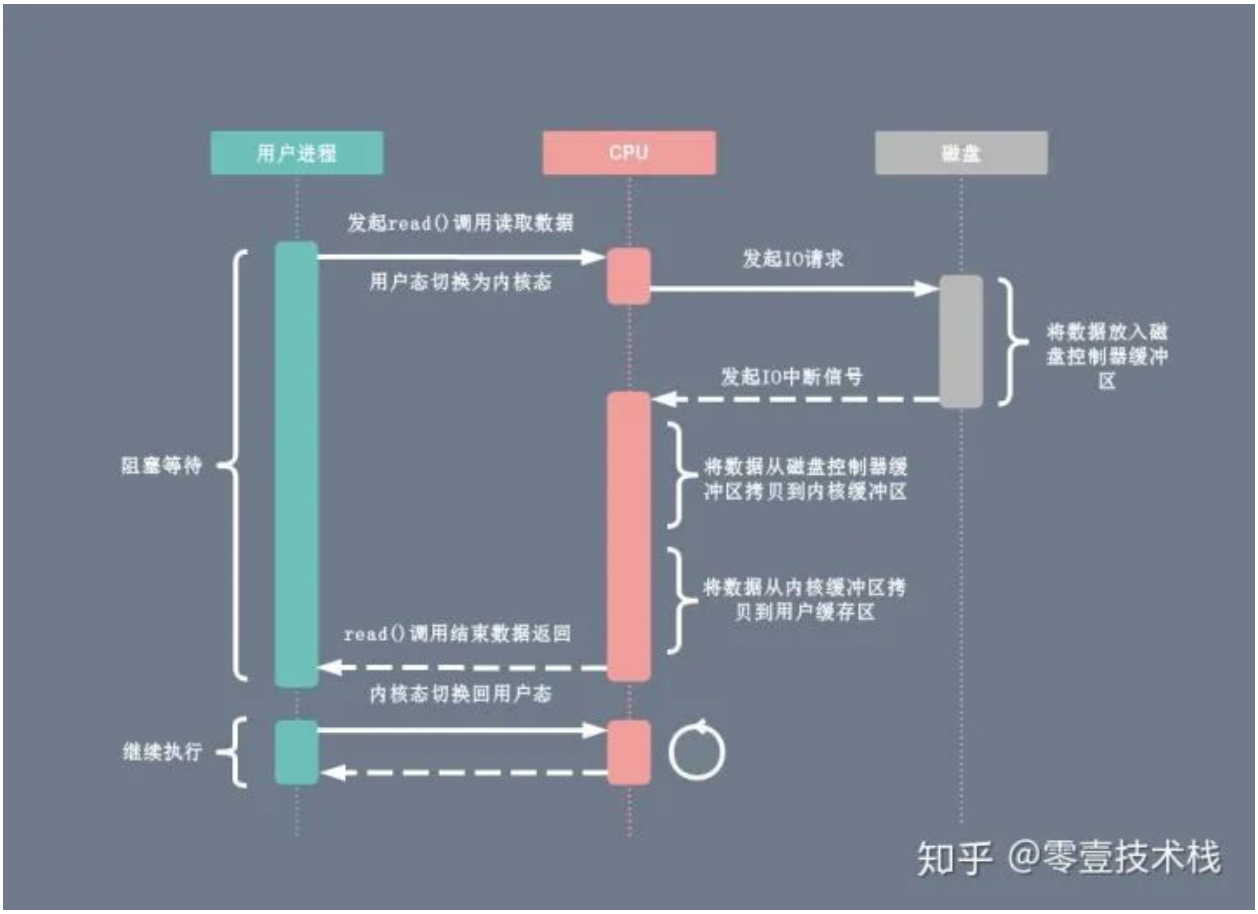
深入分析IO原理和几种零拷贝方式

在 DMA 技术出现之前，应用程序与磁盘之间的 I/O 操作都是通过 CPU 的中断完成的。每次用户进程读取磁盘数据时，都需要 CPU 中断，然后发起 I/O 请求等待数据读取和拷贝完成，每次的 I/O 中断都导致 CPU 的上下文切换。



1. 用户进程向 CPU 发起 read 系统调用读取数据，由用户态切换为内核态，然后一直阻塞等待数据的返回。

2. CPU 在接收到指令以后对磁盘发起 I/O 请求，将磁盘数据先放入磁盘控制器缓冲区。

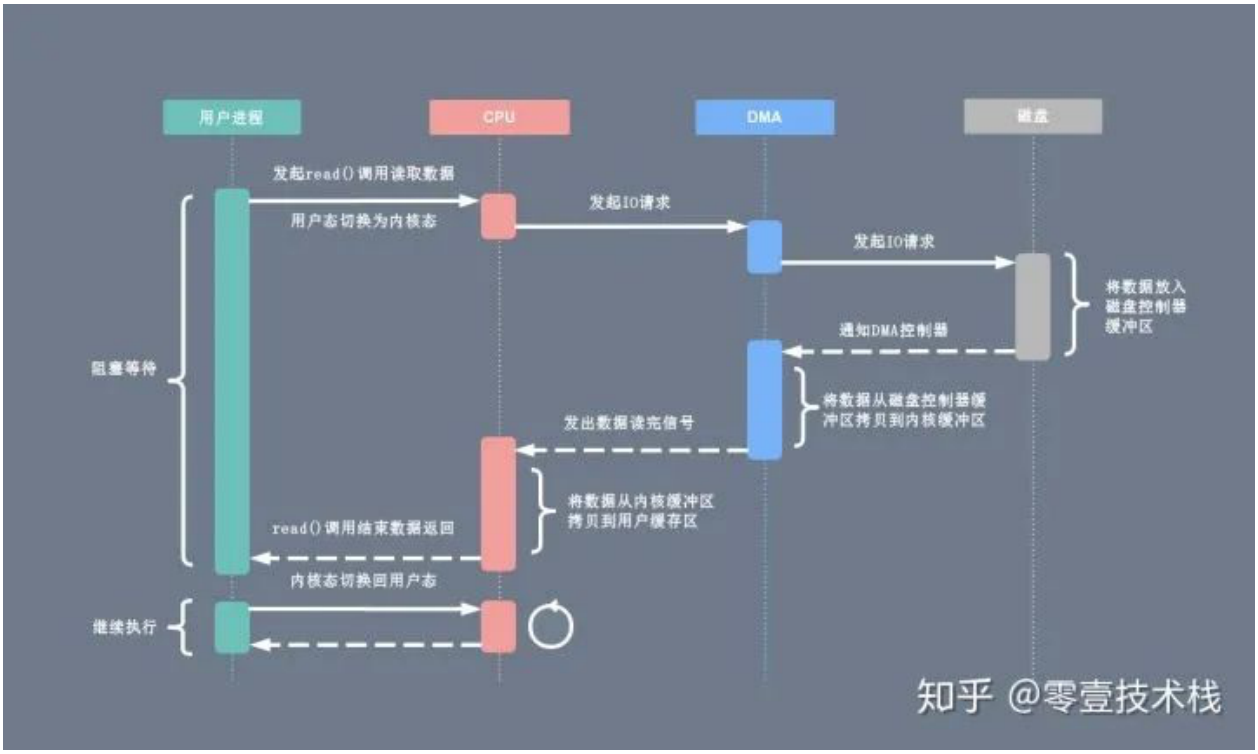
3. 数据准备完成以后，磁盘向 CPU 发起 I/O 中断。

4. CPU 收到 I/O 中断以后将磁盘缓冲区中的数据拷贝到内核缓冲区，然后再从内核缓冲区拷贝到用户缓冲区。

5. 用户进程由内核态切换回用户态，解除阻塞状态，然后等待 CPU 的下一个执行时间钟。

4.2. DMA传输原理

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。整个数据传输操作在一个 DMA 控制器的控制下进行的。CPU 除了在数据传输开始和结束时做一点处理外（开始和结束时候要做中断处理），在传输过程中 CPU 可以继续进行其他的工作。这样在大部分时间里，CPU 计算和 I/O 操作都处于并行操作，使整个计算机系统的效率大大提高。



有了 DMA 磁盘控制器接管数据读写请求以后，CPU 从繁重的 I/O 操作中解脱，数据读取操作的流程如下：

1. 用户进程向 CPU 发起 read 系统调用读取数据，由用户态切换为内核态，然后一直阻塞等待数据的返回。

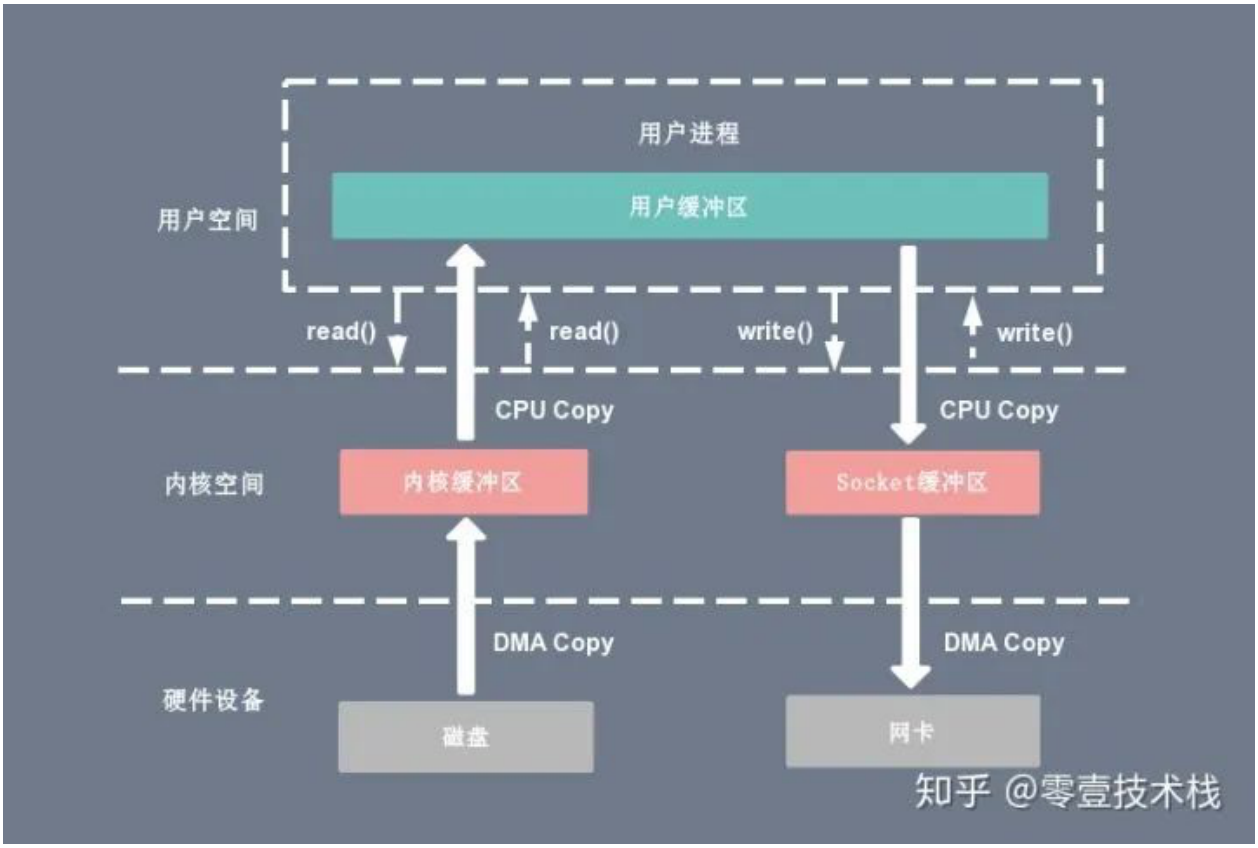
2. CPU 在接收到指令以后对 DMA 磁盘控制器发起调度指令。
3. DMA 磁盘控制器对磁盘发起 I/O 请求，将磁盘数据先放入磁盘控制器缓冲区，CPU 全程不参与此过程。
4. 数据读取完成后，DMA 磁盘控制器会接受到磁盘的通知，将数据从磁盘控制器缓冲区拷贝到内核缓冲区。
5. DMA 磁盘控制器向 CPU 发出数据读完的信号，由 CPU 负责将数据从内核缓冲区拷贝到用户缓冲区。
6. 用户进程由内核态切换回用户态，解除阻塞状态，然后等待 CPU 的下一个执行时间钟。

5. 传统I/O方式

为了更好的理解零拷贝解决的问题，我们首先了解一下传统 I/O 方式存在的问题。在 Linux 系统中，传统的访问方式是通过 write() 和 read() 两个系统调用实现的，通过 read() 函数读取文件到到缓存区中，然后通过 write() 方法把缓存中的数据输出到网络端口，伪代码如下：

```
read(file_fd, tmp_buf, len);
write(socket_fd, tmp_buf, len);
```

下图分别对应传统 I/O 操作的数据读写流程，整个过程涉及 2 次 CPU 拷贝、2 次 DMA 拷贝总共 4 次拷贝，以及 4 次上下文切换，下面简单地阐述一下相关的概念。



- 上下文切换：当用户程序向内核发起系统调用时，CPU 将用户进程从用户态切换到内核态；当系统调用返回时，CPU 将用户进程从内核态切换回用户态。
- CPU拷贝：由 CPU 直接处理数据的传送，数据拷贝时会一直占用 CPU 的资源。
- DMA拷贝：由 CPU 向DMA磁盘控制器下达指令，让 DMA 控制器来处理数据的传送，数据传送完毕再把信息反馈给 CPU，从而减轻了 CPU 资源的占有率。

5.1. 传统读操作

当应用程序执行 read 系统调用读取一块数据的时候，如果这块数据已经存在于用户进程的页内存中，就直接从内存中读取数据；如果数据不存在，则先将数据从磁盘加载数据到内核空间的读缓存（read buffer）中，再从读缓存拷贝到用户进程的页内存中。

```
read(file_fd, tmp_buf, len);
```

基于传统的 I/O 读取方式，read 系统调用会触发 2 次上下文切换，1 次 DMA 拷贝和 1 次 CPU 拷贝，发起数据读取的流程如下：

1. 用户进程通过 read() 函数向内核（kernel）发起系统调用，上下文从用户态（user space）切换为内核态（kernel space）。
2. CPU利用DMA控制器将数据从主存或硬盘拷贝到内核空间（kernel space）的读缓冲区（read buffer）。
3. CPU将读缓冲区（read buffer）中的数据拷贝到用户空间（user space）的用户缓冲区（user buffer）。
4. 上下文从内核态（kernel space）切换回用户态（user space），read 调用执行返回。

5.2. 传统写操作

当应用程序准备好数据，执行 write 系统调用发送网络数据时，先将数据从用户空间的页缓存拷贝到内核空间的网络缓冲区（socket buffer）中，然后再将写缓存中的数据拷贝到网卡设备完成数据发送。

```
write(socket_fd, tmp_buf, len);
```

基于传统的 I/O 写入方式，write() 系统调用会触发 2 次上下文切换，1 次 CPU 拷贝和 1 次 DMA 拷贝，用户程序发送网络数据的流程如下：

1. 用户进程通过 write() 函数向内核（kernel）发起系统调用，上下文从用户态（user space）切换为内核态（kernel space）。

2. CPU 将用户缓冲区（user buffer）中的数据拷贝到内核空间（kernel space）的网络缓冲区（socket buffer）。

3. CPU 利用 DMA 控制器将数据从网络缓冲区（socket buffer）拷贝到网卡进行数据传输。

4. 上下文从内核态（kernel space）切换回用户态（user space），write 系统调用执行返回。

6. 零拷贝方式

在 Linux 中零拷贝技术主要有 3 个实现思路：用户态直接 I/O、减少数据拷贝次数、写时复制技术。

- 用户态直接 I/O：应用程序可以直接访问硬件存储，操作系统内核只是辅助数据传输。这种方式依旧存在用户空间和内核空间的上下文切换，硬件上的数据直接拷贝至了用户空间，不经过内核空间。因此，直接 I/O 不存在内核空间缓冲区和用户空间缓冲区之间的数据拷贝。

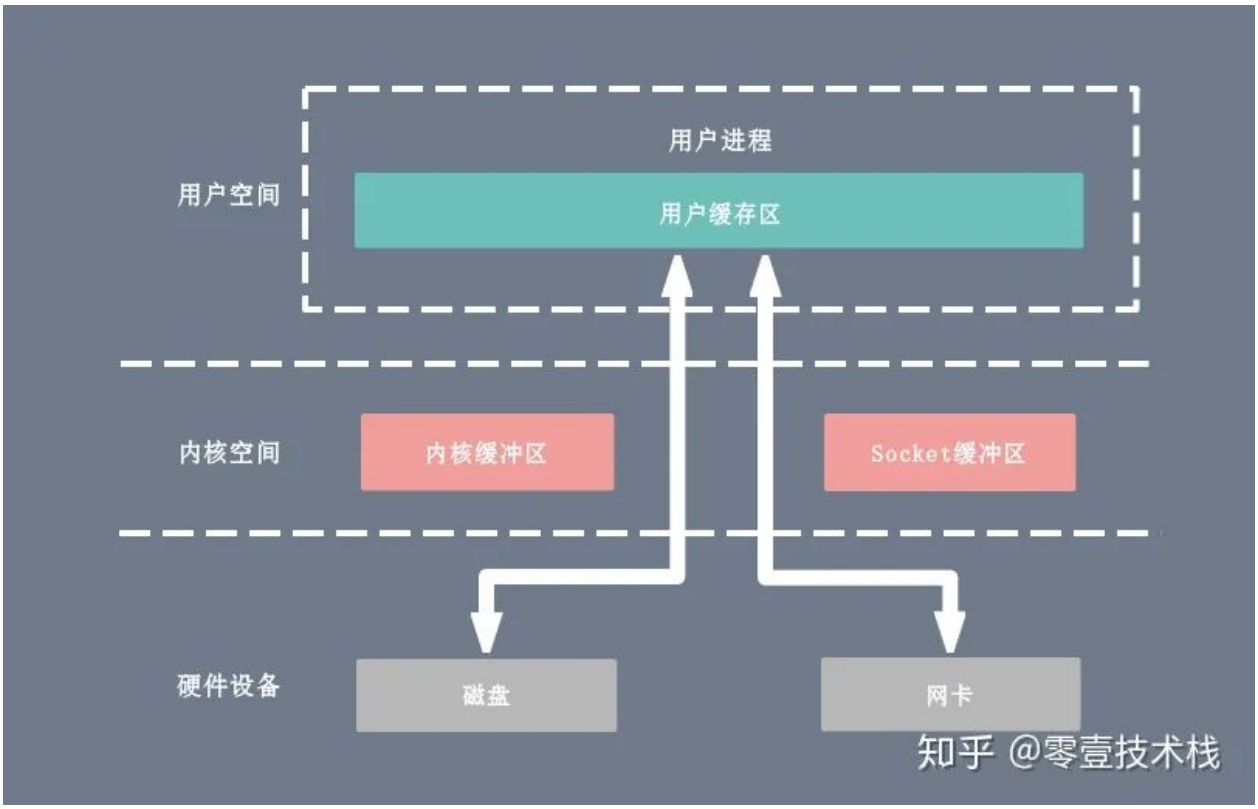
- 减少数据拷贝次数：在数据传输过程中，避免数据在用户空间缓冲区和系统内核空间缓冲区之间的CPU拷贝，以及数据在系统内核空间内的CPU拷贝，这也是当前主流零拷贝技术的实现思路。

- 写时复制技术：写时复制指的是当多个进程共享同一块数据时，如果其中一个进程需要对这份数据进行修改，那么将其拷贝到自己的进程地址空间中，如果只是数据读取操作则不需要进行拷贝操作。

6.1. 用户态直接I/O

用户态直接 I/O 使得应用进程或运行在用户态（user space）下的库函数直接访问硬件设备，数据直接跨过内核进行传输，内核在数据

传输过程除了进行必要的虚拟存储配置工作之外，不参与任何其他工作，这种方式能够直接绕过内核，极大提高了性能。



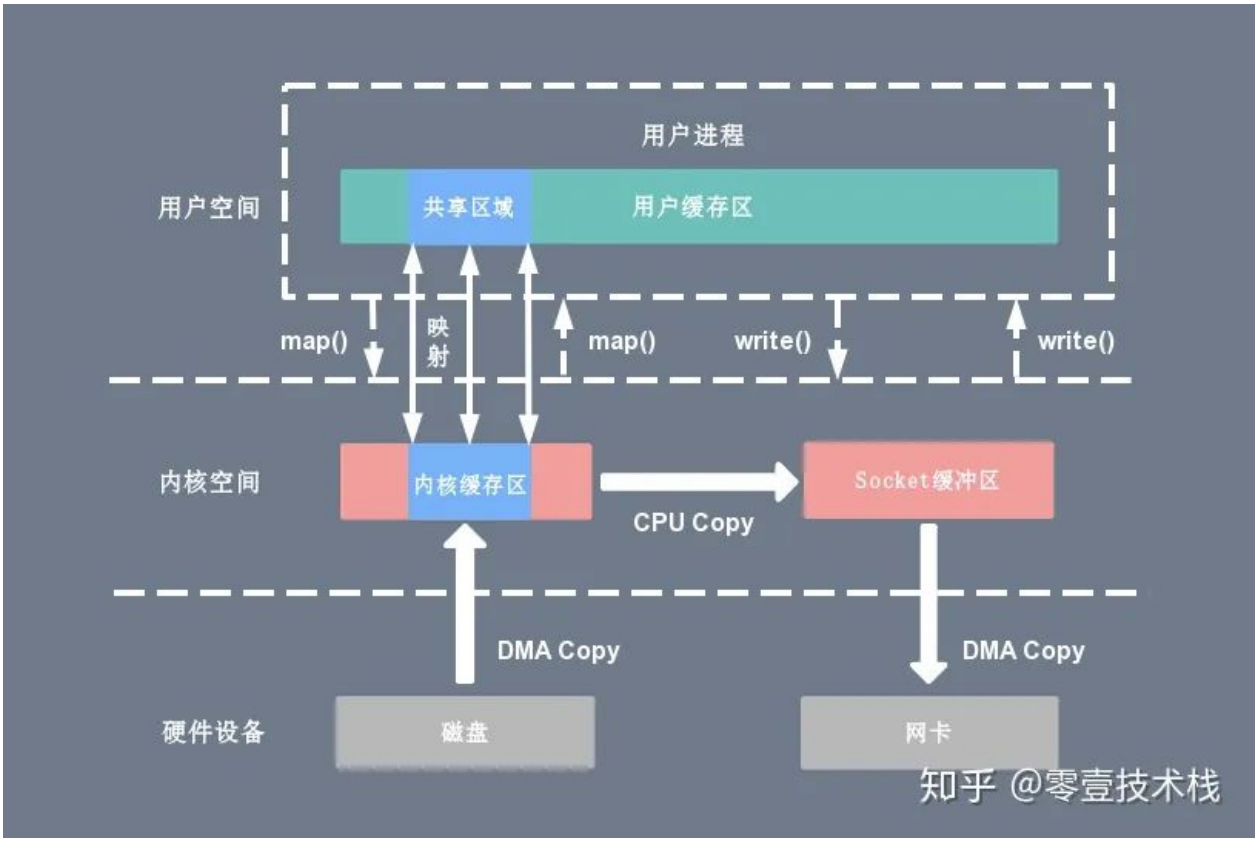
用户态直接 I/O 只能适用于不需要内核缓冲区处理的应用程序，这些应用程序通常在进程地址空间有自己的数据缓存机制，称为自缓存应用程序，如数据库管理系统就是一个代表。其次，这种零拷贝机制会直接操作磁盘 I/O，由于 CPU 和磁盘 I/O 之间的执行时间差距，会造成大量资源的浪费，解决方案是配合异步 I/O 使用。

6.2. 减少数据拷贝次数(mmap + write)

一种零拷贝方式是使用 mmap + write 代替原来的 read + write 方式，减少了 1 次 CPU 拷贝操作。mmap 是 Linux 提供的一种内存映射文件方法，即将一个进程的地址空间中的一段虚拟地址映射到磁盘文件地址，mmap + write 的伪代码如下：

```
tmp_buf = mmap(file_fd, len);
write(socket_fd, tmp_buf, len);
```

使用 mmap 的目的是将内核中读缓冲区（read buffer）的地址与用户空间的缓冲区（user buffer）进行映射，从而实现内核缓冲区与应用程序内存的共享，省去了将数据从内核读缓冲区（read buffer）拷贝到用户缓冲区（user buffer）的过程，然而内核读缓冲区（read buffer）仍需将数据拷贝到内核写缓冲区（socket buffer），大致的流程如下图所示：



基于 mmap + write 系统调用的零拷贝方式，整个拷贝过程会发生 4 次上下文切换，1 次 CPU 拷贝和 2 次 DMA 拷贝，用户程序读写数据的流程如下：

1. 用户进程通过 mmap() 函数向内核（kernel）发起系统调用，上下文从用户态（user space）切换为内核态（kernel space）。

2. 将用户进程的内核空间的读缓冲区（read buffer）与用户空间的缓存区（user buffer）进行内存地址映射。

3. CPU利用DMA控制器将数据从主存或硬盘拷贝到内核空间（kernel space）的读缓冲区（read buffer）。

4. 上下文从内核态（kernel space）切换回用户态（user space），mmap 系统调用执行返回。

5. 用户进程通过 write() 函数向内核（kernel）发起系统调用，上下文从用户态（user space）切换为内核态（kernel space）。

6. CPU将读缓冲区（read buffer）中的数据拷贝到的网络缓冲区（socket buffer）。

7. CPU利用DMA控制器将数据从网络缓冲区（socket buffer）拷贝到网卡进行数据传输。

8. 上下文从内核态（kernel space）切换回用户态（user space），write 系统调用执行返回。

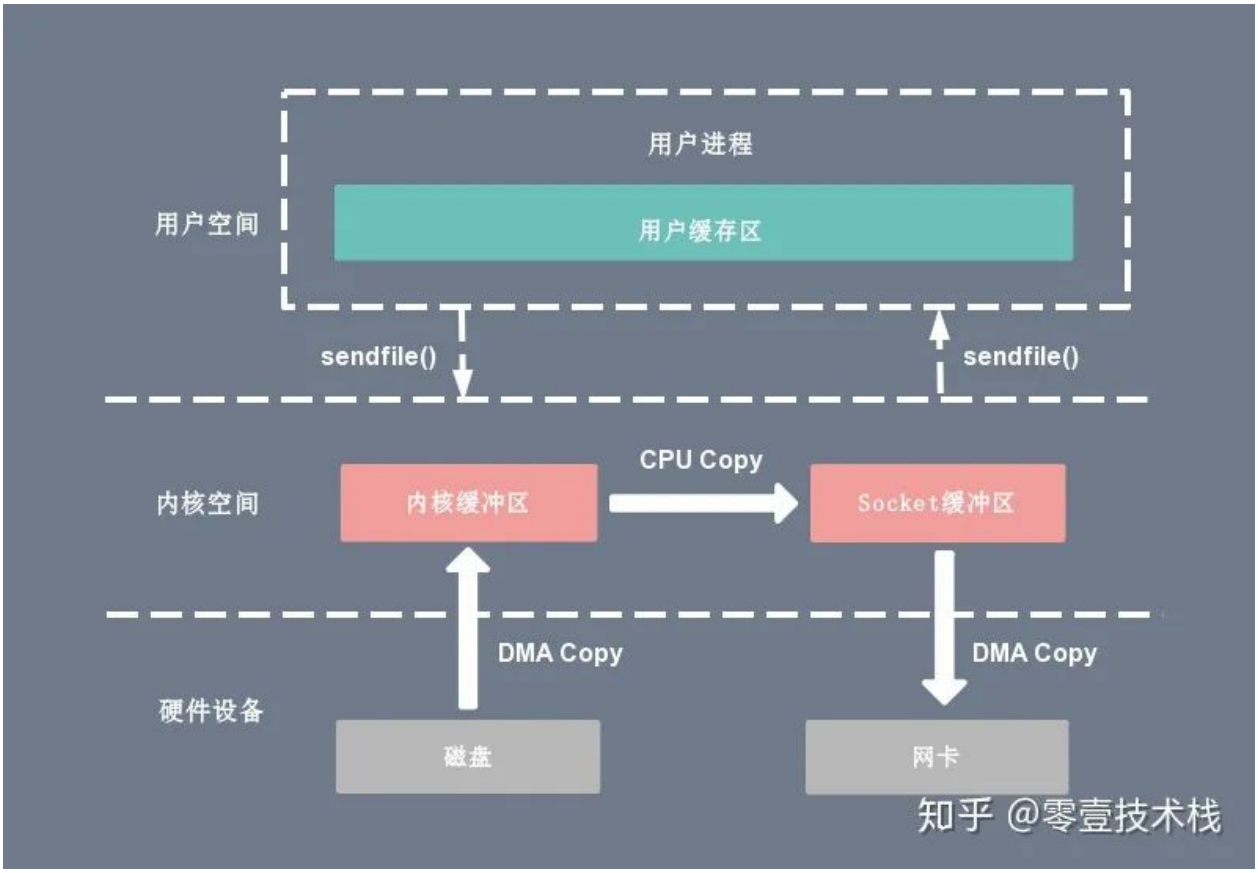
mmap 主要的用处是提高 I/O 性能，特别是针对大文件。对于小文件，内存映射文件反而会导致碎片空间的浪费，因为内存映射总是要对齐页边界，最小单位是 4 KB，一个 5 KB 的文件将会映射占用 8 KB 内存，也就会浪费 3 KB 内存。mmap 的拷贝虽然减少了 1 次CPU拷贝，提升了效率，但也存在一些隐藏的问题。当 mmap 一个文件时，如果这个文件被另一个进程所截获，那么 write 系统调用会因为访问非法地址被 SIGBUS 信号终止，SIGBUS 默认会杀死进程并产生一个 coredump，服务器可能因此被终止。

6.3. 减少数据拷贝次数(sendfile)

sendfile 系统调用在 Linux 内核版本 2.1 中被引入，目的是简化通过网络在两个通道之间进行的数据传输过程。sendfile 系统调用的引入，不仅减少了 CPU 拷贝的次数，还减少了上下文切换的次数，它的伪代码如下：

```
sendfile(socket_fd, file_fd, len);
```

通过 sendfile 系统调用，数据可以直接在内核空间内部进行 I/O 传输，从而省去了数据在用户空间和内核空间之间的来回拷贝。与 mmap 内存映射方式不同的是， sendfile 调用中 I/O 数据对用户空间是完全不可见的。也就是说，这是一次完全意义上的数据传输过程。



基于 sendfile 系统调用的零拷贝方式，整个拷贝过程会发生 2 次上下文切换，1 次 CPU 拷贝和 2 次 DMA 拷贝，用户程序读写数据的流程如下：

1. 用户进程通过 sendfile() 函数向内核（kernel）发起系统调用，上下文从用户态（user space）切换为内核态（kernel space）。
2. CPU 利用 DMA 控制器将数据从主存或硬盘拷贝到内核空间（kernel space）的读缓冲区（read buffer）。
3. CPU 将读缓冲区（read buffer）中的数据拷贝到的网络缓冲区

（socket buffer）。

4. CPU 利用 DMA 控制器将数据从网络缓冲区（socket buffer）拷贝到网卡进行数据传输。

5. 上下文从内核态（kernel space）切换回用户态（user space），sendfile 系统调用执行返回。

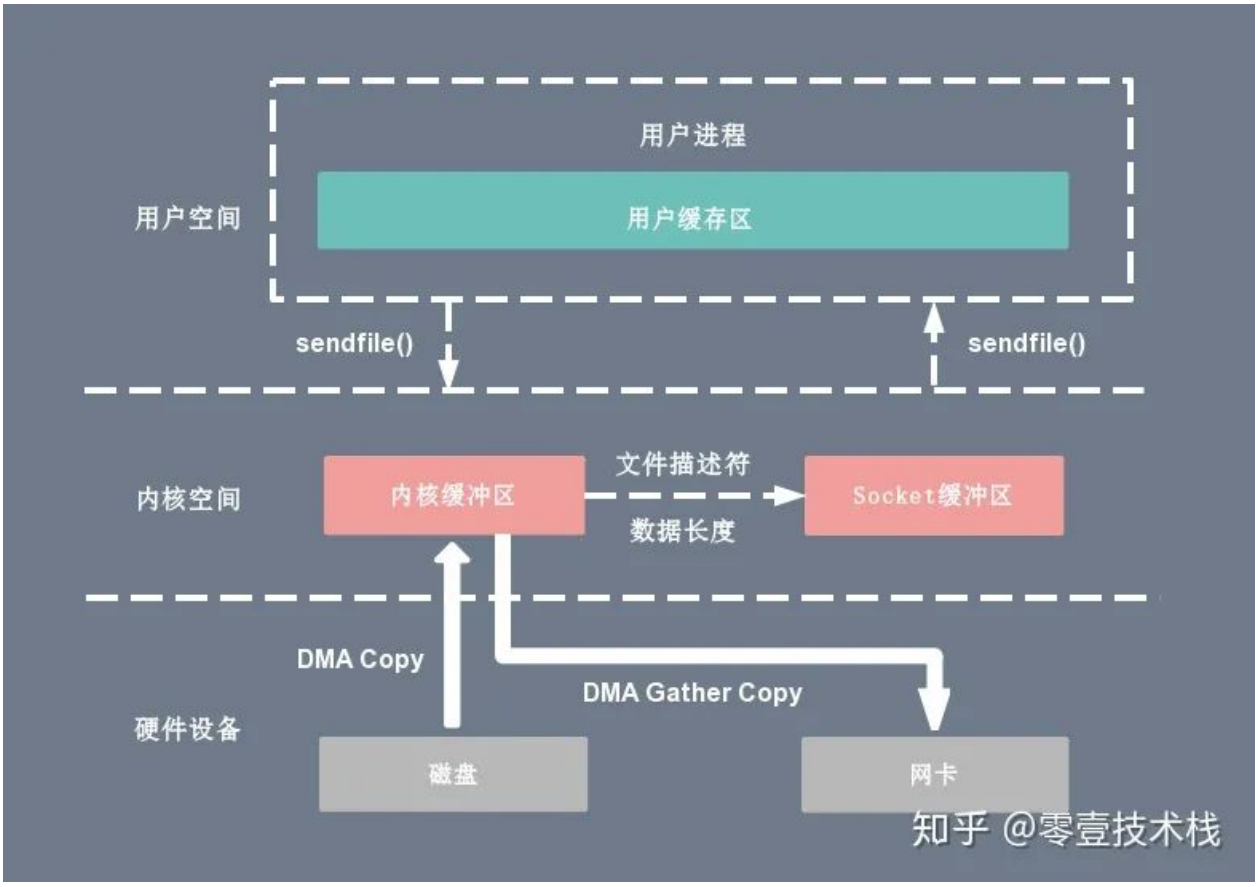
相比较于 mmap 内存映射的方式，sendfile 少了 2 次上下文切换，但是仍然有 1 次 CPU 拷贝操作。sendfile 存在的问题是用户程序不能对数据进行修改，而只是单纯地完成了一次数据传输过程。

6.4. 减少数据拷贝次数(sendfile + DMA gather copy)

Linux 2.4 版本的内核对 sendfile 系统调用进行修改，为 DMA 拷贝引入了 gather 操作。它将内核空间（kernel space）的读缓冲区（read buffer）中对应的数据描述信息（内存地址、地址偏移量）记录到相应的网络缓冲区（ socket buffer）中，由 DMA 根据内存地址、地址偏移量将数据批量地从读缓冲区（read buffer）拷贝到网卡设备中，这样就省去了内核空间中仅剩的 1 次 CPU 拷贝操作，sendfile 的伪代码如下：

```
sendfile(socket_fd, file_fd, len);
```

在硬件的支持下，sendfile 拷贝方式不再从内核缓冲区的数据拷贝到 socket 缓冲区，取而代之的仅仅是缓冲区文件描述符和数据长度的拷贝，这样 DMA 引擎直接利用 gather 操作将页缓存中数据打包发送到网络中即可，本质就是和虚拟内存映射的思路类似。



基于 sendfile + DMA gather copy 系统调用的零拷贝方式，整个拷贝过程会发生 2 次上下文切换、0 次 CPU 拷贝以及 2 次 DMA 拷贝，用户程序读写数据的流程如下：

1. 用户进程通过 sendfile() 函数向内核（kernel）发起系统调用，上下文从用户态（user space）切换为内核态（kernel space）。
2. CPU 利用 DMA 控制器将数据从主存或硬盘拷贝到内核空间（kernel space）的读缓冲区（read buffer）。
3. CPU 把读缓冲区（read buffer）的文件描述符（file descriptor）和数据长度拷贝到网络缓冲区（socket buffer）。
4. 基于已拷贝的文件描述符（file descriptor）和数据长度，CPU 利用 DMA 控制器的 gather/scatter 操作直接批量地将数据从内核的读缓冲区（read buffer）拷贝到网卡进行数据传输。

5. 上下文从内核态（kernel space）切换回用户态（user space），sendfile 系统调用执行返回。

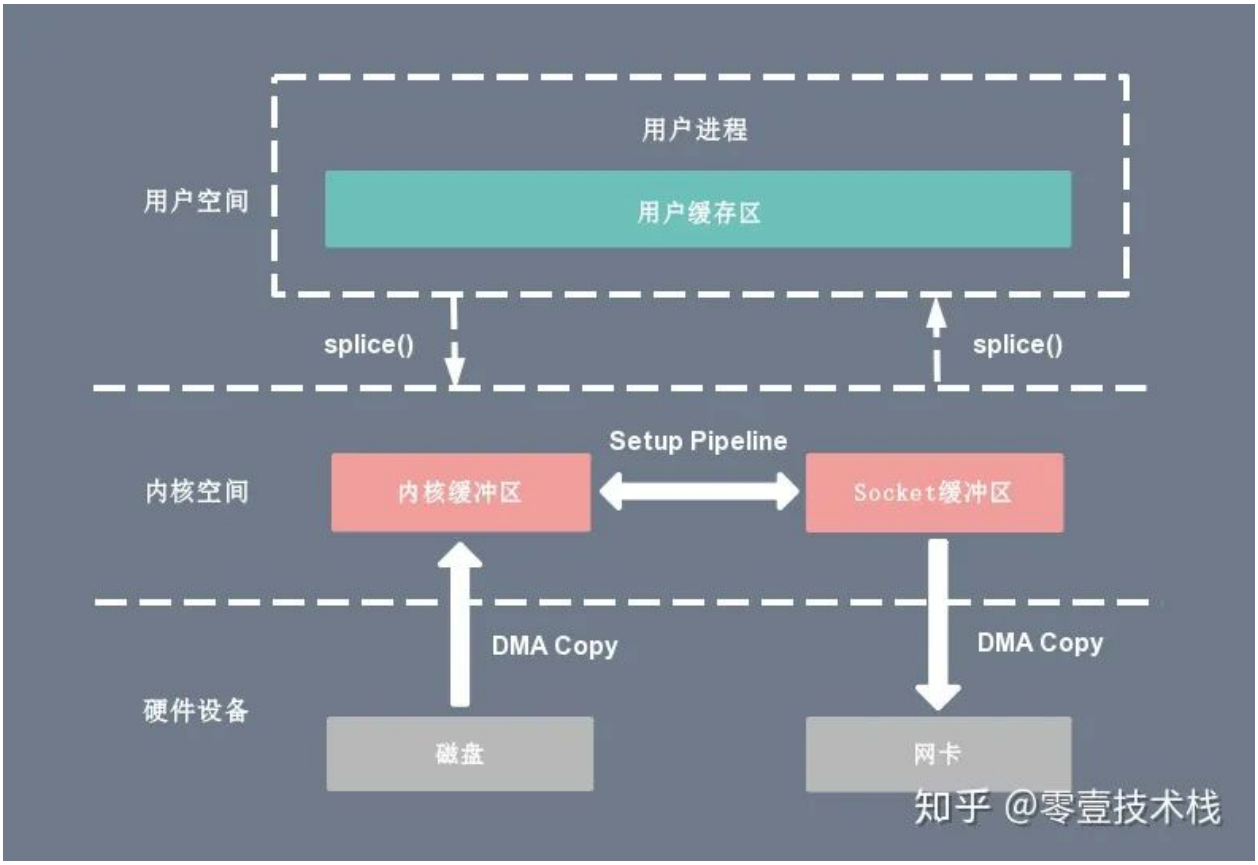
sendfile + DMA gather copy 拷贝方式同样存在用户程序不能对数据进行修改的问题，而且本身需要硬件的支持，它只适用于将数据从文件拷贝到 socket 套接字上的传输过程。

6.5. 减少数据拷贝次数(splice)

sendfile 只适用于将数据从文件拷贝到 socket 套接字上，同时需要硬件的支持，这也限定了它的使用范围。Linux 在 2.6.17 版本引入 splice 系统调用，不仅不需要硬件支持，还实现了两个文件描述符之间的数据零拷贝。splice 的伪代码如下：

```
splice(fd_in, off_in, fd_out, off_out, len, flags);
```

splice 系统调用可以在内核空间的读缓冲区（read buffer）和网络缓冲区（socket buffer）之间建立管道（pipeline），从而避免了两者之间的 CPU 拷贝操作。



基于 splice 系统调用的零拷贝方式，整个拷贝过程会发生 2 次上下文切换，0 次 CPU 拷贝以及 2 次 DMA 拷贝，用户程序读写数据的流程如下：

1. 用户进程通过 splice() 函数向内核（kernel）发起系统调用，上下文从用户态（user space）切换为内核态（kernel space）。
2. CPU 利用 DMA 控制器将数据从主存或硬盘拷贝到内核空间（kernel space）的读缓冲区（read buffer）。
3. CPU 在内核空间的读缓冲区（read buffer）和网络缓冲区（socket buffer）之间建立管道（pipeline）。
4. CPU 利用 DMA 控制器将数据从网络缓冲区（socket buffer）拷贝到网卡进行数据传输。
5. 上下文从内核态（kernel space）切换回用户态（user space），splice 系统调用执行返回。

splice 拷贝方式也同样存在用户程序不能对数据进行修改的问题。除此之外，它使用了 Linux 的管道缓冲机制，可以用于任意两个文件描述符中传输数据，但是它的两个文件描述符参数中有一个必须是管道设备。

7. Linux零拷贝对比

无论是传统 I/O 拷贝方式还是引入零拷贝的方式，2 次 DMA Copy 是都少不了的，因为两次 DMA 都是依赖硬件完成的。下面从 CPU 拷贝次数、DMA 拷贝次数以及系统调用几个方面总结一下上述几种 I/O 拷贝方式的差别。

拷贝方式	CPU拷贝	DMA拷贝	系统调用	上下文切换
传统方式（read + write）	2	2	read / write	4
内存映射（mmap + write）	1	2	mmap / write	4
sendfile	1	2	sendfile	2
sendfile + DMA gather copy	0	2	sendfile	2
splice	0	2	splice	2

8. 其它的零拷贝实现

8.1. RocketMQ和Kafka对比

RocketMQ 选择了 mmap + write 这种零拷贝方式，适用于业务级消息这种小块文件的数据持久化和传输；而 Kafka 采用的是 sendfile 这种零拷贝方式，适用于系统日志消息这种高吞吐量的大块文件的数据持久化和传输。但是值得注意的一点是，Kafka 的索引文件使用的是 mmap + write 方式，数据文件使用的是 sendfile 方式。

消息队列	零拷贝方式	优点	缺点
RocketMQ	mmap + write	适用于小块文件传输，频繁调用时，效率很高	不能很好的利用 DMA 方式，会比 sendfile 多消耗 CPU，内存安全性控制复杂，需要避免 JVM Crash 问题
Kafka	sendfile	可以利用 DMA 方式，消耗 CPU 较少，大块文件传输效率高，无内存安全性问题	小块文件效率低于 mmap 方式，只能是 BIO 方式传输，不能使用 NIO 方式

参考：

https://zhuanlan.zhihu.com/p/83398714