Socket：网络中不同主机的应用进程进行双向通信的端点的抽象。

<https://www.jianshu.com/p/397449cadc9a> IO多路复用的三种机制Select，Poll，Epoll

# 一、linux mmap和零拷贝

## 1. 传统IO的劣势

File file = new File("index.html");

RandomAccessFile raf = new RandomAccessFile(file, "rw");

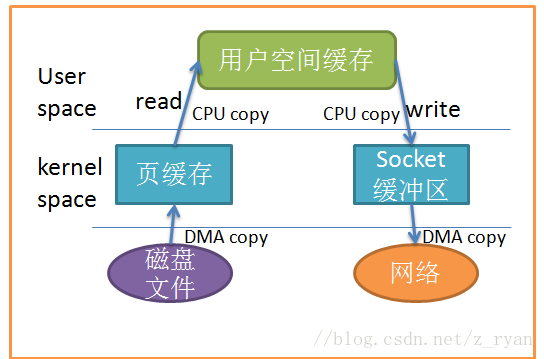
byte[] arr = new byte[(int) file.length()];

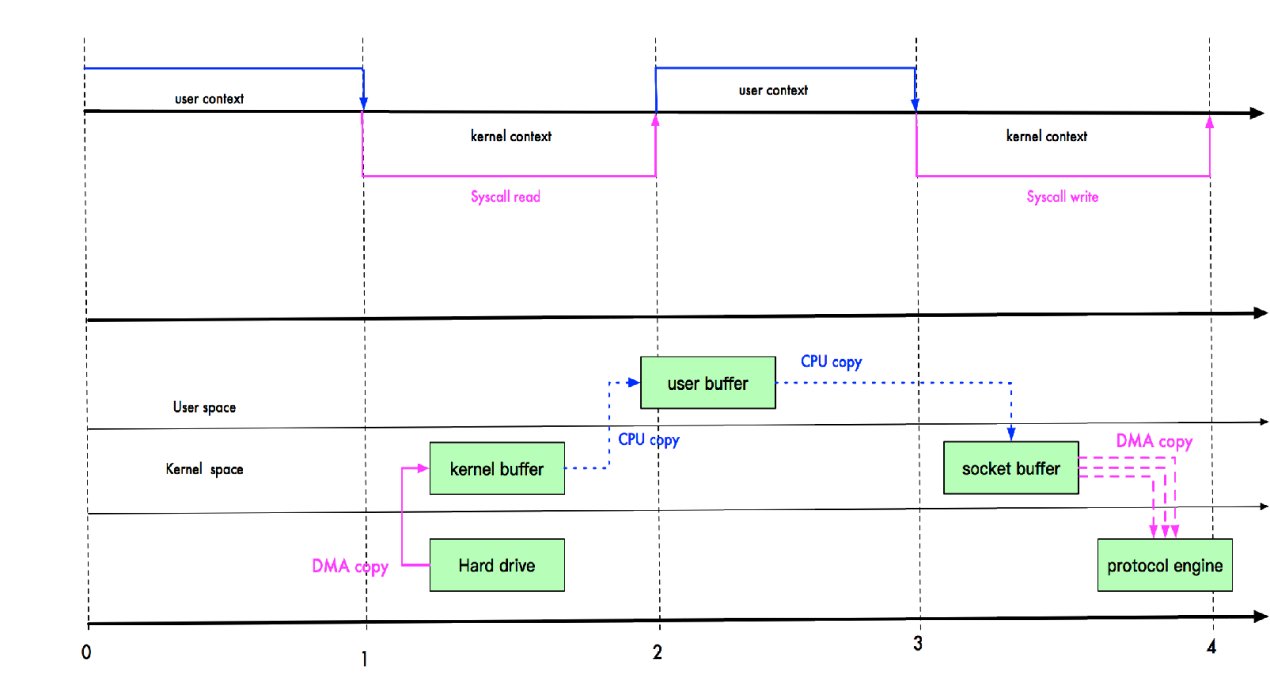
raf.read(arr);

Socket socket = new ServerSocket(8080).accept();

socket.getOutputStream().write(arr);

调用read方法读取index.html的内容，然后调用write方法将index.html字节流写到socket缓存中。





上图中，上半部分表示用户态和内核态的上下文切换。下半部分表示数据复制操作。下面说说他们的步骤：

1. read 调用导致用户态到内核态的一次变化，同时，第一次复制开始：DMA（Direct Memory Access，直接内存存取，即不使用 CPU 拷贝数据到内存，而是 DMA 引擎传输数据到内存，用于解放 CPU） 引擎从磁盘读取 index.html 文件，并将数据放入到内核缓冲区。

2. 发生第二次数据拷贝，即：将内核缓冲区的数据拷贝到用户缓冲区，同时，发生了一次用内核态到用户态的上下文切换。

3. 发生第三次数据拷贝，我们调用 write 方法，系统将用户缓冲区的数据拷贝到 Socket 缓冲区。此时，又发生了一次用户态到内核态的上下文切换。

4. 第四次拷贝，数据异步的从 Socket 缓冲区，使用 DMA 引擎拷贝到网络协议引擎。这一段，不需要进行上下文切换。

5. write 方法返回，再次从内核态切换到用户态。

**4次上下文切换，4次拷贝**

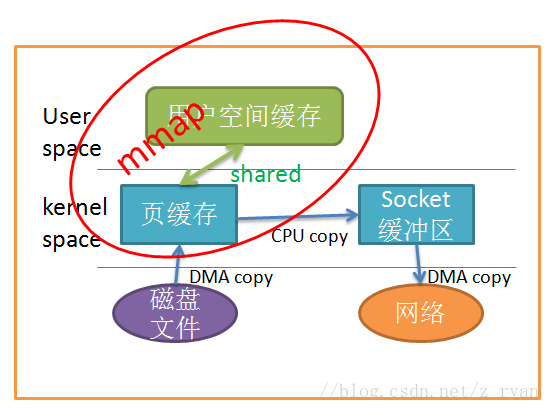
## 2. mmap优化

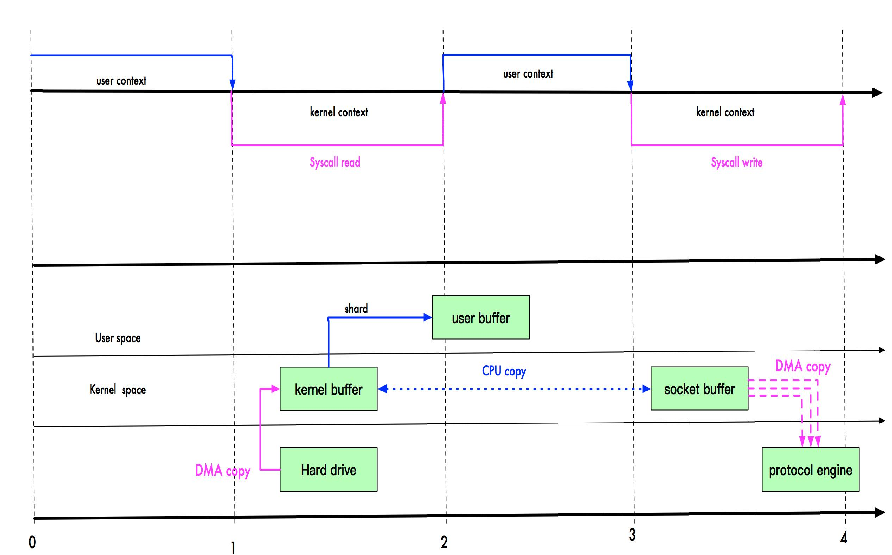
memery mapping（内存映射）

 mmap 通过内存映射，将文件映射到内核缓冲区，同时，用户空间可以共享内核空间的数据

用户态可以直接读取内核态缓存，减少了传统io中第一次CPU拷贝（无需将内核态数据拷贝到用户态缓存中）。

**4次上下文切换，3次拷贝**



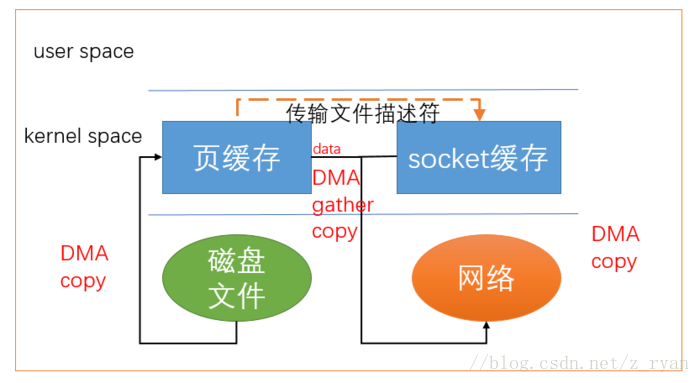


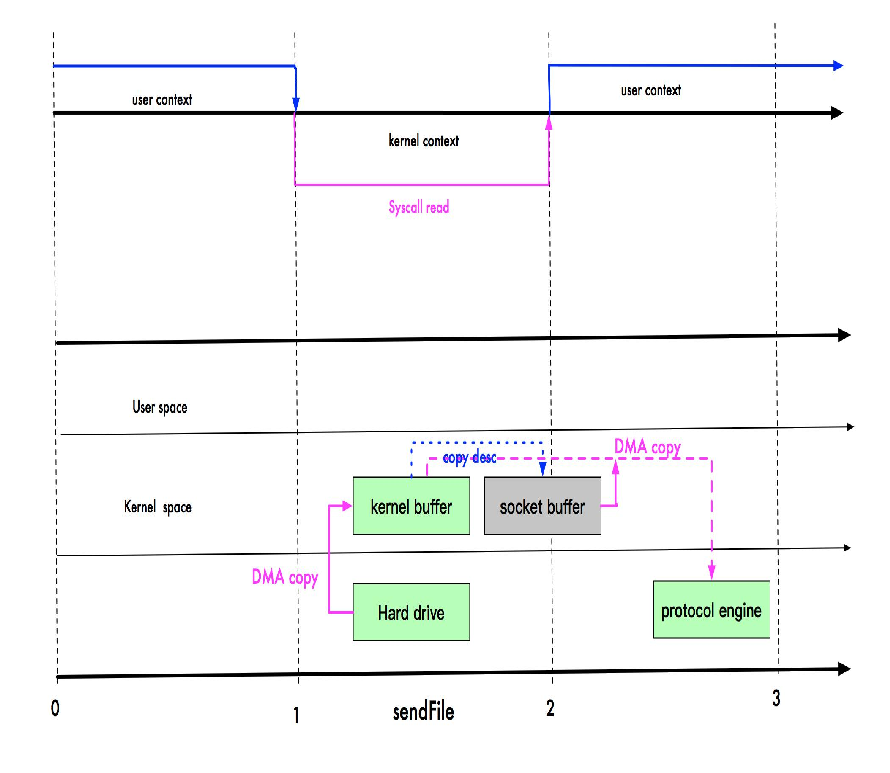
## 3. sendFile

数据不经过用户态，直接从内核态缓冲区拷贝到网络协议栈

2次拷贝，2次上下文切换

数据拷贝工作全部在内核态完成，且没有CPU拷贝，全部是DMA拷贝，所以又称零拷贝





## 4. mmap和sendFile的区别

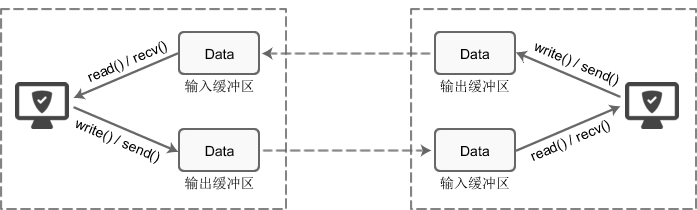
1. mmap 适合小数据量读写，sendFile 适合大文件传输。
2. mmap 需要 4 次上下文切换，3 次数据拷贝；sendFile 需要 3 次上下文切换，最少 2 次数据拷贝。
3. sendFile 可以利用 DMA 方式，减少 CPU 拷贝，mmap 则不能（必须从内核拷贝到 Socket 缓冲区）。

## 5. java中应用

transferTo 和 transferFrom 方法，即对应 Linux 的 sendFile

# 二、Socket缓冲区

每个 socket 被创建后，都会分配两个缓冲区，输入缓冲区和输出缓冲区。  
  
write()/send() 并不立即向网络中传输数据，而是先将数据写入缓冲区中，再由TCP协议将数据从缓冲区发送到目标机器。一旦将数据写入到缓冲区，函数就可以成功返回，不管它们有没有到达目标机器，也不管它们何时被发送到网络，这些都是TCP协议负责的事情。  
  
TCP协议独立于 write()/send() 函数，数据有可能刚被写入缓冲区就发送到网络，也可能在缓冲区中不断积压，多次写入的数据被一次性发送到网络，这取决于当时的网络情况、当前线程是否空闲等诸多因素，不由程序员控制。  
  
read()/recv() 函数也是如此，也从输入缓冲区中读取数据，而不是直接从网络中读取。



# 三、IO模型

在讨论之前先说明一下IO发生时涉及到的对象和步骤，对于一个network IO，它会涉及到两个系统对象：

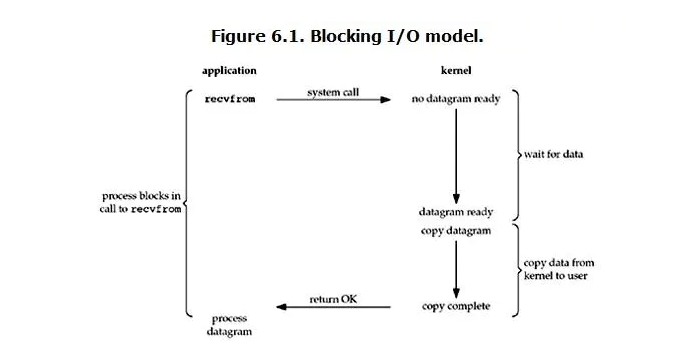
* **application** 调用这个IO的进程
* **kernel** 系统内核

那他们经历的两个交互过程是：

* **阶段1 wait for data** 等待数据准备（从磁盘将数据复制到kernel buffer中）
* **阶段2 copy data from kernel to user** 将数据从内核拷贝到用户进程中

之所以会有同步、异步、阻塞和非阻塞这几种说法就是根据程序在这两个阶段的处理方式不同而产生的。了解了这些背景之后，我们就分别针对四种IO模型进行讲解

## 1. Blocking IO – 阻塞IO

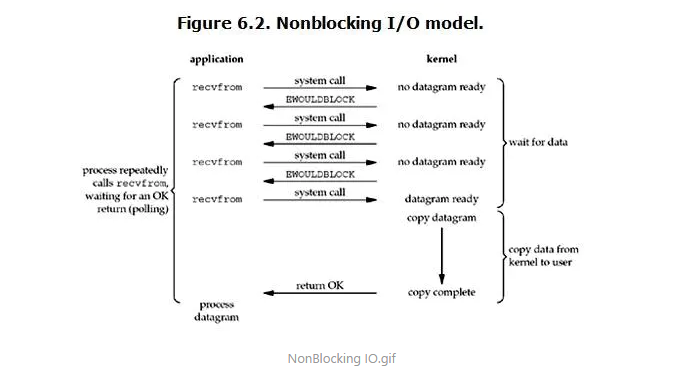


1. 当用户进程调用了recvfrom这个系统调用，kernel就开始了IO的第一个阶段：准备数据。而在用户进程这边，整个进程会被阻塞。

2. 当kernel一直等到数据准备好了，它就会将数据从kernel中拷贝到用户内存，然后kernel返回结果，用户进程才解除block的状态，重新运行起来。

所以，blocking IO的特点就是在IO执行的两个阶段都被block了。

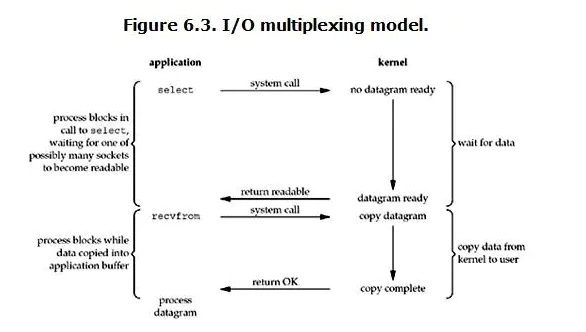
## 2. NoneBlocking IO 非阻塞IO



从图中可以看出，当用户进程发出recvfrom这个系统调用后，如果kernel中的数据还没有准备好，那么它并不会block用户进程，而是立刻返回一个结果（no datagram ready）。从用户进程角度讲 ，它发起一个操作后，并没有等待，而是马上就得到了一个结果。用户进程得知数据还没有准备好后，它可以每隔一段时间再次发送recvfrom操作。一旦kernel中的数据准备好了，并且又再次收到了用户进程的system call，那么它马上就将数据拷贝到了用户内存，然后返回。

所以，用户进程其实是需要不断的主动询问kernel数据好了没有。

## 3. IO multiplexing –IO多路复用

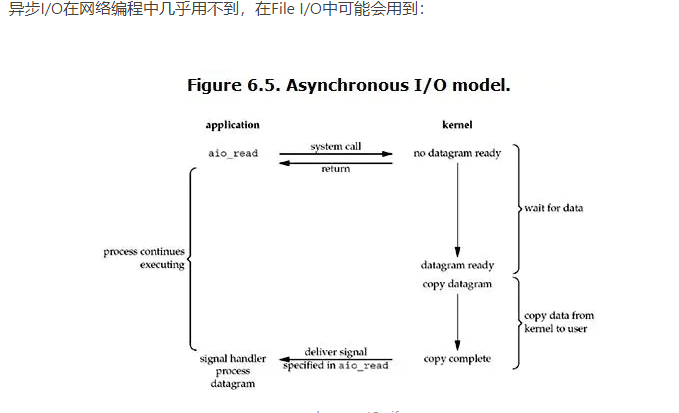


看起来它与blocking I/O很相似，两个阶段都阻塞。但它与blocking I/O的一个重要区别就是它可以等待多个数据报就绪（datagram ready），即可以处理多个连接。这里的select相当于一个“代理”，调用select以后进程会被select阻塞，这时候在内核空间内select会监听指定的多个datagram (如socket连接)，如果其中任意一个数据就绪了就返回。此时程序再进行数据读取操作，将数据拷贝至当前进程内。由于select可以监听多个socket，我们可以用它来处理多个连接。

在select模型中每个socket一般都设置成non-blocking，虽然等待数据阶段仍然是阻塞状态，但是它是被select调用阻塞的，而不是直接被I/O阻塞的。select底层通过轮询机制来判断每个socket读写是否就绪。

当然select也有一些缺点，比如底层轮询机制会增加开销、支持的文件描述符数量过少等。为此，Linux引入了epoll作为select的改进版本。

## 4. asynchronous IO – 异步IO



这里面的读取操作的语义与上面的几种模型都不同。这里的读取操作(aio\_read)会通知内核进行读取操作并将数据拷贝至进程中，完事后通知进程整个操作全部完成（绑定一个回调函数处理数据）。读取操作会立刻返回，程序可以进行其它的操作，所有的读取、拷贝工作都由内核去做，做完以后通知进程，进程调用绑定的回调函数来处理数据。

## 总结

我们来总结一下阻塞、非阻塞，同步和异步这两组概念。

先来说阻塞和非阻塞：

* 阻塞调用会一直等待远程数据就绪再返回，即上面的**阶段1**会阻塞调用者，直到读取结束。
* 而非阻塞无论在什么情况下都会立即返回，虽然非阻塞大部分时间不会被block，但是它仍要求进程不断地去主动询问kernel是否准备好数据，也需要进程主动地再次调用recvfrom来将数据拷贝到用户内存。

再说一说同步和异步：

* 同步方法会一直阻塞进程，直到I/O操作结束，注意这里相当于上面的**阶段1，阶段2**都会阻塞调用者。其中 Blocking IO - 阻塞IO，Nonblocking IO - 非阻塞IO，IO multiplexing - IO多路复用，signal driven IO - 信号驱动IO 这四种IO都可以归类为同步IO。
* 而异步方法不会阻塞调用者进程，即使是从内核空间的缓冲区将数据拷贝到进程中这一操作也不会阻塞进程，拷贝完毕后内核会通知进程数据拷贝结束。