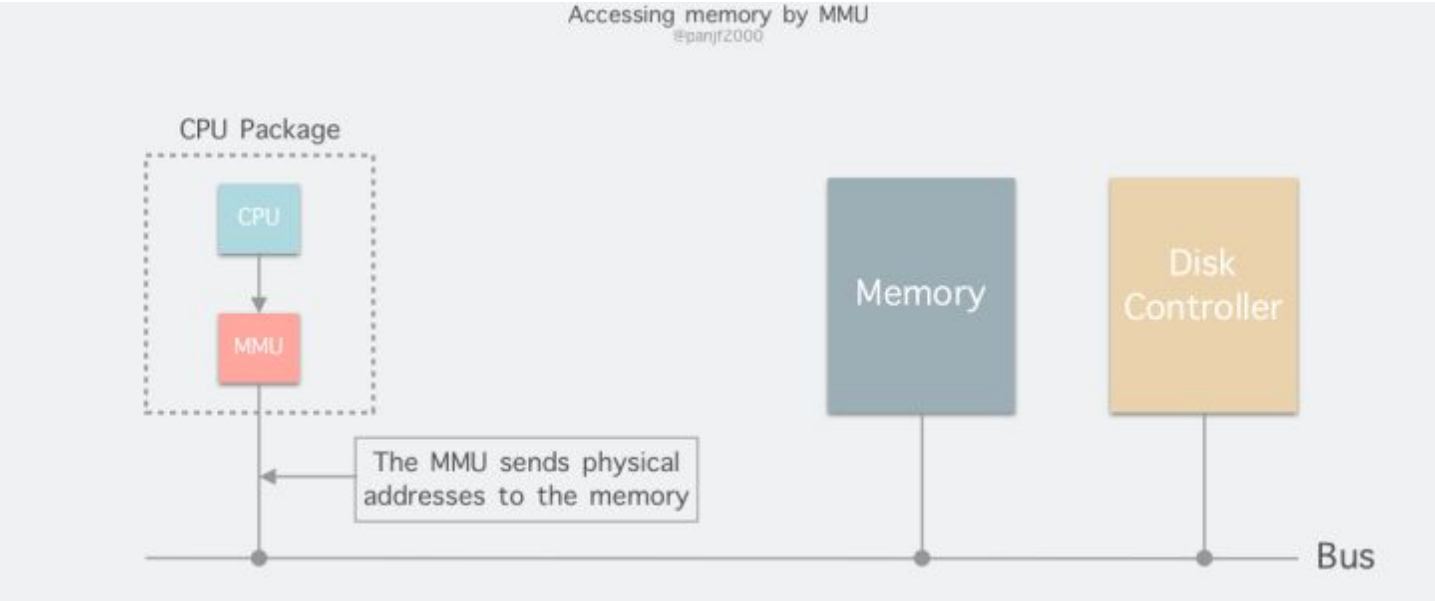
虚拟内存的核心原理是：为每个程序设置一段"连续"的虚拟地址空间，把这个地址空间分割成多个具有连续地址范围的页 (page)，并把这些页和物理内存做映射，在程序运行期间动态映射到物理内存。当程序引用到一段在物理内存的地址空间时，由硬件立刻执行必要的映射；而当程序引用到一段不在物理内存中的地址空间时，由操作系统负责将缺失的部分装入物理内存并重新执行失败的指令：



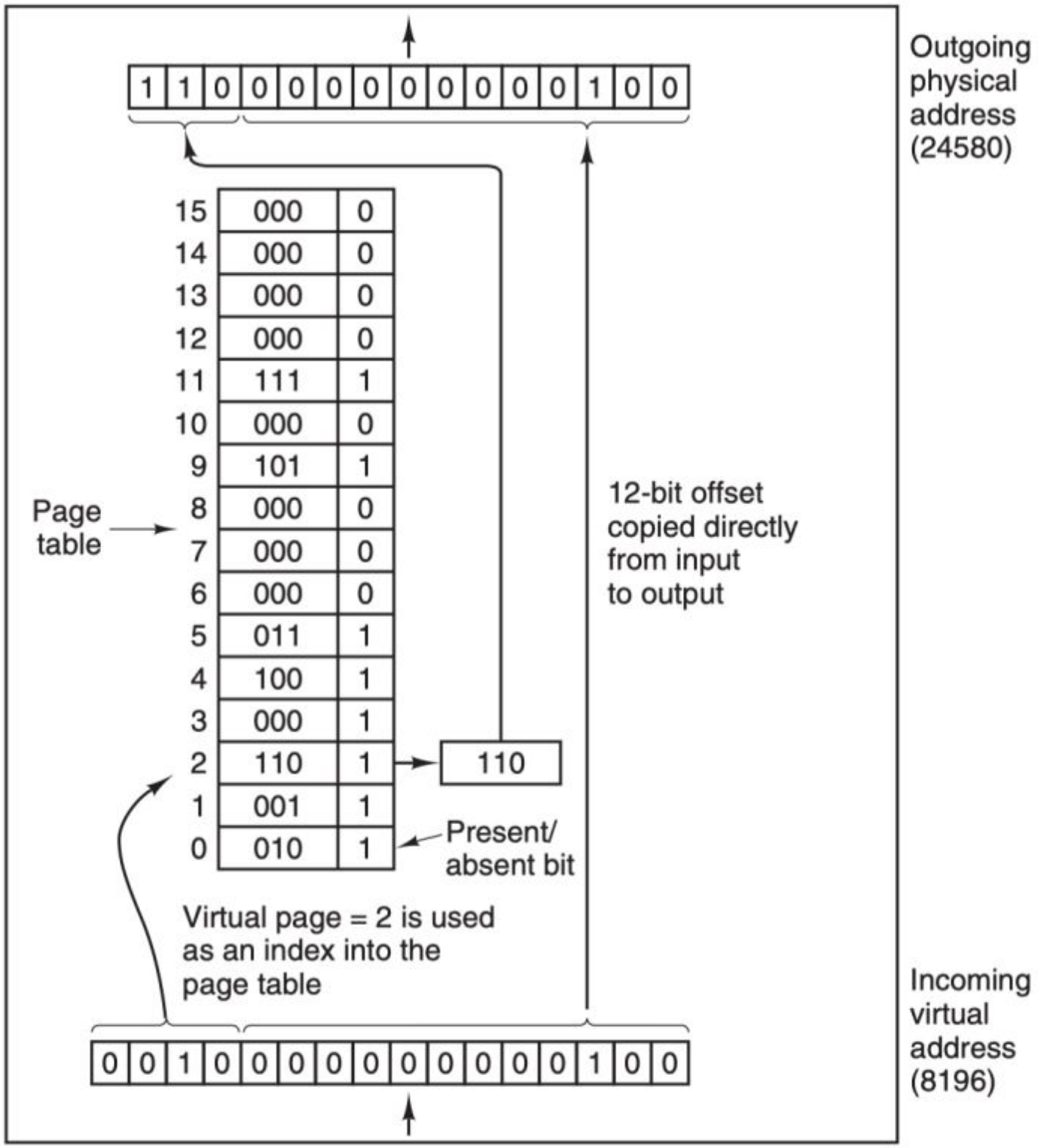
虚拟地址空间按照固定大小划分成被称为页（page）的若干单元，物理内存中对应的则是页框（page frame）。这两者一般来说是一样的大小，如上图中的是 4KB，不过实际上计算机系统中一般是 512 字节到 1 GB，这就是虚拟内存的分页技术。因为是虚拟内存空间，每个进程分配的大小是 4GB (32 位架构)，而实际上当然不可能给所有在运行中的进程都分配 4GB 的物理内存，所以虚拟内存技术还需要利用到前面介绍的 `交换（swapping）` 技术，在进程运行期间只分配映射当前使用到的内存，暂时不使用的数据则写回磁盘作为副本保存，需要用的时候再读入内存，动态地在磁盘和内存之间交换数据。

其实虚拟内存技术从某种角度来看的话，很像是糅合了基址寄存器和界限寄存器之后的新技术。它使得整个进程的地址空间可以通过较小的单元映射到物理内存，而不需要为程序的代码和数据地址进行重定位。

进程在运行期间产生的内存地址都是虚拟地址，如果计算机没有引入虚拟内存这种存储器抽象技术的话，则 CPU 会把这些地址直接发送到内存地址总线上，直接访问和虚拟地址相同值的物理地址；如果使用虚拟内存技术的话，CPU 则是把这些虚拟地址通过地址总线送到内存管理单元（Memory Management Unit，MMU），MMU 将虚拟地址映射为物理地址之后再通过内存总线去访问物理内存：



虚拟地址（比如 16 位地址 8196=0010 000000000100）分为两部分：虚拟页号（高位部分）和偏移量（低位部分），虚拟地址转换成物理地址是通过页表（page table）来实现的，页表由页表项构成，页表项中保存了页框号、修改位、访问位、保护位和 "在/不在" 位等信息，从数学角度来说页表就是一个函数，入参是虚拟页号，输出是物理页框号，得到物理页框号之后复制到寄存器的高三位中，最后直接把 12 位的偏移量复制到寄存器的末 12 位构成 15 位的物理地址，即可以把该寄存器的存储的物理内存地址发送到内存总线：



在 MMU 进行地址转换时，如果页表项的 "在/不在" 位是 0，则表示该页面并没有映射到真实的物理页框，则会引发一个**缺页中断**，CPU 陷入操作系统内核，接着操作系统就会通过页面置换算法选择一个页面将其换出 (swap)，以便为即将调入的新页面腾出位置，如果要换出的页面的页表项里的修改位已经被设置过，也就是被更新过，则这是一个脏页 (dirty page)，需要写回磁盘更新改页面在磁盘上的副本，如果该页面是"干净"的，也就是没有被修改过，则直接用调入的新页面覆盖掉被换出的旧页面即可。

最后，还需要了解的一个概念是转换检测缓冲器（Translation Lookaside Buffer，TLB），也叫快表，是用来加速虚拟地址映射的，因为虚拟内存的分页机制，页表一般是保存内存中的一块固定的存储区，导致进程通过 MMU 访问内存比直接访问内存多了一次内存访问，性能至少下降一半，因此需要引入加速机制，即 TLB 快表，TLB 可以简单地理解成页表的高速缓存，保存了最高频被访问的页表项，由于一般是硬件实现的，因此速度极快，MMU 收到虚拟地址时一般会先通过硬件 TLB 查询对应的页表号，若命中且该页表项的访问操作合法，则直接从 TLB 取出对应的物

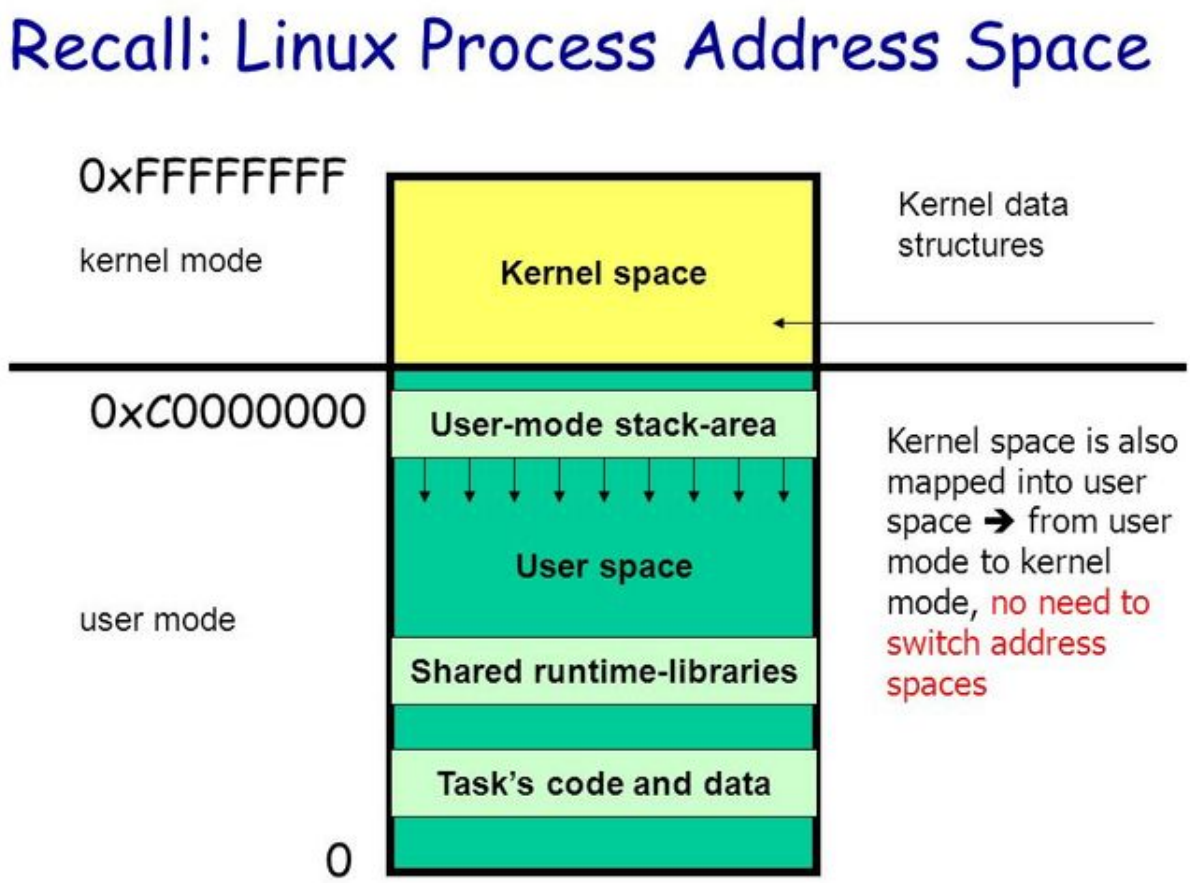
理页框号返回，若不命中则穿透到内存页表里查询，并且会用这个从内存页表里查询到最新页表项替换到现有 TLB 里的其中一个，以备下次缓存命中。

至此，我们介绍完了包含虚拟内存在内的多项计算机存储器抽象技术，虚拟内存的其他内容比如针对大内存的多级页表、倒排页表，以及处理缺页中断的页面置换算法等等，以后有机会再单独写一篇文章介绍，或者各位读者也可以先行去查阅相关资料了解，这里就不再深入了。

用户态和内核态

一般来说，我们在编写程序操作 Linux I/O 之时十有八九是在用户空间和内核空间之间传输数据，因此有必要先了解一下 Linux 的用户态和内核态的概念。

首先是用户态和内核态：



从宏观上来看，Linux 操作系统的体系架构分为用户态和内核态（或者用户空间和内核）。内核从本质上看是一种软件 —— 控制计算机的硬件资源，并提供上层应用程序 (进程) 运行的环境。用户态即上层应用程序 (进程) 的运行空间，应用程序 (进程) 的执行必须依托于内核提供的资源，这其中包括但不限于 CPU 资源、存储资源、I/O 资源等等。

现代操作系统都是采用虚拟存储器，那么对 32 位操作系统而言，它的寻址空间（虚拟存储空间）为 2^32 B = 4G。操作系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底层硬件设备的所有权限。为了保证用户进程不能直接操作内核（kernel），保证内核的安全，操心系统将虚拟空间划分为两部分，一部分为内核空间，一部分为用户空间。针对 Linux 操作系统而言，将最高的 1G 字节（从虚拟地址 0xC0000000 到 0xFFFFFFFF），供内核使用，称为内核空间，而将较低的 3G 字节（从虚拟地址 0x00000000 到 0xBFFFFFFF），供各个进程使用，称为用户空间。

因为操作系统的资源是有限的，如果访问资源的操作过多，必然会消耗过多的系统资源，而且如果不对这些操作加以区分，很可能造成资源访问的冲突。所以，为了减少有限资源的访问和使用冲突，Unix/Linux 的设计哲学之一就是：对不同的操作赋予不同的执行等级，就是所谓特权的概念。简单说就是有多大能力做多大的事，与系统相关的一些特别关键的操作必须由最高特权的程序来完成。Intel 的 x86 架构的 CPU 提供了 0 到 3 四个特权级，数字越小，特权越高，Linux 操作系统中主要采用了 0 和 3 两个特权级，分别对应的就是内核态和用户态。运行于用户态的进程可以执行的操作和访问的资源都会受到极大的限制，而运行在内核态的进程则可以执行任何操作并且在资源的使用上没有限制。很多程序开始时运行于用户态，但在执行的过程中，一些操作需要在内核权限下才能执行，这就涉及到一个从用户态切换到内核态的过程。比如 C 函数库中的内存

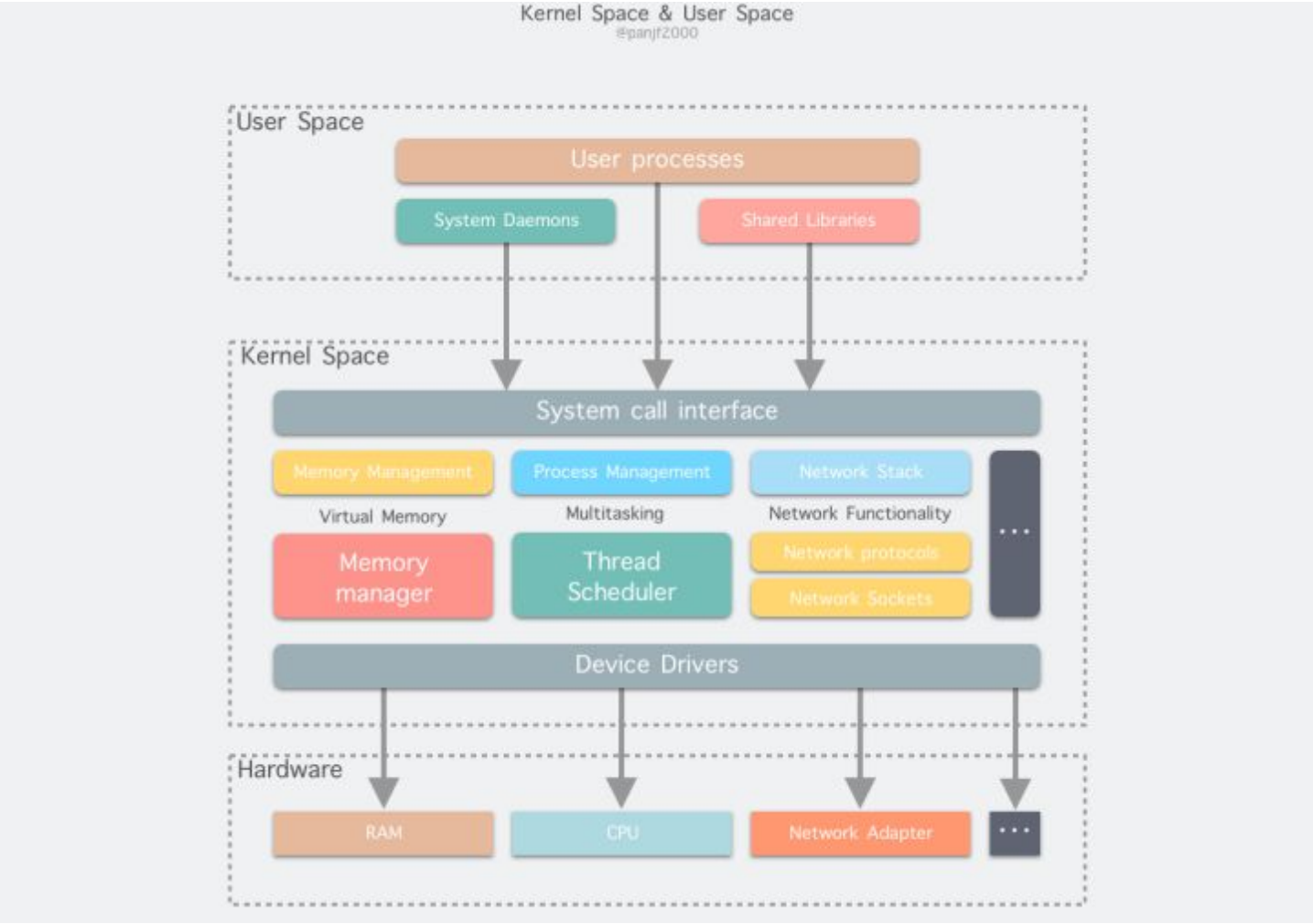
分配函数 malloc()，它具体是使用 sbrk() 系统调用来分配内存，当 malloc 调用 sbrk() 的时候就涉及一次从用户态到内核态的切换，类似的函数还有 printf()，调用的是 wirte() 系统调用来输出字符串，等等。

用户进程在系统中运行时，大部分时间是处在用户态空间里的，在其需要操作系统帮助完成一些用户态没有特权和能力完成的操作时就需要切换到内核态。那么用户进程如何切换到内核态去使用那些内核资源呢？答案是：1) 系统调用（trap），2) 异常（exception）和 3) 中断（interrupt）。

- **系统调用**：用户进程主动发起的操作。用户态进程发起系统调用主动要求切换到内核态，陷入内核之后，由操作系统来操作系统资源，完成之后再返回到进程。
- **异常**：被动的操作，且用户进程无法预测其发生的时机。当用户进程在运行期间发生了异常（比如某条指令出了问题），这时会触发由当前运行进程切换到处理此异常的内核相关进程中，也即是切换到了内核态。异常包括程序运算引起的各种错误如除 0、缓冲区溢出、缺页等。
- **中断**：当外围设备完成用户请求的操作后，会向 CPU 发出相应的中断信号，这时 CPU 会暂停执行下一条即将要执行的指令而转到与中断信号对应的处理程序去执行，如果前面执行的指令是用户态下的程序，那么转换的过程自然就会是从用户态到内核态的切换。中断包括 I/O 中断、外部信号中断、各种定时器引起的时钟中断等。中断和异常类似，都是通过中断向量表来找到相应的处理程序进行处理。区别在于，中断来自处理器外部，不是由任何一条专门的指令造成，而异常是执行当前指令的结果。

通过上面的分析，我们可以得出 Linux 的内部层级可分为三大部分：

- 用户空间；
- 内核空间；
- 硬件。



Linux I/O

I/O 缓冲区

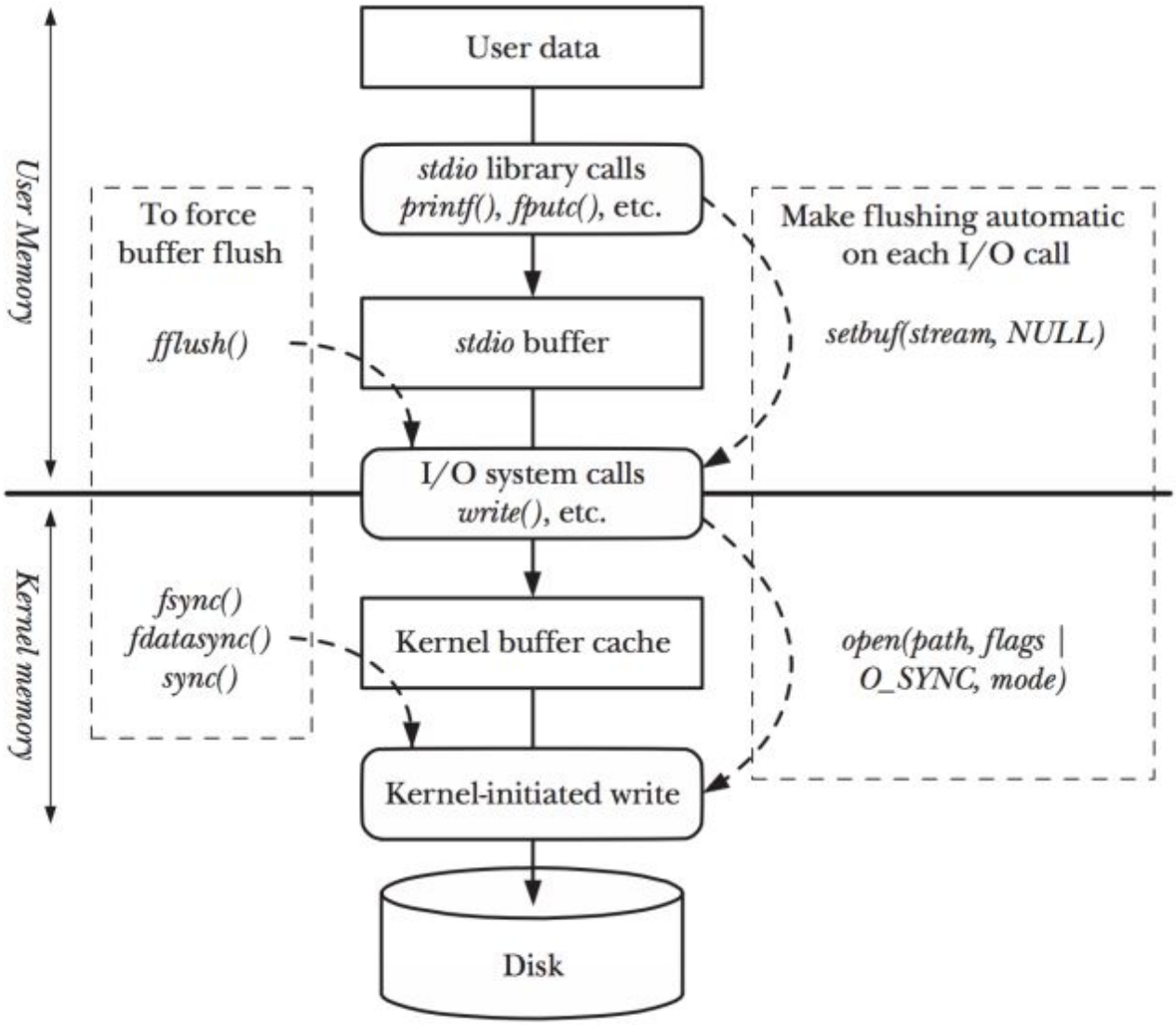


Figure 13-1: Summary of I/O buffering

在 Linux 中，当程序调用各类文件操作函数后，用户数据（User Data）到达磁盘（Disk）的流程如上图所示。

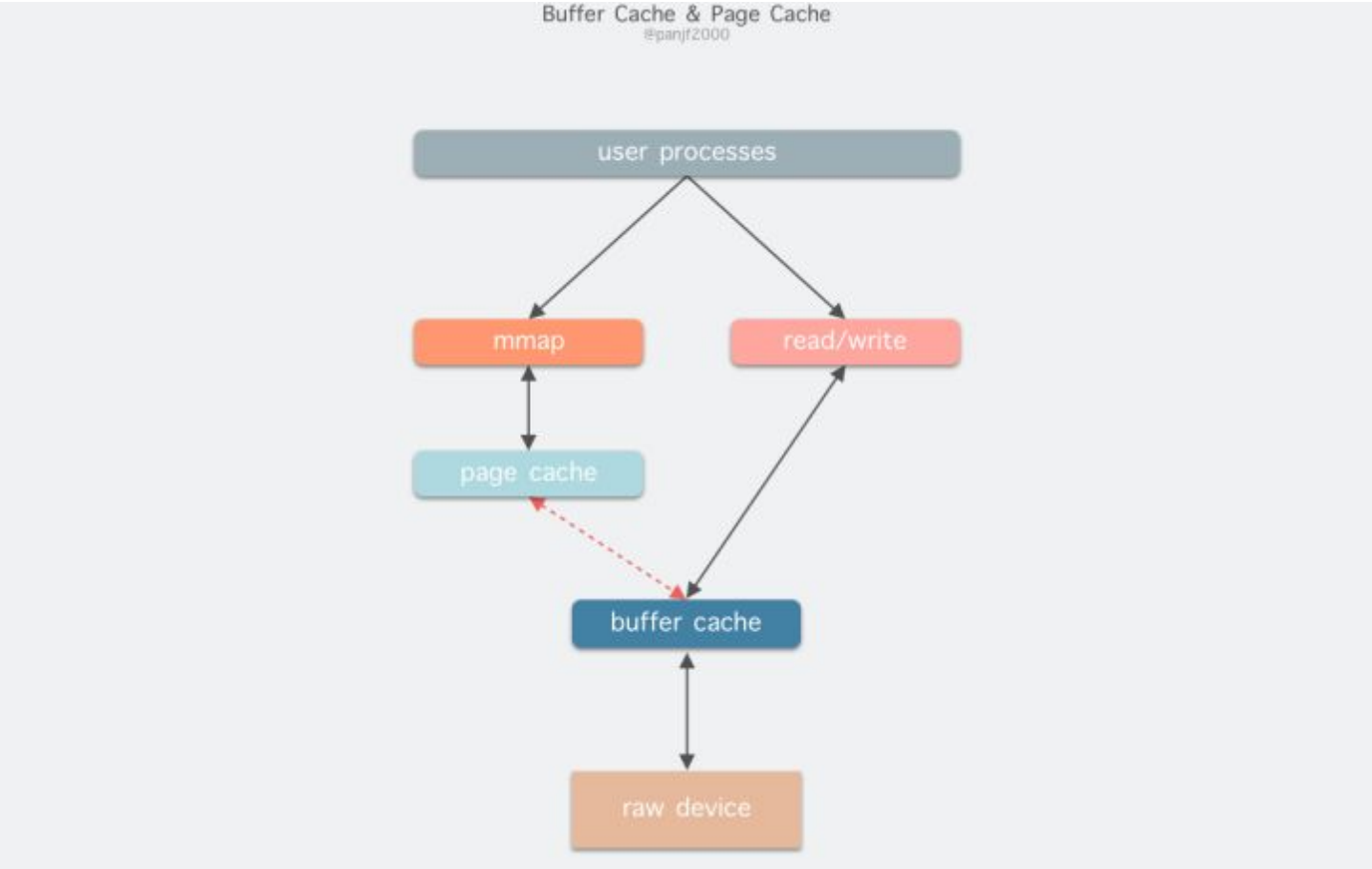
图中描述了 Linux 中文件操作函数的层级关系和内存缓存层的存在位置，中间的黑色实线是用户态和内核态的分界线。

`read(2)/write(2)` 是 Linux 系统中最基本的 I/O 读写系统调用，我们开发操作 I/O 的程序时必定会接触到它们，而在这两个系统调用和真实的磁盘读写之间存在一层称为 `Kernel buffer cache` 的缓冲区缓存。在 Linux 中 I/O 缓存其实可以细分为两个：`Page Cache` 和 `Buffer Cache`，这两个其实是一体两面，共同组成了 Linux 的内核缓冲区（Kernel Buffer Cache）：

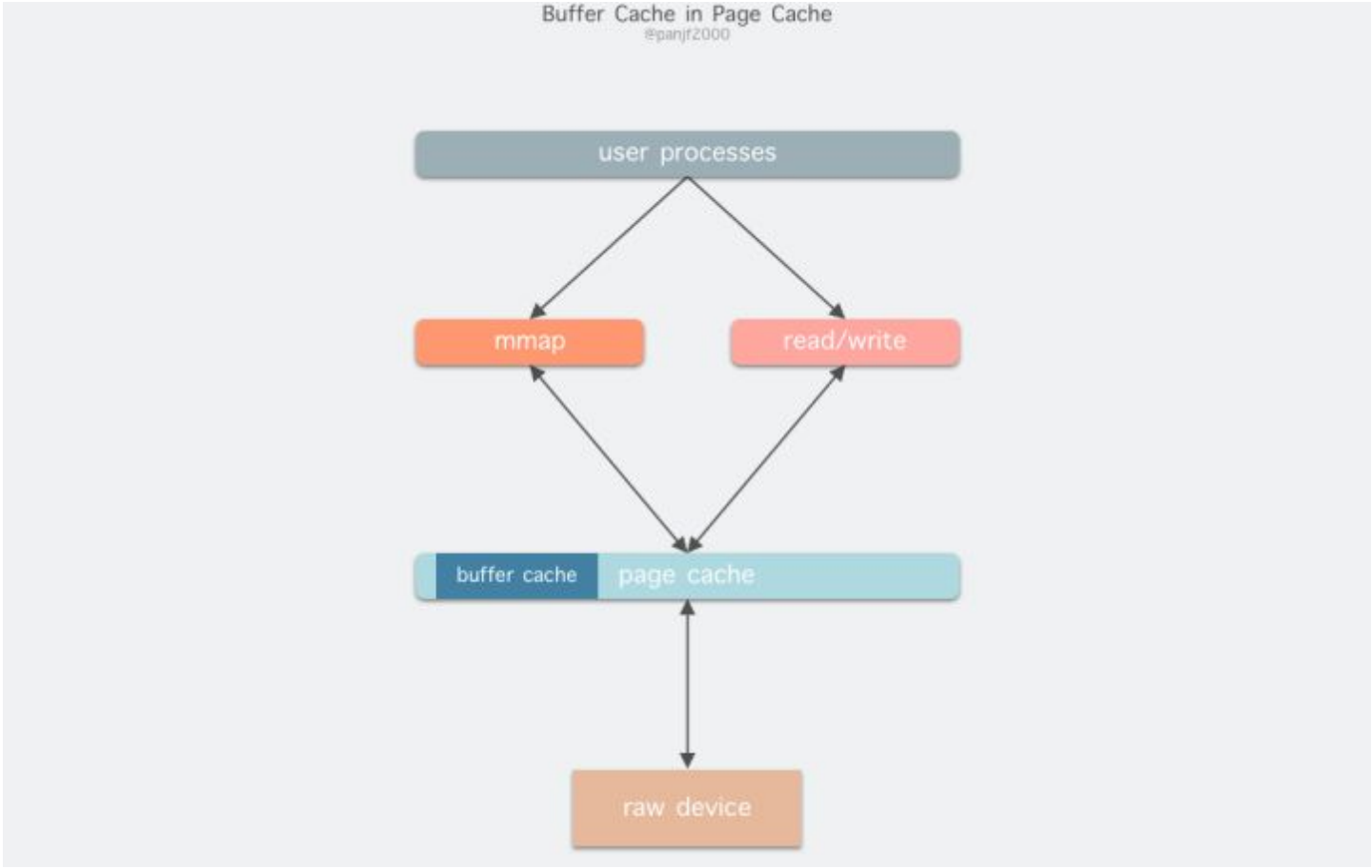
- **读磁盘**：内核会先检查 `Page Cache` 里是不是已经缓存了这个数据，若是，直接从这个内存缓冲区里读取返回，若否，则穿透到磁盘去读取，然后再缓存在 `Page Cache` 里，以备下次缓存命中；
- **写磁盘**：内核直接把数据写入 `Page Cache`，并把对应的页标记为 dirty，添加到 dirty list 里，然后就直接返回，内核会定期把 dirty list 的页缓存 flush 到磁盘，保证页缓存和磁盘的最终一致性。

`Page Cache` 会通过页面置换算法如 LRU 定期淘汰旧的页面，加载新的页面。可以看出，所谓 I/O 缓冲区缓存就是在内核和磁盘、网卡等外设之间的一层缓冲区，用来提升读写性能的。

在 Linux 还不支持虚拟内存技术之前，还没有页的概念，因此 `Buffer Cache` 是基于操作系统读写磁盘的最小单位 -- 块（block）来进行的，所有的磁盘块操作都是通过 `Buffer Cache` 来加速，Linux 引入虚拟内存的机制来管理内存后，页成为虚拟内存管理的最小单位，因此也引入了 `Page Cache` 来缓存 Linux 文件内容，主要用来作为文件系统上的文件数据的缓存，提升读写性能，常见的是针对文件的 `read()/write()` 操作，另外也包括了通过 `mmap()` 映射之后的块设备，也就是说，事实上 Page Cache 负责了大部分的块设备文件的缓存工作。而 `Buffer Cache` 用来在系统对块设备进行读写的时候，对块进行数据缓存的系统来使用，实际上负责所有对磁盘的 I/O 访问：



因为 `Buffer Cache` 是对粒度更细的设备块的缓存，而 `Page Cache` 是基于虚拟内存的页单元缓存，因此还是会基于 `Buffer Cache`，也就是说如果是缓存文件内容数据就会在内存里缓存两份相同的数据，这就会导致同一份文件保存了两份，冗余且低效。另外一个问题是，调用 `write` 后，有效数据是在 `Buffer Cache` 中，而非 `Page Cache` 中。这就导致 `mmap` 访问的文件数据可能存在不一致问题。为了规避这个问题，所有基于磁盘文件系统的 `write`，都需要调用 `update_vm_cache()` 函数，该操作会把调用 `write` 之后的 `Buffer Cache` 更新到 `Page Cache` 去。由于有这些设计上的弊端，因此在 Linux 2.4 版本之后，kernel 就将两者进行了统一，`Buffer Cache` 不再以独立的形式存在，而是以融合的方式存在于 `Page Cache` 中：



融合之后就可以统一操作 `Page Cache` 和 `Buffer Cache`：处理文件 I/O 缓存交给 `Page Cache`，而当底层 RAW device 刷新数据时以 `Buffer Cache` 的块单位来实际处理。

I/O 模式

在 Linux 或者其他 Unix-like 操作系统里，I/O 模式一般有三种：

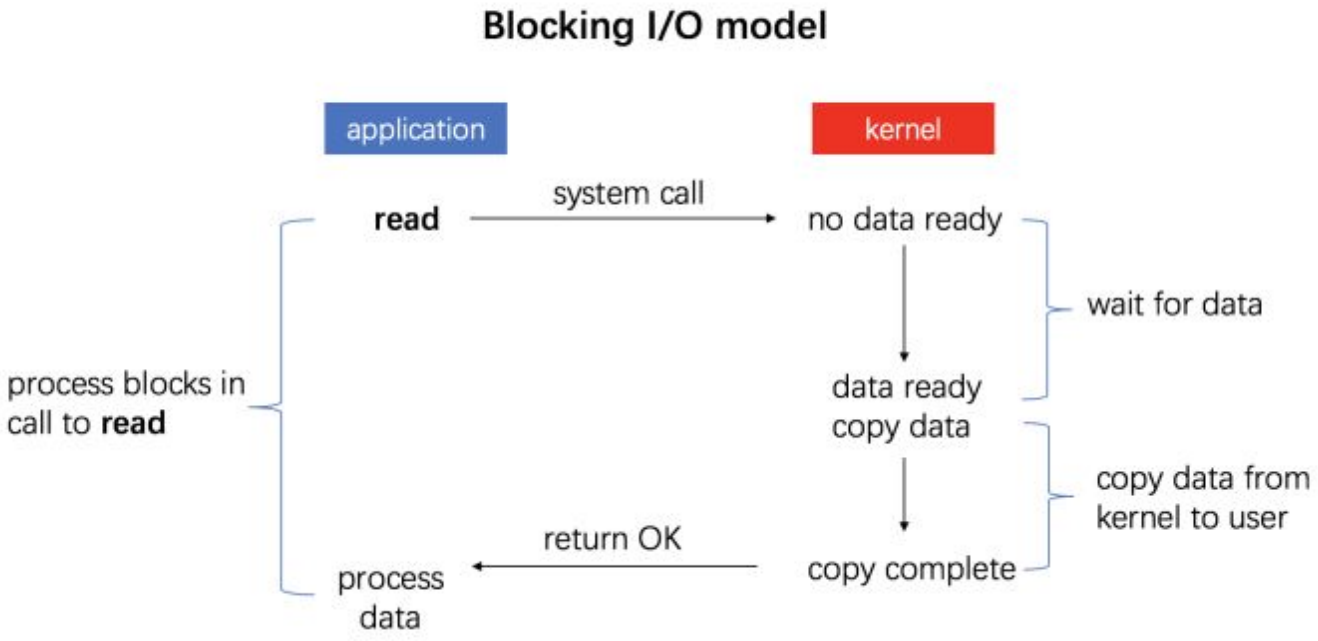
1. 程序控制 I/O
2. 中断驱动 I/O
3. DMA I/O

下面我分别详细地讲解一下这三种 I/O 模式。

程序控制 I/O

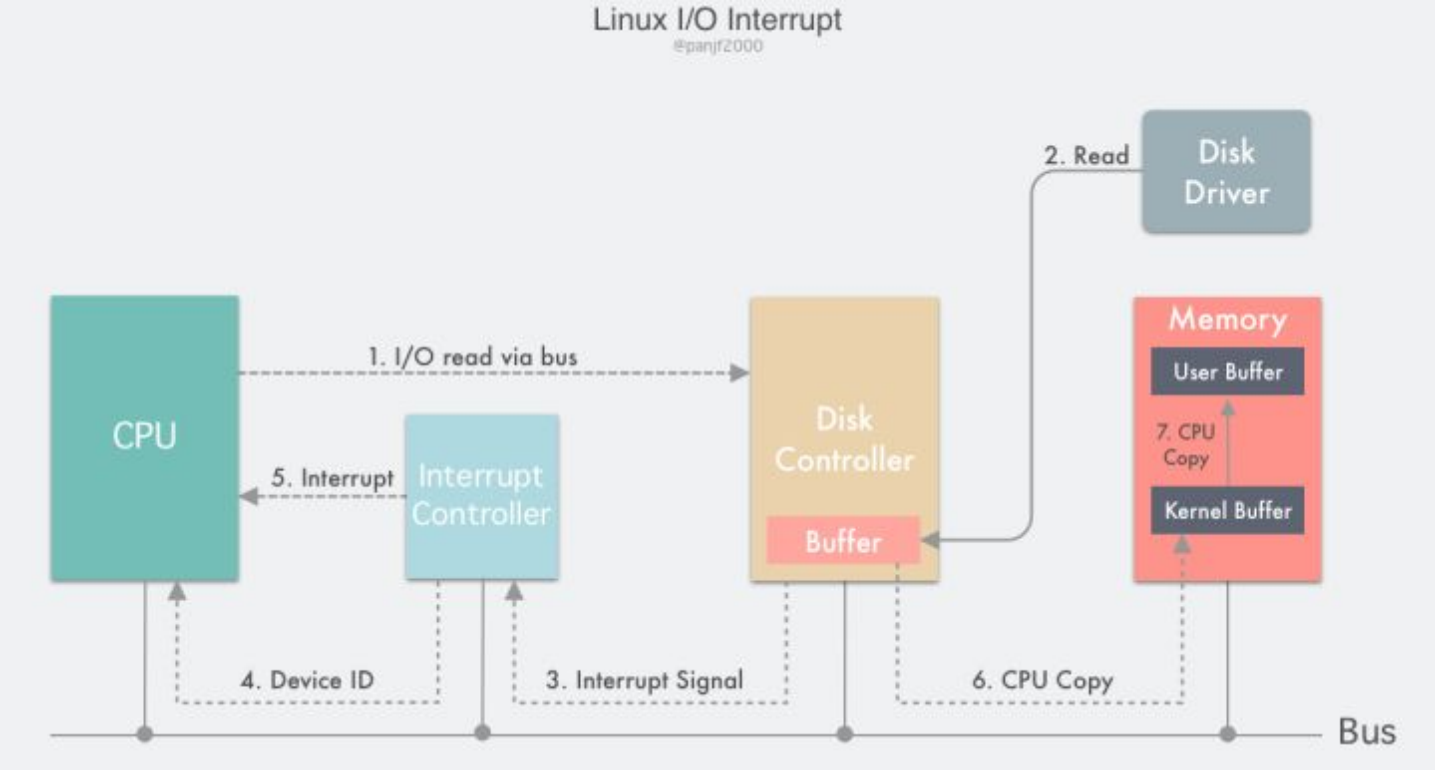
这是最简单的一种 I/O 模式，也叫忙等待或者轮询：用户通过发起一个系统调用，陷入内核态，内核将系统调用翻译成一个对应设备驱动程序的过程调用，接着设备驱动程序会启动 I/O 不断循环去检查该设备，看看是否已经就绪，一般通过返回码来表示，I/O 结束之后，设备驱动程序会把数据送到指定的地方并返回，切回用户态。

比如发起系统调用 `read()`：



中断驱动 I/O

第二种 I/O 模式是利用中断来实现的：



流程如下：

1. 用户进程发起一个 `read()` 系统调用读取磁盘文件，陷入内核态并由其所在的 CPU 通过设备驱动程序向设备寄存器写入一个通知信号，告知设备控制器 (我们这里是磁盘控制器)要读取数据；
2. 磁盘控制器启动磁盘读取的过程，把数据从磁盘拷贝到磁盘控制器缓冲区里；
3. 完成拷贝之后磁盘控制器会通过总线发送一个中断信号到中断控制器，如果此时中断控制器手头还有正在处理的中断或者有一个和该中断信号同时到达的更高优先级的中断，则这个中断信号将被忽略，而磁盘控制器会在后面持续发送中断信号直至中断控制器受理；
4. 中断控制器收到磁盘控制器的中断信号之后会通过地址总线存入一个磁盘设备的编号，表示这次中断需要关注的设备是磁盘；
5. 中断控制器向 CPU 置起一个磁盘中断信号；

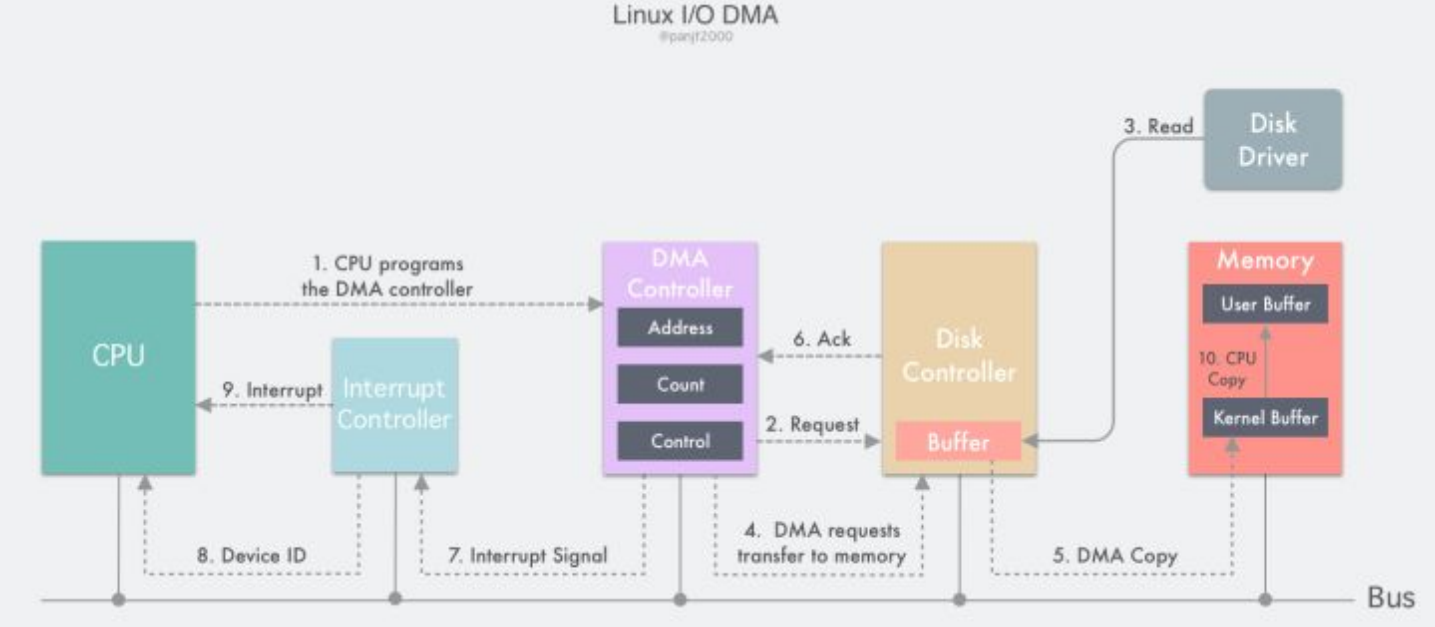
6. CPU 收到中断信号之后停止当前的工作，把当前的 PC/PSW 等寄存器压入堆栈保存现场，然后从地址总线取出设备编号，通过编号找到中断向量所包含的中断服务的入口地址，压入 PC 寄存器，开始运行磁盘中断服务，把数据从磁盘控制器的缓冲区拷贝到主存里的内核缓冲区；
7. 最后 CPU 再把数据从内核缓冲区拷贝到用户缓冲区，完成读取操作，`read()` 返回，切换回用户态。

DMA I/O

并发系统的性能高低究其根本，是取决于如何对 CPU 资源的高效调度和使用，而回头看前面的中断驱动 I/O 模式的流程，可以发现第 6、7 步的数据拷贝工作都是由 CPU 亲自完成的，也就是在这两次数据拷贝阶段中 CPU 是完全被占用而不能处理其他工作的，那么这里明显是有优化空间的；第 7 步的数据拷贝是从内核缓冲区到用户缓冲区，都是在主存里，所以这一步只能由 CPU 亲自完成，但是第 6 步的数据拷贝，是从磁盘控制器的缓冲区到主存，是两个设备之间的数据传输，这一步并非一定要 CPU 来完成，可以借助 DMA 来完成，减轻 CPU 的负担。

DMA 全称是 Direct Memory Access，也即直接存储器存取，是一种用来提供在外设和存储器之间或者存储器和存储器之间的高速数据传输。整个过程无须 CPU 参与，数据直接通过 DMA 控制器进行快速地移动拷贝，节省 CPU 的资源去做其他工作。

目前，大部分的计算机都配备了 DMA 控制器，而 DMA 技术也支持大部分的外设和存储器。借助于 DMA 机制，计算机的 I/O 过程就能更加高效：



DMA 控制器内部包含若干个可以被 CPU 读写的寄存器：一个主存地址寄存器 MAR（存放要交换数据的主存地址）、一个外设地址寄存器 ADR（存放 I/O 设备的设备码，或者是设备信息存储区的寻址信息）、一个字节数寄存器 WC（对传送数据的总字数进行统计）、和一个或多个控制寄存器。

1. 用户进程发起一个 `read()` 系统调用读取磁盘文件，陷入内核态并由其所在的 CPU 通过设置 DMA 控制器的寄存器对它进行编程：把内核缓冲区和磁盘文件的地址分别写入 MAR 和 ADR 寄存器，然后把期望读取的字节数写入 WC 寄存器，启动 DMA 控制器；
2. DMA 控制器根据 ADR 寄存器里的信息知道这次 I/O 需要读取的外设是磁盘的某个地址，便向磁盘控制器发出一个命令，通知它从磁盘读取数据到其内部的缓冲区里；
3. 磁盘控制器启动磁盘读取的过程，把数据从磁盘拷贝到磁盘控制器缓冲区里，并对缓冲区内数据的校验和进行检验，如果数据是有效的，那么 DMA 就可以开始了；
4. DMA 控制器通过总线向磁盘控制器发出一个读请求信号从而发起 DMA 传输，这个信号和前面的中断驱动 I/O 小节里 CPU 发给磁盘控制器的读请求是一样的，它并不知道或者并不关心这个读请求是来自 CPU 还是 DMA 控制器；
5. 紧接着 DMA 控制器将引导磁盘控制器将数据传输到 MAR 寄存器里的地址，也就是内核缓冲区；
6. 数据传输完成之后，返回一个 ack 给 DMA 控制器，WC 寄存器里的值会减去相应的数据长度，如果 WC 还不为 0，则重复第 4 步到第 6 步，一直到 WC 里的字节数等于 0；
7. 收到 ack 信号的 DMA 控制器会通过总线发送一个中断信号到中断控制器，如果此时中断控制器手头还有正在处理的中断或者有一个和该中断信号同时到达的更高优先级的中断，则这个中断信号将被忽略，而 DMA 控制器会在后面持续发送中断信号直至中断控制器受理；
8. 中断控制器收到磁盘控制器的中断信号之后会通过地址总线存入一个主存设备的编号，表示这次中断需要关注的设备是主存；
9. 中断控制器向 CPU 置起一个 DMA 中断的信号；
10. CPU 收到中断信号之后停止当前的工作，把当前的 PC/PSW 等寄存器压入堆栈保存现场，然后从地址总线取出设备编号，通过编号找到中断向量所包含的中断服务的入口地址，压入 PC 寄存器，开始运行 DMA 中断服务，把数据从内核缓冲区拷贝到用户缓冲区，完成读取操作，`read()` 返回，切换回用户态。

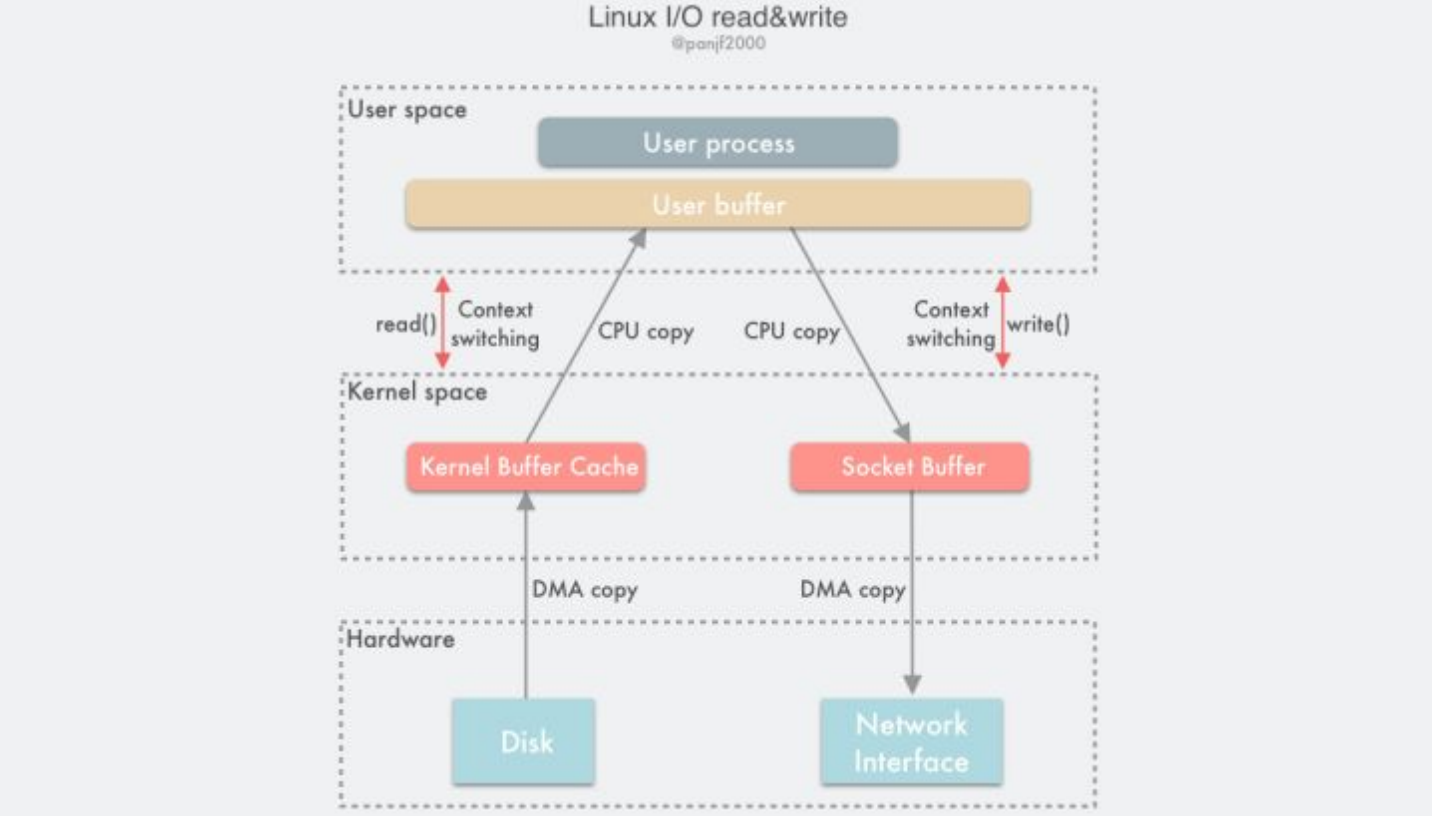
传统 I/O 读写模式

Linux 中传统的 I/O 读写是通过 `read()/write()` 系统调用完成的， `read()` 把数据从存储器 (磁盘、网卡等) 读取到用户缓冲区， `write()` 则是把数据从用户缓冲区写出到存储器：

```
#include <unistd.h>

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

一次完整的读磁盘文件然后写出到网卡的底层传输过程如下：



可以清楚看到这里一共触发了 4 次用户态和内核态的上下文切换，分别是 `read()/write()` 调用和返回时的切换，2 次 DMA 拷贝，2 次 CPU 拷贝，加起来一共 4 次拷贝操作。

通过引入 DMA，我们已经把 Linux 的 I/O 过程中的 CPU 拷贝次数从 4 次减少到了 2 次，但是 CPU 拷贝依然是代价很大的操作，对系统性能的影响还是很大，特别是那些频繁 I/O 的场景，更是会因为 CPU 拷贝而损失掉很多性能，我们需要进一步优化，降低、甚至是完全避免 CPU 拷贝。

零拷贝 (Zero-copy)

Zero-copy 是什么?

Wikipedia 的解释如下：

> "**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 周期和**内存带宽**。

Zero-copy 能做什么?

- 减少甚至完全避免操作系统内核和用户应用程序地址空间这两者之间进行数据拷贝操作，从而减少用户态 -- 内核态上下文切换带来的系统开

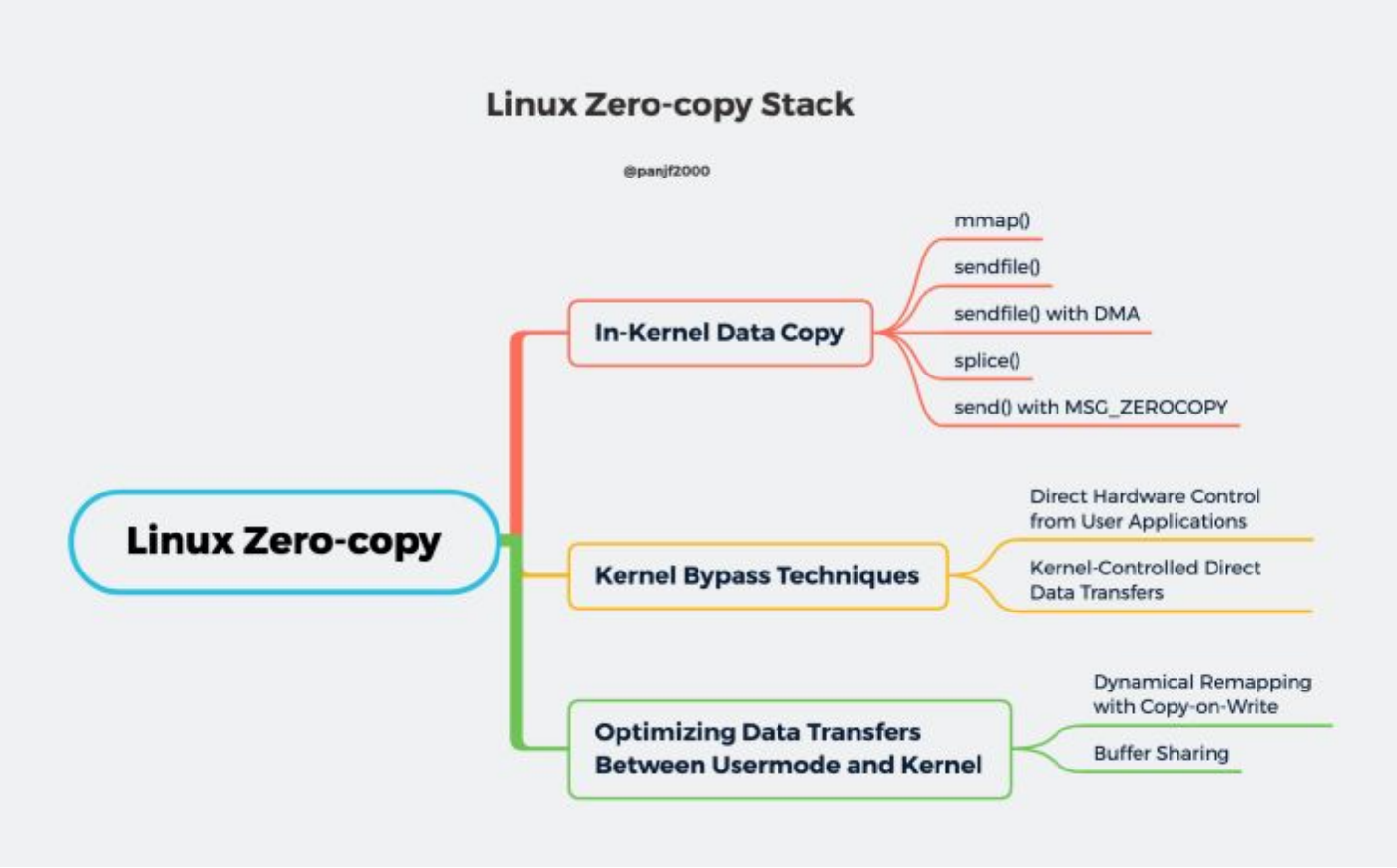
销。
- 减少甚至完全避免操作系统内核缓冲区之间进行数据拷贝操作。
- 帮助用户进程绕开操作系统内核空间直接访问硬件存储接口操作数据。
- 利用 DMA 而非 CPU 来完成硬件接口和内核缓冲区之间的数据拷贝，从而解放 CPU，使之能去执行其他的任务，提升系统性能。

Zero-copy 的实现方式有哪些?

从 zero-copy 这个概念被提出以来，相关的实现技术便犹如雨后春笋，层出不穷。但是截至目前为止，并没有任何一种 zero-copy 技术能满足所有的场景需求，还是计算机领域那句无比经典的名言："There is no silver bullet"!

而在 Linux 平台上，同样也有很多的 zero-copy 技术，新旧各不同，可能存在于不同的内核版本里，很多技术可能有了很大的改进或者被更新的实现方式所替代，这些不同的实现技术按照其核心思想可以归纳成大致的以下三类：

- **减少甚至避免用户空间和内核空间之间的数据拷贝**：在一些场景下，用户进程在数据传输过程中并不需要对数据进行访问和处理，那么数据在 Linux 的 `Page Cache` 和用户进程的缓冲区之间的传输就完全可以避免，让数据拷贝完全在内核里进行，甚至可以通过更巧妙的方式避免在内核里的数据拷贝。这一类实现一般是通过增加新的系统调用来完成的，比如 Linux 中的 mmap()，sendfile() 以及 splice() 等。
- **绕过内核的直接 I/O**：允许在用户态进程绕过内核直接和硬件进行数据传输，内核在传输过程中只负责一些管理和辅助的工作。这种方式其实和第一种有点类似，也是试图避免用户空间和内核空间之间的数据传输，只是第一种方式是把数据传输过程放在内核态完成，而这种方式则是直接绕过内核和硬件通信，效果类似但原理完全不同。
- **内核缓冲区和用户缓冲区之间的传输优化**：这种方式侧重于在用户进程的缓冲区和操作系统的页缓存之间的 CPU 拷贝的优化。这种方法延续了以往那种传统的通信方式，但更灵活。



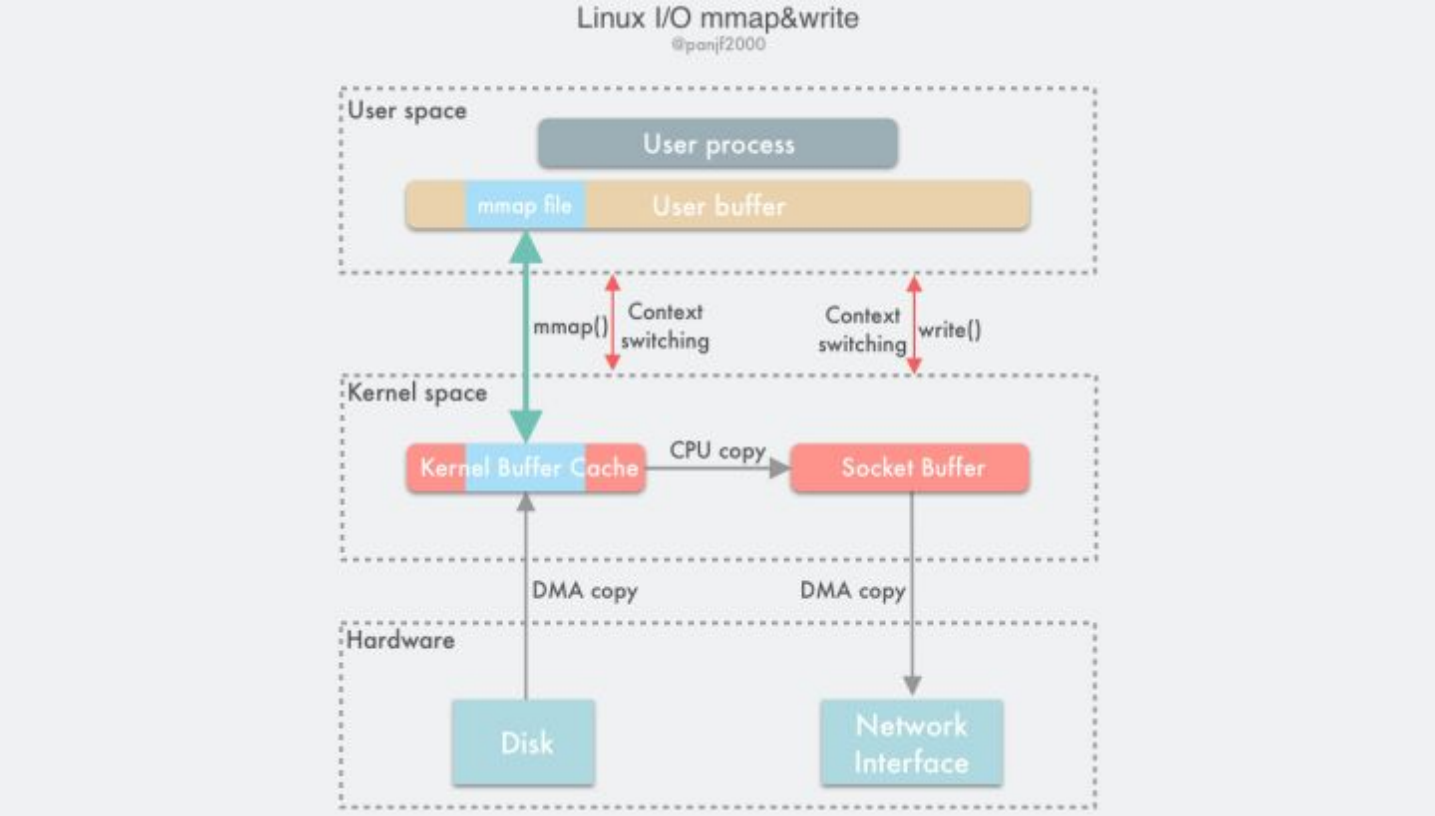
减少甚至避免用户空间和内核空间之间的数据拷贝

mmap()

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

一种简单的实现方案是在一次读写过程中用 Linux 的另一个系统调用 `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。



利用 `mmap()` 替换 `read()`，配合 `write()` 调用的整个流程如下：

1. 用户进程调用 `mmap()`，从用户态陷入内核态，将内核缓冲区映射到用户缓存区；
2. DMA 控制器将数据从硬盘拷贝到内核缓冲区；
3. `mmap()` 返回，上下文从内核态切换回用户态；
4. 用户进程调用 `write()`，尝试把文件数据写到内核里的套接字缓冲区，再次陷入内核态；
5. CPU 将内核缓冲区中的数据拷贝到的套接字缓冲区；
6. DMA 控制器将数据从套接字缓冲区拷贝到网卡完成数据传输；
7. `write()` 返回，上下文从内核态切换回用户态。

通过这种方式，有两个优点：一是节省内存空间，因为用户进程上的这一段内存是虚拟的，并不真正占据物理内存，只是映射到文件所在的内核缓冲区上，因此可以节省一半的内存占用；二是省去了一次 CPU 拷贝，对比传统的 Linux I/O 读写，数据不需要再经过用户进程进行转发了，而是直接在内核里就完成了拷贝。所以使用 `mmap()` 之后的拷贝次数是 2 次 DMA 拷贝，1 次 CPU 拷贝，加起来一共 3 次拷贝操作，比传统的 I/O 方式节省了一次 CPU 拷贝以及一半的内存，不过因为 `mmap()` 也是一个系统调用，因此用户态和内核态的切换还是 4 次。

`mmap()` 因为既节省 CPU 拷贝次数又节省内存，所以比较适合大文件传输的场景。虽然 `mmap()` 完全是符合 POSIX 标准的，但是它也不是完美的，因为它并不总是能达到理想的数据传输性能。首先是因为数据数据传输过程中依然需要一次 CPU 拷贝，其次是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。但是，因为内存映射通常针对的是相对较大的数据区域，所以对于相同大小的数据来说，内存映射所带来的开销远远低于 CPU 拷贝所带来的开销。此外，使用 `mmap()` 还可能会遇到一些需要值得关注的特殊情况，例如，在 `mmap()` --> `write()` 这两个系统调用的整个传输过程中，如果有其他的进程突然截断了这个文件，那么这时用户进程就会因为访问非法地址而被一个从总线传来的 SIGBUS 中断信号杀死并且产生一个 core dump。有两种解决办法：

1. 设置一个信号处理器，专门用来处理 SIGBUS 信号，这个处理器直接返回，`write()` 就可以正常返回已写入的字节数而不会被 SIGBUS 中断，errno 错误码也会被设置成 success。然而这实际上是一个掩耳盗铃的解决方案，因为 BIGBUS 信号的带来的信息是系统发生了一些很严重的错误，而我们却选择忽略掉它，一般不建议采用这种方式。
2. 通过内核的文件租借锁（这是 Linux 的叫法，Windows 上称之为机会锁）来解决这个问题，这种方法相对来说更好一些。我们可以通过内核对文件描述符上读/写的租借锁，当另外一个进程尝试对当前用户进程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正在破坏你加在那个文件上的读/写租借锁，这时 `write()` 系统调用会被中断，并且当前用户进程会被 SIGBUS 信号杀死，返回值则是中断前写的字节数，errno 同样会被设置为 success。文件租借锁需要在对文件进行内存映射之前设置，最后在用户进程结束之前释放掉。

sendfile()

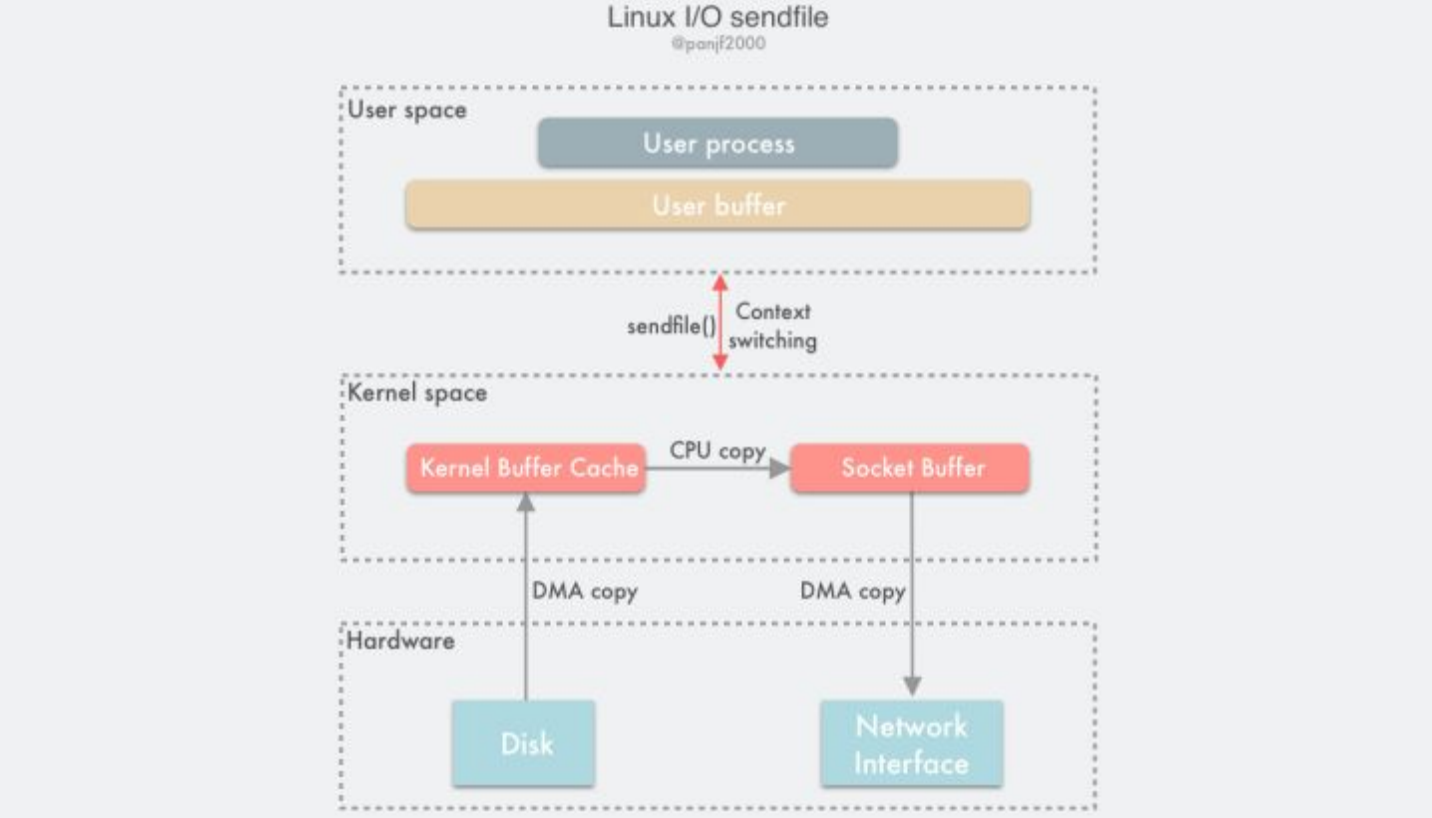
在 Linux 内核 2.1 版本中，引入了一个新的系统调用 `sendfile()`：

```
#include <sys/sendfile.h>

ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

从功能上来看，这个系统调用将 `mmap()` + `write()` 这两个系统调用合二为一，实现了一样效果的同时还简化了用户接口，其他的一些 Unix-like 的系统像 BSD、Solaris 和 AIX 等也有类似的实现，甚至 Windows 上也有一个功能类似的 API 函数 `TransmitFile`。

out_fd 和 in_fd 分别代表了写入和读出的文件描述符，in_fd 必须是一个指向文件的文件描述符，且要能支持类 `mmap()` 内存映射，不能是 Socket 类型，而 out_fd 在 Linux 内核 2.6.33 版本之前只能是一个指向 Socket 的文件描述符，从 2.6.33 之后则可以是任意类型的文件描述符。off_t 是一个代表了 in_fd 偏移量的指针，指示 `sendfile()` 该从 in_fd 的哪个位置开始读取，函数返回后，这个指针会被更新成 `sendfile()` 最后读取的字节位置处，表明此次调用共读取了多少文件数据，最后的 count 参数则是此次调用需要传输的字节总数。



使用 `sendfile()` 完成一次数据读写的流程如下：

1. 用户进程调用 `sendfile()` 从用户态陷入内核态；
2. DMA 控制器将数据从硬盘拷贝到内核缓冲区；
3. CPU 将内核缓冲区中的数据拷贝到套接字缓冲区；
4. DMA 控制器将数据从套接字缓冲区拷贝到网卡完成数据传输；
5. `sendfile()` 返回，上下文从内核态切换回用户态。

基于 `sendfile()`， 整个数据传输过程中共发生 2 次 DMA 拷贝和 1 次 CPU 拷贝，这个和 `mmap()` + `write()` 相同，但是因为 `sendfile()` 只是一次系统调用，因此比前者少了一次用户态和内核态的上下文切换开销。读到这里，聪明的读者应该会开始提问了：" `sendfile()` 会不会遇到和 `mmap()` + `write()` 相似的文件截断问题呢？"，很不幸，答案是肯定的。 `sendfile()` 一样会有文件截断的问题，但欣慰的是，`sendfile()` 不仅比 `mmap()` + `write()` 在接口使用上更加简洁，而且处理文件截断时也更加优雅：如果 `sendfile()` 过程中遭遇文件截断，则 `sendfile()` 系统调用会被中断杀死之前返回给用户进程其中断前所传输的字节数，errno 会被设置为 success，无需用户提前设置信号处理器，当然你要设置一个进行个性化处理也可以，也不需要像之前那样提前给文件描述符设置一个租借锁，因为最终结果还是一样的。

`sendfile()` 相较于 `mmap()` 的另一个优势在于数据在传输过程中始终没有越过用户态和内核态的边界，因此极大地减少了存储管理的开销。即便如此， `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。而且根据 Linus 在 2001 年和其他内核维护者的邮件列表内容，其实当初之所以决定在 Linux 上实现 `sendfile()` 仅仅是因为在其他操作系统平台上已经率先实现了，而且有大名鼎鼎的

Apache Web 服务器已经在使用了，为了兼容 Apache Web 服务器才决定在 Linux 上也实现这个技术，而且 `sendfile()` 实现上的简洁性也和 Linux 内核的其他部分集成得很好，所以 Linus 也就同意了这个提案。

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：

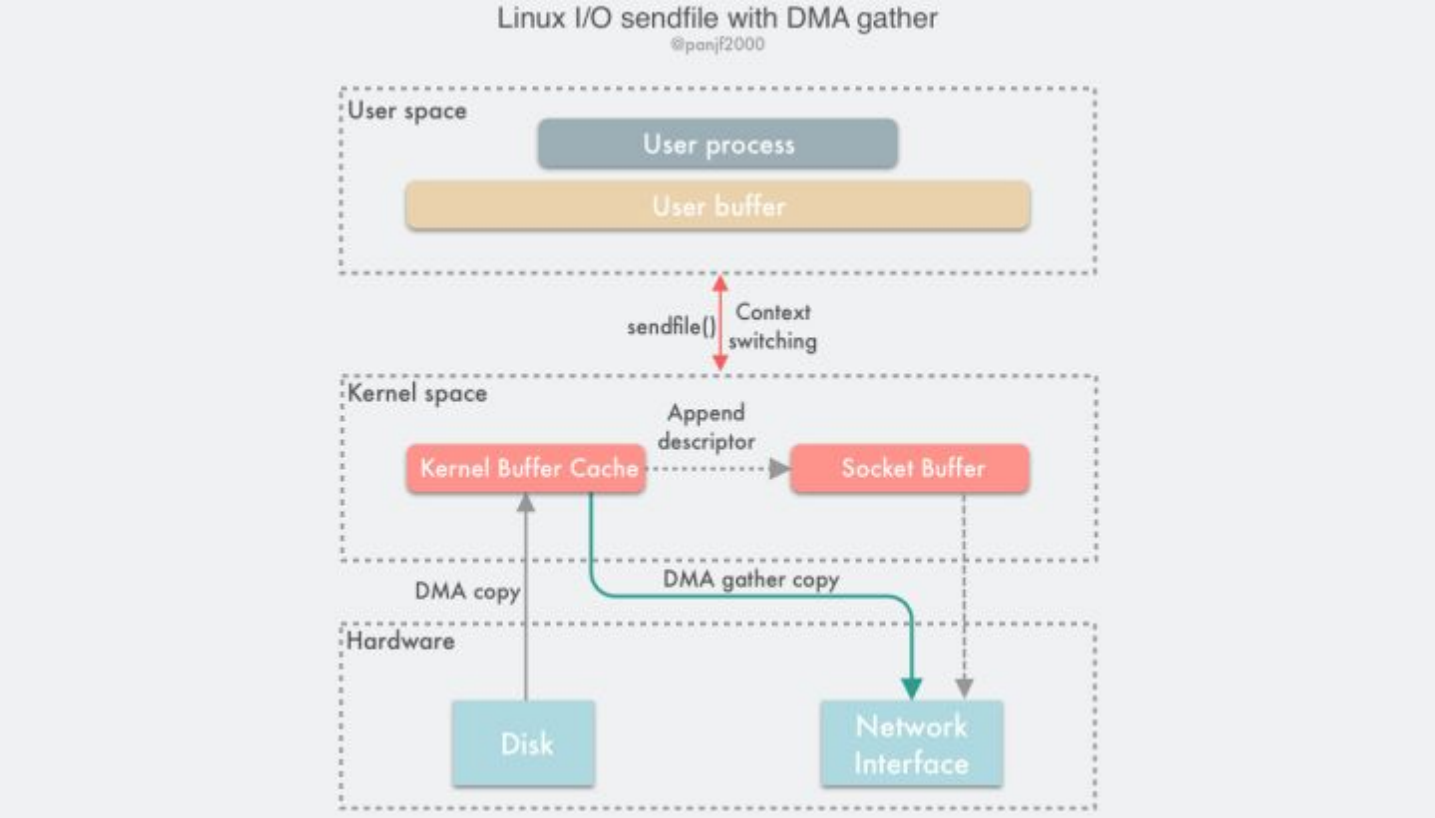
1. 首先一个是这个接口并没有进行标准化，导致 `sendfile()` 在 Linux 上的接口实现和其他类 Unix 系统的实现并不相同；
2. 其次由于网络传输的异步性，很难在接收端实现和 `sendfile()` 对接的技术，因此接收端一直没有实现对应的这种技术；
3. 最后从性能方面考量，因为 `sendfile()` 在把磁盘文件从内核缓冲区（page cache）传输到到套接字缓冲区的过程中依然需要 CPU 参与，这就很难避免 CPU 的高速缓存被传输的数据所污染。

此外，需要说明下，`sendfile()` 的最初设计并不是用来处理大文件的，因此如果需要处理很大的文件的话，可以使用另一个系统调用 `sendfile64()`，它支持对更大的文件内容进行寻址和偏移。

sendfile() with DMA Scatter/Gather Copy

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：



`sendfile() + DMA gather` 的数据传输过程如下：

1. 用户进程调用 `sendfile()`，从用户态陷入内核态；
2. DMA 控制器使用 scatter 功能把数据从硬盘拷贝到内核缓冲区进行离散存储；
3. CPU 把包含内存地址和数据长度的缓冲区描述符拷贝到套接字缓冲区，DMA 控制器能够根据这些信息生成网络包数据分组的报头和报尾
4. DMA 控制器根据缓冲区描述符里的内存地址和数据大小，使用 scatter-gather 功能开始从内核缓冲区收集离散的数据并组包，最后直接把网络包数据拷贝到网卡完成数据传输；
5. `sendfile()` 返回，上下文从内核态切换回用户态。

基于这种方案，我们就可以把这仅剩的唯一一次 CPU 拷贝也给去除了（严格来说还是会有一次，但是因为这次 CPU 拷贝的只是那些微乎其微的元信息，开销几乎可以忽略不计），理论上，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。

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()` 系统调用函数定义如下：

```
#include <fcntl.h>
#include <unistd.h>

int pipe(int pipefd[2]);
int pipe2(int pipefd[2], int flags);

ssize_t splice(int fd_in, loff_t *off_in, int fd_out, loff_t *off_out, size_t len, unsigned int flags);
```

fd_in 和 fd_out 也是分别代表了输入端和输出端的文件描述符，这两个文件描述符必须有一个是指向管道设备的，这也是一个不太友好的限制，虽然 Linux 内核开发的官方从这个系统调用推出之时就承诺未来可能会重构去掉这个限制，然而他们许下这个承诺之后就如同石沉大海，如今 14 年过去了，依旧杳无音讯...

off_in 和 off_out 则分别是 fd_in 和 fd_out 的偏移量指针，指示内核从哪里读取和写入数据，len 则指示了此次调用希望传输的字节数，最后的 flags 是系统调用的标记选项位掩码，用来设置系统调用的行为属性的，由以下 0 个或者多个值通过『或』操作组合而成：

- SPLICE_F_MOVE：指示 `splice()` 尝试仅仅是移动内存页面而不是复制，设置了这个值不代表就一定不会复制内存页面，复制还是移动取决于内核能否从管道中移动内存页面，或者管道中的内存页面是否是完整的；这个标记的初始实现有很多 bug，所以从 Linux 2.6.21 版本开始就已经无效了，但还是保留了下来，因为在未来的版本里可能会重新被实现。
- SPLICE_F_NONBLOCK：指示 `splice()` 不要阻塞 I/O，也就是使得 `splice()` 调用成为一个非阻塞调用，可以用来实现异步数据传输，不过需要注意的是，数据传输的两个文件描述符也最好是预先通过 O_NONBLOCK 标记成非阻塞 I/O，不然 `splice()` 调用还是有可能被阻塞。
- SPLICE_F_MORE：通知内核下一个 `splice()` 系统调用将会有更多的数据传输过来，这个标记对于输出端是 socket 的场景非常有用。

`splice()` 是基于 Linux 的管道缓冲区 (pipe buffer) 机制实现的，所以 `splice()` 的两个入参文件描述符才要求必须有一个是管道设备，一个典型的 `splice()` 用法是：

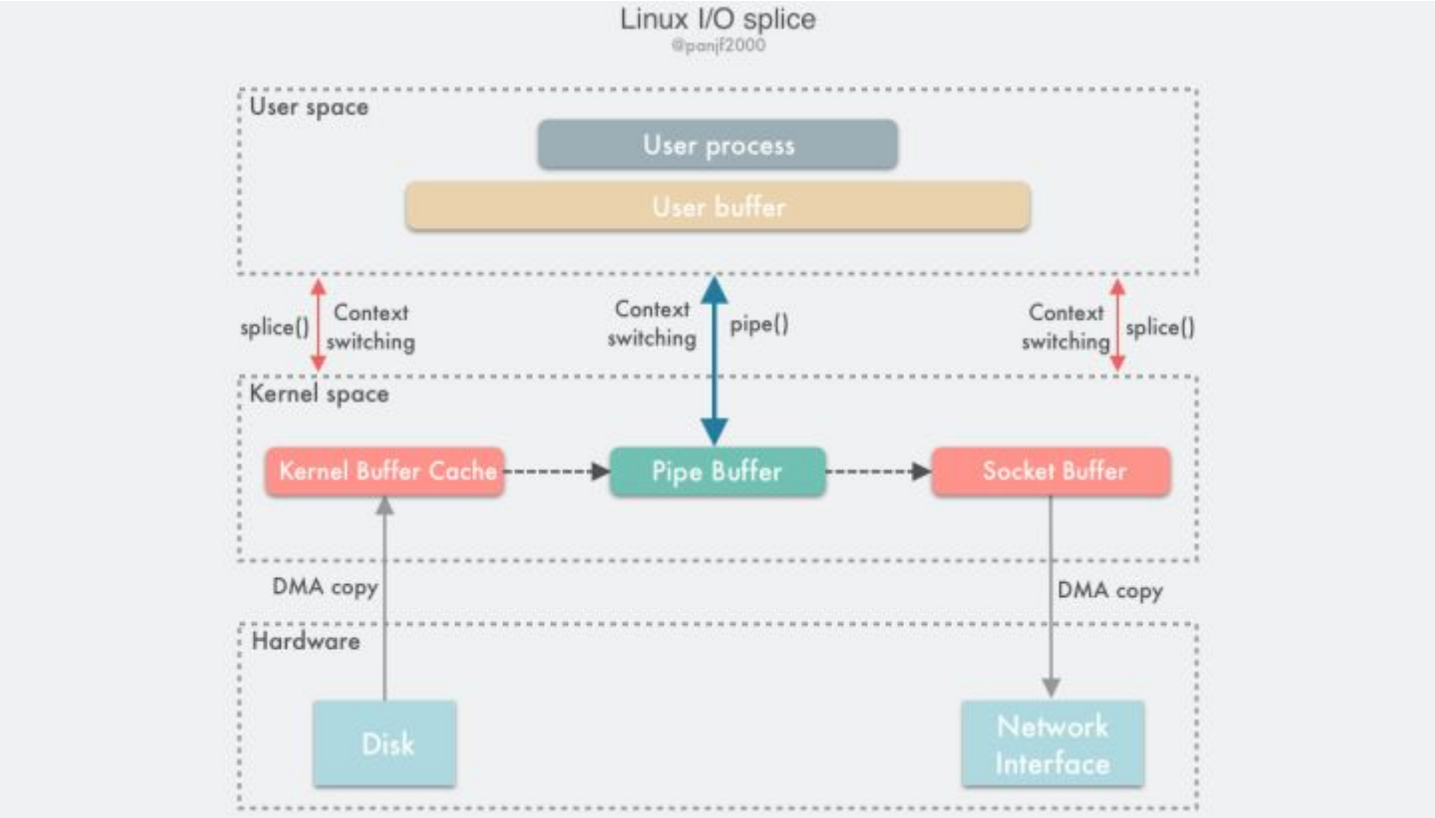
```
int pfd[2];

pipe(pfd);

ssize_t bytes = splice(file_fd, NULL, pfd[1], NULL, 4096, SPLICE_F_MOVE);
assert(bytes != -1);

bytes = splice(pfd[0], NULL, socket_fd, NULL, bytes, SPLICE_F_MOVE | SPLICE_F_MORE);
assert(bytes != -1);
```

数据传输过程图：



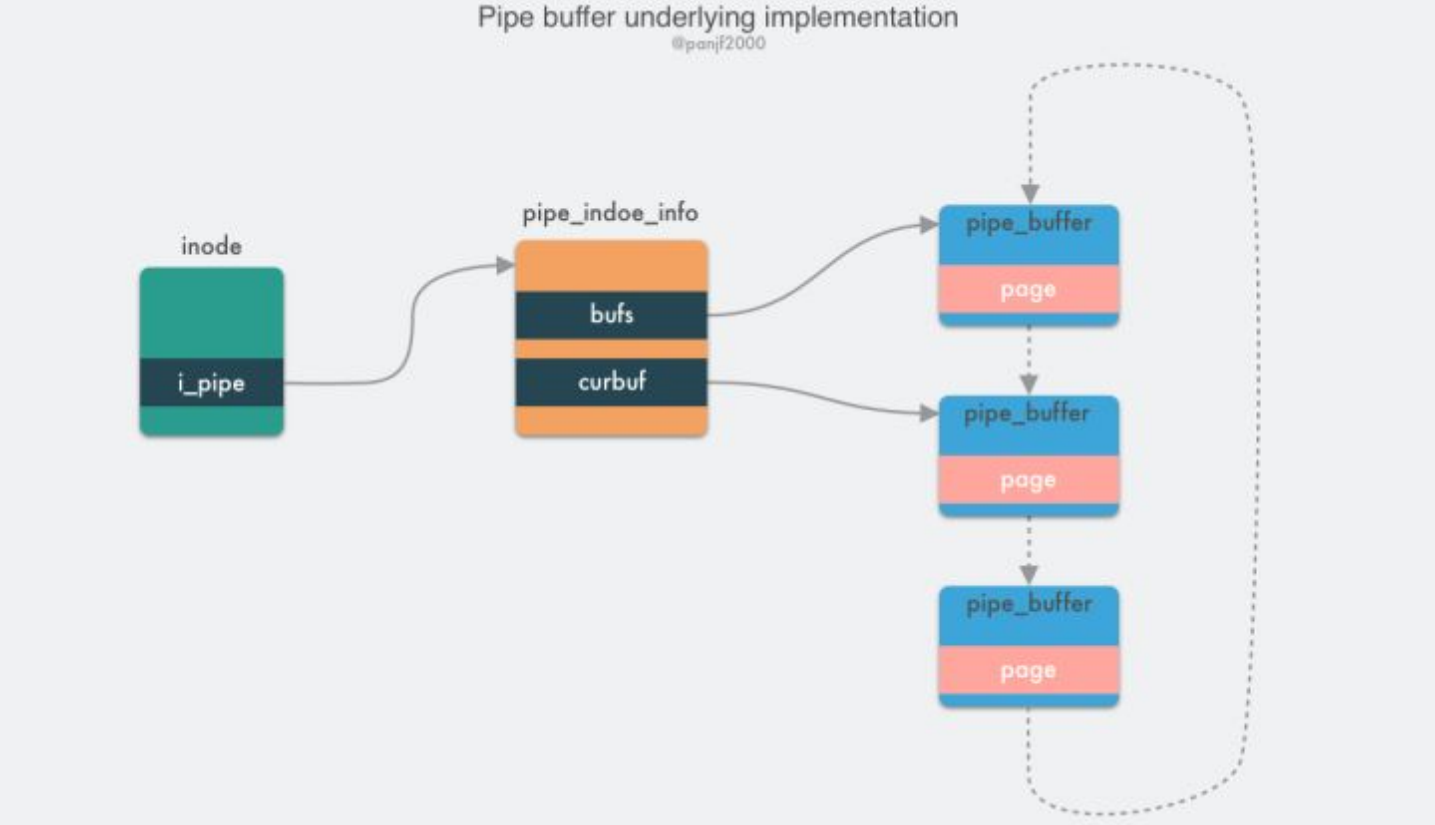
使用 `splice()` 完成一次磁盘文件到网卡的读写过程如下：

1. 用户进程调用 `pipe()`，从用户态陷入内核态，创建匿名单向管道，`pipe()` 返回，上下文从内核态切换回用户态；
2. 用户进程调用 `splice()`，从用户态陷入内核态；
3. DMA 控制器将数据从硬盘拷贝到内核缓冲区，从管道的写入端"拷贝"进管道，`splice()` 返回，上下文从内核态回到用户态；
4. 用户进程再次调用 `splice()`，从用户态陷入内核态；
5. 内核把数据从管道的读取端"拷贝"到套接字缓冲区，DMA 控制器将数据从套接字缓冲区拷贝到网卡；
6. `splice()` 返回，上下文从内核态切换回用户态。

相信看完上面的读写流程之后，读者肯定会非常困惑：说好的 `splice()` 是 `sendfile()` 的改进版呢？`sendfile()` 好歹只需要一次系统调用，`splice()` 居然需要三次，这也就罢了，居然中间还搞出来一个管道，而且还要在内核空间拷贝两次，这算个毛的改进啊？

我最开始了解 `splice()` 的时候，也是这个反应，但是深入学习它之后，才渐渐知晓个中奥妙，且听我细细道来：

先来了解一下 pipe buffer 管道，管道是 Linux 上用来供进程之间通信的信道，管道有两个端：写入端和读出端，从进程的视角来看，管道表现为一个 FIFO 字节流环形队列：



管道本质上是一个内存中的文件，也就是本质上还是基于 Linux 的 VFS，用户进程可以通过 `pipe()` 系统调用创建一个匿名管道，创建完成之后会有两个 VFS 的 file 结构体的 inode 分别指向其写入端和读出端，并返回对应的两个文件描述符，用户进程通过这两个文件描述符读写管道；管道的容量单位是一个虚拟内存的页，也就是 4KB，总大小一般是 16 个页，基于其环形结构，管道的页可以循环使用，提高内存利用率。 Linux 中以 pipe_buffer 结构体封装管道页，file 结构体里的 inode 字段里会保存一个 pipe_inode_info 结构体指代管道，其中会保存很多读写管道时所需的元信息，环形队列的头部指针页，读写时的同步机制如互斥锁、等待队列等：

```
struct pipe_buffer {
 struct page *page; // 内存页结构
 unsigned int offset, len; // 偏移量，长度
 const struct pipe_buf_operations *ops;
 unsigned int flags;
 unsigned long private;
};

struct pipe_inode_info {
 struct mutex mutex;
 wait_queue_head_t wait;
 unsigned int nrbufs, curbuf, buffers;
 unsigned int readers;
 unsigned int writers;
 unsigned int files;
 unsigned int waiting_writers;
 unsigned int r_counter;
 unsigned int w_counter;
 struct page *tmp_page;
 struct fasync_struct *fasync_readers;
 struct fasync_struct *fasync_writers;
 struct pipe_buffer *bufs;
 struct user_struct *user;
};
```

pipe_buffer 中保存了数据在内存中的页、偏移量和长度，以这三个值来定位数据，注意这里的页不是虚拟内存的页，而用的是物理内存的页框，因为管道时跨进程的信道，因此不能使用虚拟内存来表示，只能使用物理内存的页框定位数据；管道的正常读写操作是通过 pipe_write()/pipe_read() 来完成的，通过把数据读取/写入环形队列的 pipe_buffer 来完成数据传输。

`splice()` 是基于 pipe buffer 实现的，但是它在通过管道传输数据的时候却是零拷贝，因为它在写入读出时并没有使用 pipe_write()/pipe_read() 真正地在管道缓冲区写入读出数据，而是通过把数据在内存缓冲区中的物理内存页框指针、偏移量和长度赋值给前文提及的 pipe_buffer 中对应的三个字段来完成数据的"拷贝"，也就是其实只拷贝了数据的内存地址等元信息。

`splice()` 在 Linux 内核源码中的内部实现是 `do_splice()` 函数，而写入读出管道则分别是通过 `do_splice_to()` 和 `do_splice_from()`，这里我们重点来解析下写入管道的源码，也就是 `do_splice_to()`，我现在手头的 Linux 内核版本是 v4.8.17，我们就基于这个版本来分析，至于读出的源码函数 `do_splice_from()`，原理是相通的，大家举一反三即可。

`splice()` 写入数据到管道的调用链式：`do_splice()` --> `do_splice_to()` --> `splice_read()`

```
static long do_splice(struct file *in, loff_t __user *off_in,
        struct file *out, loff_t __user *off_out,
        size_t len, unsigned int flags)
{
...

 // 判断是写出 fd 是一个管道设备，则进入数据写入的逻辑
 if (opipe) {
  if (off_out)
   return -ESPIPE;
  if (off_in) {
   if (!(in->f_mode & FMODE_PREAD))
    return -EINVAL;
   if (copy_from_user(&offset, off_in, sizeof(loff_t)))
    return -EFAULT;
  } else {
   offset = in->f_pos;
  }

  // 调用 do_splice_to 把文件内容写入管道
  ret = do_splice_to(in, &offset, opipe, len, flags);

  if (!off_in)
   in->f_pos = offset;
  else if (copy_to_user(off_in, &offset, sizeof(loff_t)))
   ret = -EFAULT;

  return ret;
 }

 return -EINVAL;
}
```

进入 `do_splice_to()` 之后，再调用 `splice_read()`：

```
static long do_splice_to(struct file *in, loff_t *ppos,
    struct pipe_inode_info *pipe, size_t len,
    unsigned int flags)
{
 ssize_t (*splice_read)(struct file *, loff_t *,
          struct pipe_inode_info *, size_t, unsigned int);
 int ret;

 if (unlikely(!(in->f_mode & FMODE_READ)))
  return -EBADF;

 ret = rw_verify_area(READ, in, ppos, len);
 if (unlikely(ret < 0))
  return ret;

 if (unlikely(len > MAX_RW_COUNT))
  len = MAX_RW_COUNT;
```

```
 // 判断文件的文件的 file 结构体的 f_op 中有没有可供使用的、支持 splice 的 splice_read 函数指针
 // 因为是 splice() 调用，因此内核会提前给这个函数指针指派一个可用的函数
 if (in->f_op->splice_read)
  splice_read = in->f_op->splice_read;
 else
  splice_read = default_file_splice_read;

 return splice_read(in, ppos, pipe, len, flags);
}
```

`in->f_op->splice_read` 这个函数指针根据文件描述符的类型不同有不同的实现，比如这里的 in 是一个文件，因此是 `generic_file_splice_read()`，如果是 socket 的话，则是 `sock_splice_read()`，其他的类型也会有对应的实现，总之我们这里将使用的是 `generic_file_splice_read()` 函数，这个函数会继续调用内部函数 `__generic_file_splice_read` 完成以下工作：

1. 在 page cache 页缓存里进行搜寻，看看我们要读取这个文件内容是否已经在缓存里了，如果是则直接用，否则如果不存在或者只有部分数据在缓存中，则分配一些新的内存页并进行读入数据操作，同时会增加页框的引用计数；
2. 基于这些内存页，初始化 splice_pipe_desc 结构，这个结构保存会保存文件数据的地址元信息，包含有物理内存页框地址，偏移、数据长度，也就是 pipe_buffer 所需的三个定位数据的值；
3. 最后，调用 `splice_to_pipe()`，splice_pipe_desc 结构体实例是函数入参。

```
ssize_t splice_to_pipe(struct pipe_inode_info *pipe, struct splice_pipe_desc *spd)
{
...

 for (;;) {
  if (!pipe->readers) {
   send_sig(SIGPIPE, current, 0);
   if (!ret)
    ret = -EPIPE;
   break;
  }

  if (pipe->nrbufs < pipe->buffers) {
   int newbuf = (pipe->curbuf + pipe->nrbufs) & (pipe->buffers - 1);
   struct pipe_buffer *buf = pipe->bufs + newbuf;

   // 写入数据到管道，没有真正拷贝数据，而是内存地址指针的移动，
   // 把物理页框、偏移量和数据长度赋值给 pipe_buffer 完成数据入队操作
   buf->page = spd->pages[page_nr];
   buf->offset = spd->partial[page_nr].offset;
   buf->len = spd->partial[page_nr].len;
   buf->private = spd->partial[page_nr].private;
   buf->ops = spd->ops;
   if (spd->flags & SPLICE_F_GIFT)
    buf->flags |= PIPE_BUF_FLAG_GIFT;

   pipe->nrbufs++;
   page_nr++;
   ret += buf->len;

   if (pipe->files)
    do_wakeup = 1;

   if (!--spd->nr_pages)
    break;
   if (pipe->nrbufs < pipe->buffers)
    continue;

   break;
  }
```

```
 ...
}
```

这里可以清楚地看到 `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。

即使 `splice()` 通过内存地址指针避免了真正的拷贝开销，但是算起来它还要使用额外的管道来完成数据传输，也就是比 `sendfile()` 多了两次系统调用，这不是又增加了上下文切换的开销吗？为什么不直接在内核创建管道并调用那两次 `splice()`，然后只暴露给用户一次系统调用呢？实际上因为 `splice()` 利用管道而非硬件来完成零拷贝的实现比 `sendfile()` + DMA Scatter/Gather 的门槛更低，因此后来的 `sendfile()` 的底层实现就已经替换成 `splice()` 了。

至于说 `splice()` 本身的 API 为什么还是这种使用模式，那是因为 Linux 内核开发团队一直想把基于管道的这个限制去掉，但不知道因为什么一直搁置，所以这个 API 也就一直没变化，只能等内核团队哪天想起来了这一茬，然后重构一下使之不再依赖管道，在那之前，使用 `splice()` 依然还是需要额外创建管道来作为中间缓冲，如果你的业务场景很适合使用 `splice()`，但又是性能敏感的，不想频繁地创建销毁 pipe buffer 管道缓冲区，那么可以参考一下 HAProxy 使用 `splice()` 时采用的优化方案：预先分配一个 pipe buffer pool 缓存管道，每次调用 `spclie()` 的时候去缓存池里取一个管道，用完就放回去，循环利用，提升性能。

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。

这个功能的使用模式如下：

```
if (setsockopt(socket_fd, SOL_SOCKET, SO_ZEROCOPY, &one, sizeof(one)))
        error(1, errno, "setsockopt zerocopy");

ret = send(socket_fd, buffer, sizeof(buffer), MSG_ZEROCOPY);
```

首先第一步，先给要发送数据的 socket 设置一个 SOCK_ZEROCOPY option，然后在调用 `send()` 发送数据时再设置一个 MSG_ZEROCOPY option，其实理论上来说只需要调用 `setsockopt()` 或者 `send()` 时传递这个 zero-copy 的 option 即可，两者选其一，但是这里却要设置同一个 option 两次，官方的说法是为了兼容 `send()` API 以前的设计上的一个错误：`send()` 以前的实现会忽略掉未知的 option，为了兼容那些可能已经不小心设置了 MSG_ZEROCOPY option 的程序，故而设计成了两步设置。不过我猜还有一种可能：就是给使用者提供更灵活的使用模式，因为这个新功能只在大包场景下才可能会有显著的性能提升，但是现实场景是很复杂的，不仅仅是全部大包或者全部小包的场景，有可能是大包小包混合的场景，因此使用者可以先调用 `setsockopt()` 设置 SOCK_ZEROCOPY option，然后再根据实际业务场景中的网络包尺寸选择是否要在调用 `send()` 时使用 MSG_ZEROCOPY 进行 zero-copy 传输。

因为 `send()` 可能是异步发送数据，因此使用 MSG_ZEROCOPY 有一个需要特别注意的点是：调用 `send()` 之后不能立刻重用或释放 buffer，因为 buffer 中的数据不一定已经被内核读走了，所以还需要从 socket 关联的错误队列里读取一下通知消息，看看 buffer 中的数据是否已经被内核读走了：

```
pfd.fd = fd;
pfd.events = 0;
if (poll(&pfd, 1, -1) != 1 || pfd.revents & POLLERR == 0)
        error(1, errno, "poll");

ret = recvmsg(fd, &msg, MSG_ERRQUEUE);
if (ret == -1)
        error(1, errno, "recvmsg");

read_notification(msg);
```

```
uint32_t read_notification(struct msghdr *msg)
{
 struct sock_extended_err *serr;
 struct cmsghdr *cm;

 cm = CMSG_FIRSTHDR(msg);
 if (cm->cmsg_level != SOL_IP &&
  cm->cmsg_type != IP_RECVERR)
   error(1, 0, "cmsg");

 serr = (void *) CMSG_DATA(cm);
 if (serr->ee_errno != 0 ||
  serr->ee_origin != SO_EE_ORIGIN_ZEROCOPY)
   error(1, 0, "serr");

 return serr->ee _ data;
}
```

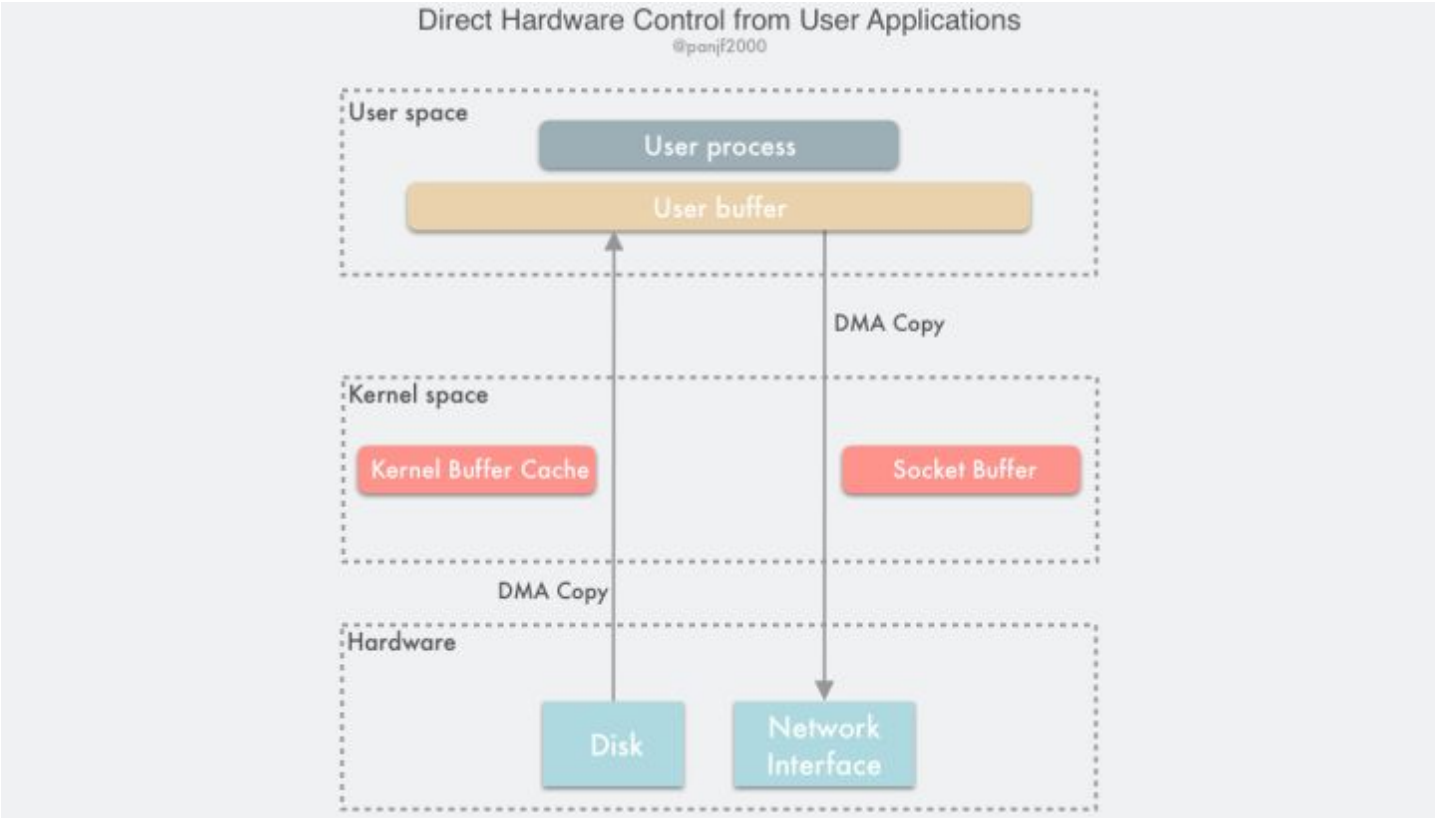
这个技术是基于 redhat 红帽在 2010 年给 Linux 内核提交的 virtio-net zero-copy 技术之上实现的，至于底层原理，简单来说就是通过 `send()` 把数据在用户缓冲区中的分段指针发送到 socket 中去，利用 page pinning 页锁定机制锁住用户缓冲区的内存页，然后利用 DMA 直接在用户缓冲区通过内存地址指针进行数据读取，实现零拷贝；具体的细节可以通过阅读 Willem de Bruijn 的**论文 (PDF)** 深入了解。

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。

绕过内核的直接 I/O

可以看出，前面种种的 zero-copy 的方法，都是在想方设法地优化减少或者去掉用户态和内核态之间以及内核态和内核态之间的数据拷贝，为了实现避免这些拷贝可谓是八仙过海，各显神通，采用了各种各样的手段，那么如果我们换个思路：其实这么费劲地去消除这些拷贝不就是因为有内核在掺和吗？如果我们绕过内核直接进行 I/O 不就没有这些烦人的拷贝问题了吗？这就是**绕过内核直接 I/O** 技术：



这种方案有两种实现方式：

1. 用户直接访问硬件
2. 内核控制访问硬件

用户直接访问硬件

这种技术赋予用户进程直接访问硬件设备的权限，这让用户进程能有直接读写硬件设备，在数据传输过程中只需要内核做一些虚拟内存配置相关的工作。这种无需数据拷贝和内核干预的直接 I/O，理论上是最高效的数据传输技术，但是正如前面所说的那样，并不存在能解决一切问题的银弹，这种直接 I/O 技术虽然有可能非常高效，但是它的适用性也非常窄，目前只适用于诸如 MPI 高性能通信、丛集计算系统中的远程共享内存等有限的场景。

这种技术实际上破坏了现代计算机操作系统最重要的概念之一 —— 硬件抽象，我们之前提过，抽象是计算机领域最最核心的设计思路，正式由于有了抽象和分层，各个层级才能不必去关心很多底层细节从而专注于真正的工作，才使得系统的运作更加高效和快速。此外，网卡通常使用功能较弱的 CPU，例如只包含简单指令集的 MIPS 架构处理器（没有不必要的功能，如浮点数计算等），也没有太多的内存来容纳复杂的软件。因此，通常只有那些基于以太网之上的专用协议会使用这种技术，这些专用协议的设计要比远比 TCP/IP 简单得多，而且多用于局域网环境中，在这种环境中，数据包丢失和损坏很少发生，因此没有必要进行复杂的数据包确认和流量控制机制。而且这种技术还需要定制的网卡，所以它是高度依赖硬件的。

与传统的通信设计相比，直接硬件访问技术给程序设计带来了各种限制：由于设备之间的数据传输是通过 DMA 完成的，因此用户空间的数据缓冲区内存页必须进行 page pinning（页锁定），这是为了防止其物理页框地址被交换到磁盘或者被移动到新的地址而导致 DMA 去拷贝数据的时候在指定的地址找不到内存页从而引发缺页错误，而页锁定的开销并不比 CPU 拷贝小，所以为了避免频繁的页锁定系统调用，应用程序必须分配和注册一个持久的内存池，用于数据缓冲。

用户直接访问硬件的技术可以得到极高的 I/O 性能，但是其应用领域和适用场景也极其的有限，如集群或网络存储系统中的节点通信。它需要定制的硬件和专门设计的应用程序，但相应地对操作系统内核的改动比较小，可以很容易地以内核模块或设备驱动程序的形式实现出来。直接访问硬件还可能会带来严重的安全问题，因为用户进程拥有直接访问硬件的极高权限，所以如果你的程序设计没有做好的话，可能会消耗本来就有限的硬件资源或者进行非法地址访问，可能也会因此间接地影响其他正在使用同一设备的应用程序，而因为绕开了内核，所以也无法让内核替你去控制和管理。

内核控制访问硬件

相较于用户直接访问硬件技术，通过内核控制的直接访问硬件技术更加的安全，它比前者在数据传输过程中会多干预一点，但也仅仅是作为一个代理人这样的角色，不会参与到实际的数据传输过程，内核会控制 DMA 引擎去替用户进程做缓冲区的数据传输工作。同样的，这种方式也是高度依赖硬件的，比如一些集成了专有网络栈协议的网卡。这种技术的一个优势就是用户集成去 I/O 时的接口不会改变，就和普通的 `read()/write()` 系统调用那样使用即可，所有的脏活累活都在内核里完成，用户接口友好度很高，不过需要注意的是，使用这种技术的过程中如果发生了什么不可预知的意外从而导致无法使用这种技术进行数据传输的话，则内核会自动切换为最传统 I/O 模式，也就是性能最差的那种模式。

这种技术也有着和用户直接访问硬件技术一样的问题：DMA 传输数据的过程中，用户进程的缓冲区内存页必须进行 page pinning 页锁定，数据传输完成后才能解锁。CPU 高速缓存内保存的多个内存地址也会被冲刷掉以保证 DMA 传输前后的数据一致性。这些机制有可能会导致数据传输的性能变得更差，因为 `read()/write()` 系统调用的语义并不能提前通知 CPU 用户缓冲区要参与 DMA 数据传输传输，因此也就无法像内核缓冲区那样可依提前加载进高速缓存，提高性能。由于用户缓冲区的内存页可能分布在物理内存中的任意位置，因此一些实现不好的 DMA 控制器引擎可能会有寻址限制从而导致无法访问这些内存区域。一些技术比如 AMD64 架构中的 IOMMU，允许通过将 DMA 地址重新映射到内存中的物理地址来解决这些限制，但反过来又可能会导致可移植性问题，因为其他的处理器架构，甚至是 Intel 64 位 x86 架构的变种 EM64T 都不具备这样的特性单元。此外，还可能存在其他限制，比如 DMA 传输的数据对齐问题，又会导致无法访问用户进程指定的任意缓冲区内存地址。

内核缓冲区和用户缓冲区之间的传输优化

到目前为止，我们讨论的 zero-copy 技术都是基于减少甚至是避免用户空间和内核空间之间的 CPU 数据拷贝的，虽然有一些技术非常高效，但是大多都有适用性很窄的问题，比如 `sendfile()`、`splice()` 这些，效率很高，但是都只适用于那些用户进程不需要直接处理数据的场景，比如静态文件服务器或者是直接转发数据的代理服务器。

现在我们已经知道，硬件设备之间的数据可以通过 DMA 进行传输，然而却并没有这样的传输机制可以应用于用户缓冲区和内核缓冲区之间的数据传输。不过另一方面，广泛应用在现代的 CPU 架构和操作系统上的虚拟内存机制表明，通过在不同的虚拟地址上重新映射页面可以实现在用户进程和内核之间虚拟复制和共享内存，尽管一次传输的内存颗粒度相对较大：4KB 或 8KB。

因此如果要在实现在用户进程内处理数据（这种场景比直接转发数据更加常见）之后再发送出去的话，用户空间和内核空间的数据传输就是不可避免的，既然避无可避，那就只能选择优化了，因此本章节我们要介绍两种优化用户空间和内核空间数据传输的技术：

1. 动态重映射与写时拷贝 (Copy-on-Write)
2. 缓冲区共享 (Buffer Sharing)

动态重映射与写时拷贝 (Copy-on-Write)

前面我们介绍过利用内存映射技术来减少数据在用户空间和内核空间之间的复制，通常简单模式下，用户进程是对共享的缓冲区进行同步阻塞读写的，这样不会有 data race 问题，但是这种模式下效率并不高，而提升效率的一种方法就是异步地对共享缓冲区进行读写，而这样的话就必须引入保护机制来避免数据冲突问题，写时复制 (Copy on Write) 就是这样的一种技术。

> **写入时复制**（**Copy-on-write**，**COW**）是一种计算机**程序设计**领域的优化策略。其核心思想是，如果有多个调用者（callers）同时请求相同资源（如内存或磁盘上的数据存储），他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源的内容时，系统才会真正复制一份专用副本（private copy）给该调用者，而其他调用者所见到的最初的资源仍然保持不变。这过程对其他的调用者都是**透明**的。此作法主要的优点是如果调用者没有修改该资源，就不会有副本（private copy）被创建，因此多个调用者只是读取操作时可以共享同一份资源。

举一个例子，引入了 COW 技术之后，用户进程读取磁盘文件进行数据处理最后写到网卡，首先使用内存映射技术让用户缓冲区和内核缓冲区共享了一段内存地址并标记为只读 (read-only)，避免数据拷贝，而当要把数据写到网卡的时候，用户进程选择了异步写的方式，系统调用会直接返回，数据传输就会在内核里异步进行，而用户进程就可以继续其他的工作，并且共享缓冲区的内容可以随时再进行读取，效率很高，但是如果该进程又尝试往共享缓冲区写入数据，则会产生一个 COW 事件，让试图写入数据的进程把数据复制到自己的缓冲区去修改，这里只需要复制要修改的内存页即可，无需所有数据都复制过去，而如果其他访问该共享内存的进程不需要修改数据则可以永远不需要进行数据拷贝。

COW 是一种建构在虚拟内存冲映射技术之上的技术，因此它需要 MMU 的硬件支持，MMU 会记录当前哪些内存页被标记成只读，当有进程尝试往这些内存页中写数据的时候，MMU 就会抛一个异常给操作系统内核，内核处理该异常时为该进程分配一份物理内存并复制数据到此内存地址，重新向 MMU 发出执行该进程的写操作。

COW 最大的优势是节省内存和减少数据拷贝，不过却是通过增加操作系统内核 I/O 过程复杂性作为代价的。当确定采用 COW 来复制页面时，重要的是注意空闲页面的分配位置。许多操作系统为这类请求提供了一个空闲的页面池。当进程的堆栈或堆要扩展时或有写时复制页面需要管理时，通常分配这些空闲页面。操作系统分配这些页面通常采用称为**按需填零**的技术。按需填零页面在需要分配之前先填零，因此会清除里面旧的内容。

局限性：

COW 这种零拷贝技术比较适用于那种多读少写从而使得 COW 事件发生较少的场景，因为 COW 事件所带来的系统开销要远远高于一次 CPU 拷贝所产生的。此外，在实际应用的过程中，为了避免频繁的内存映射，可以重复使用同一段内存缓冲区，因此，你不需要在只用过一次共享缓冲区之后就解除掉内存页的映射关系，而是重复循环使用，从而提升性能，不过这种内存页映射的持久化并不会减少由于页表往返移动和 TLB 冲刷所带来的系统开销，因为每次接收到 COW 事件之后对内存页而进行加锁或者解锁的时候，页面的只读标志 (read-ony) 都要被更改为 (write-only)。

缓冲区共享 (Buffer Sharing)

从前面的介绍可以看出，传统的 Linux I/O 接口，都是基于复制/拷贝的：数据需要在操作系统内核空间和用户空间的缓冲区之间进行拷贝。在进行 I/O 操作之前，用户进程需要预先分配好一个内存缓冲区，使用 `read()` 系统调用时，内核会将从存储器或者网卡等设备读入的数据拷贝到这个用户缓冲区里；而使用 `write()` 系统调用时，则是把用户内存缓冲区的数据拷贝至内核缓冲区。

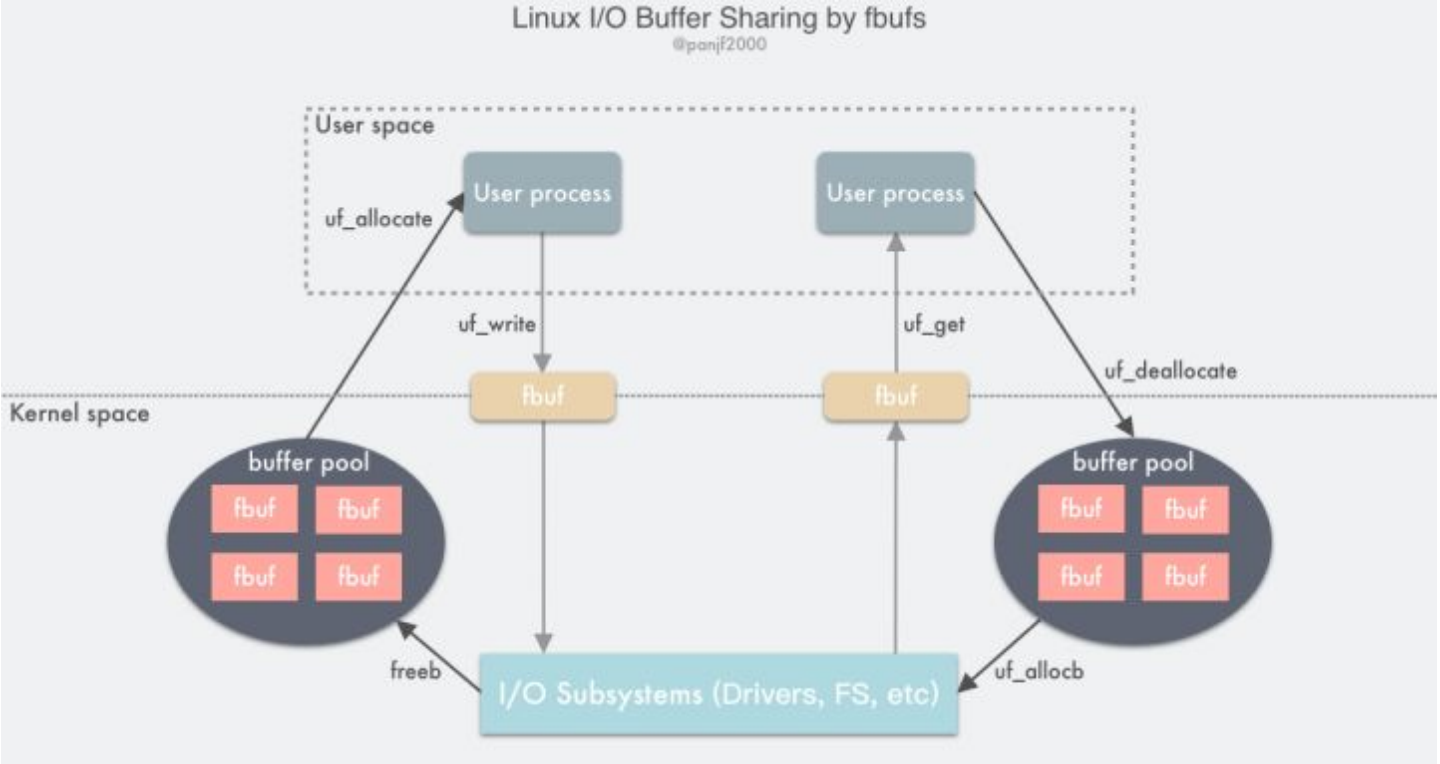
为了实现这种传统的 I/O 模式，Linux 必须要在每一个 I/O 操作时都进行内存虚拟映射和解除。这种内存页重映射的机制的效率严重受限于缓存体系结构、MMU 地址转换速度和 TLB 命中率。如果能够避免处理 I/O 请求的虚拟地址转换和 TLB 刷新所带来的开销，则有可能极大地提升 I/O 性能。而缓冲区共享就是用来解决上述问题的一种技术。

最早支持 Buffer Sharing 的操作系统是 Solaris。后来，Linux 也逐步支持了这种 Buffer Sharing 的技术，但时至今日依然不够完整和成熟。

操作系统内核开发者们实现了一种叫 fbufs 的缓冲区共享的框架，也即**快速缓冲区（Fast Buffers）**，使用一个 fbuf 缓冲区作为数据传输的最小单位，使用这种技术需要调用新的操作系统 API，用户区和内核区、内核区之间的数据都必须严格地在 fbufs 这个体系下进行通信。fbufs 为每一个用户进程分配一个 buffer pool，里面会储存预分配 (也可以使用的时候再分配) 好的 buffers，这些 buffers 会被同时映射到用户内存空间和内核内存空间。fbufs 只需通过一次虚拟内存映射操作即可创建缓冲区，有效地消除那些由存储一致性维护所引发的大多数性能损耗。

传统的 Linux I/O 接口是通过把数据在用户缓冲区和内核缓冲区之间进行拷贝传输来完成的，这种数据传输过程中需要进行大量的数据拷贝，同时由于虚拟内存技术的存在，I/O 过程中还需要频繁地通过 MMU 进行虚拟内存地址到物理内存地址的转换，高速缓存的汰换以及 TLB 的刷新，这些操作均会导致性能的损耗。而如果利用 fbufs 框架来实现数据传输的话，首先可以把 buffers 都缓存到 pool 里循环利用，而不需要每次都去重

新分配，而且缓存下来的不止有 buffers 本身，而且还会把虚拟内存地址到物理内存地址的映射关系也缓存下来，也就可以避免每次都进行地址转换，从发送接收数据的层面来说，用户进程和 I/O 子系统比如设备驱动程序、网卡等可以直接传输整个缓冲区本身而不是其中的数据内容，也可以理解成是传输内存地址指针，这样就就避免了大量的数据内容拷贝：用户进程/ IO 子系统通过发送一个个的 fbuf 写出数据到内核而非直接传递数据内容，相对应的，用户进程/ IO 子系统通过接收一个个的 fbuf 而从内核读入数据，这样就能减少传统的 `read()/write()` 系统调用带来的数据拷贝开销：



1. 发送方用户进程调用 `uf_allocate` 从自己的 buffer pool 获取一个 fbuf 缓冲区，往其中填充内容之后调用 `uf_write` 向内核区发送指向 fbuf 的文件描述符；
2. I/O 子系统接收到 fbuf 之后，调用 `uf_allocb` 从接收方用户进程的 buffer pool 获取一个 fubf 并用接收到的数据进行填充，然后向用户区发送指向 fbuf 的文件描述符；
3. 接收方用户进程调用 `uf_get` 接收到 fbuf，读取数据进行处理，完成之后调用 `uf_deallocate` 把 fbuf 放回自己的 buffer pool。

fbufs 的缺陷

共享缓冲区技术的实现需要依赖于用户进程、操作系统内核、以及 I/O 子系统 (设备驱动程序，文件系统等)之间协同工作。比如，设计得不好的用户进程容易就会修改已经发送出去的 fbuf 从而污染数据，更要命的是这种问题很难 debug。虽然这个技术的设计方案非常精彩，但是它的门槛和限制却不比前面介绍的其他技术少：首先会对操作系统 API 造成变动，需要使用新的一些 API 调用，其次还需要设备驱动程序配合改动，还有由于是内存共享，内核需要很小心谨慎地实现对这部分共享的内存进行数据保护和同步的机制，而这种并发的同步机制是非常容易出 bug 的从而又增加了内核的代码复杂度，等等。因此这一类的技术还远远没有到发展成熟和广泛应用的阶段，目前大多数的实现都还处于实验阶段。

总结

本文中我主要讲解了 Linux I/O 底层原理，然后介绍并解析了 Linux 中的 Zero-copy 技术，并给出了 Linux 对 I/O 模块的优化和改进思路。

Linux 的 Zero-copy 技术可以归纳成以下三大类：

- **减少甚至避免用户空间和内核空间之间的数据拷贝**：在一些场景下，用户进程在数据传输过程中并不需要对数据进行访问和处理，那么数据在 Linux 的 `Page Cache` 和用户进程的缓冲区之间的传输就完全可以避免，让数据拷贝完全在内核里进行，甚至可以通过更巧妙的方式避免在内核里的数据拷贝。这一类实现一般是是通过增加新的系统调用来完成的，比如 Linux 中的 mmap()，sendfile() 以及 splice() 等。
- **绕过内核的直接 I/O**：允许在用户态进程绕过内核直接和硬件进行数据传输，内核在传输过程中只负责一些管理和辅助的工作。这种方式其实和第一种有点类似，也是试图避免用户空间和内核空间之间的数据传输，只是第一种方式是把数据传输过程放在内核态完成，而这种方式则是直接绕过内核和硬件通信，效果类似但原理完全不同。
- **内核缓冲区和用户缓冲区之间的传输优化**：这种方式侧重于在用户进程的缓冲区和操作系统的页缓存之间的 CPU 拷贝的优化。这种方法延续了以往那种传统的通信方式，但更灵活。

本文从虚拟内存、I/O 缓冲区，用户态&内核态以及 I/O 模式等等知识点全面而又详尽地剖析了 Linux 系统的 I/O 底层原理，分析了 Linux 传统的 I/O 模式的弊端，进而引入 Linux Zero-copy 零拷贝技术的介绍和原理解析，通过将零拷贝技术和传统的 I/O 模式进行区分和对比，带领读者经历了 Linux I/O 的演化历史，通过帮助读者理解 Linux 内核对 I/O 模块的优化改进思路，相信不仅仅是让读者了解 Linux 底层系统的设计原理，更能对读者们在以后优化改进自己的程序设计过程中能够有所启发。

参考&延伸阅读

- **MODERN OPERATING SYSTEMS**
- **Zero Copy I: User-Mode Perspective**
- **Message Passing for Gigabit/s Networks with “Zero-Copy” under Linux**
- **ZeroCopy: Techniques, Benefits and Pitfalls**
- **Zero-copy networking**
- **Driver porting: Zero-copy user-space access**
- **sendmsg copy avoidance with MSG_ZEROCOPY**
- **It’s all about buffers: zero-copy, mmap and Java NIO**
- **Linux Zero Copy**
- **Two new system calls: splice() and sync_file_range()**
- **Circular pipes**
- **The future of the page cache**
- Provide a zero-copy method on KVM virtio-net.

发布于 2020-11-23 17:39

Linux 内核　操作系统　Linux