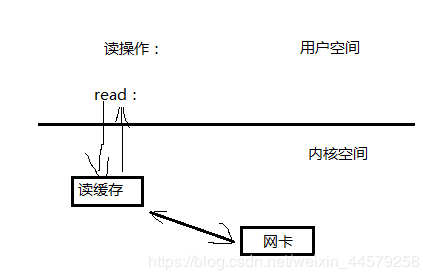
**JAVA五种IO模型**

## 网络中获取数据的读操作步骤：



**同步与异步：**

同步与异步是针对应用程序与内核的交互而言。也就是上图的read操作，从缓存中读取数据，如果缓存中换没有准备好，如果是同步操作，它会一直等待，知道操作完成，如果是异步操作，那么他会去做别的事情，等待数据准备好，内核通知它，它再去读取数据

同步过程中进程触发IO操作并等待或者轮询的去查看IO操作是否完成。

异步过程中进程触发IO操作以后，直接返回，做自己的事情，IO操作交给内核来处理， 处理 完后通知进程IO完成

同步和异步是相对于操作结果来说，会不会等待结果返回

**阻塞与非阻塞：**

应用进程请求IO操作时，如果数据未准备好，立即返回就是非阻塞，不立即返回就是阻塞。简单来说，就是做一件事如果不能立即获得返回，需要等待，就是阻塞，否则就是非阻塞。

阻塞与非阻塞是相对于线程是否被阻塞

## 异步/同步/ 阻塞/非阻塞 的区别

两者存在本质的区别，他们的修饰对象是不同的。

阻塞和非阻塞是指进程访问的数据如果尚未准备就绪，进程是否需要等待，简单说：相等于函数内部的实现区别，就是未就绪时，直接返回还是等待就绪 。

同步和异步是指访问数据的机制，同步一般主动请求等待IO操作完毕的方式。当数据就绪后，再读写的时候必须阻塞，异步则主动请求数据后便可以继续处理其他任务，随后等待IO完毕通知，这可以使进程在数据读写时也不阻塞

**异步/同步 阻塞/非阻塞的组合方式**

**同步阻塞**：效率最低，实际程序中，就是fd未设置O\_NONBLOCK标志位的read/write操作。

老王用水壶烧水，并且站在那里，看着水壶，等水开。

**异步阻塞**：异步操作是可以阻塞住的，只不过它不是在处理消息是阻塞，而是在等到消息时阻塞，

老王用响水壶烧水，站在那里，这次不看水壶了，而是等水开了水壶自动通知发出声音，老王听见了，知道水开了

**同步非阻塞**：实际效率 还是低下的，注意fd设置O\_NONBLOCK标志位

老王用水壶烧水，不在站在那里直接等，而是跑出去去干别的事情，比如打游戏，看电视，但是，老王心里不放心，每隔一段时间回来看一下，看水开了没（异步内核通知进程）

**异步非阻塞**：效率高效，注册一个回调函数，就可以去做别的事情，

老王用响水壶烧水，跑去做别的事情，等待响水壶发出声音。（异步内核通知进程）

**socket的fd是什么？**

fd是(file descriptor/文件描述符) ，这种一般是BSD Socket的用法，用在Unix/linux系统上。在Unix/linux系统下，一个Socket句柄，可以看做是一个文件，在socket上收发数据，相当于读一个文件进行读写，所以一个socket句柄，通常也用表示文件句柄的fd来表示。

**缓存IO**

缓存IO又被称为标准IO，大多数文件系统的默认IO都是缓存IO，在linux的缓存IO机制中，造作系统会将IO的数据缓存在文件系统的页缓存（page cache），也就是说，数据会先被拷贝到操作系统内核的缓冲区中，然后才会从操作系统内核的缓冲拷贝到应用程序的地址空间

缓存IO的缺点：

数据在传输过程中需要在应用程序地址空间和内核进行多次数据拷贝操作，这些数据拷贝带来的cpu以及内存开销是非常大的。

## unix提供了5种IO模型

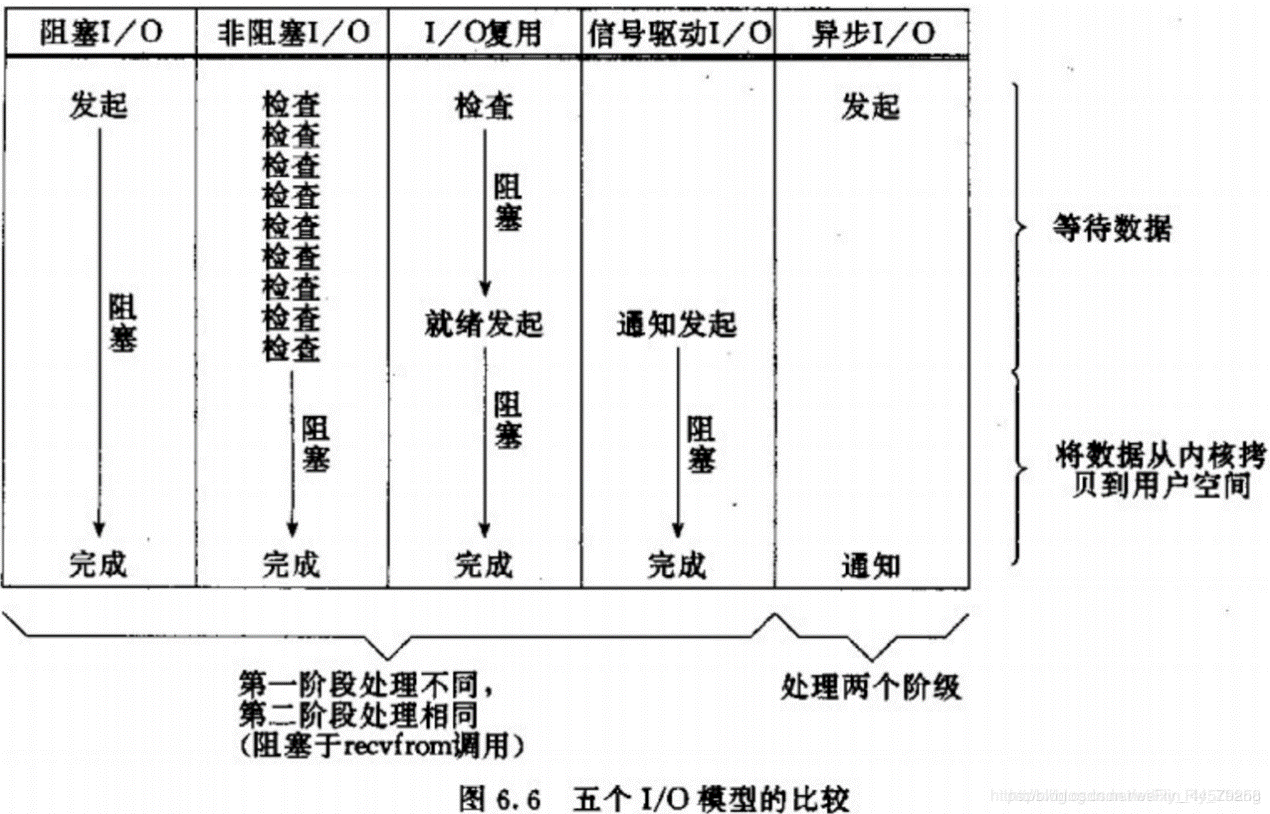
1、阻塞IO

2、非阻塞IO

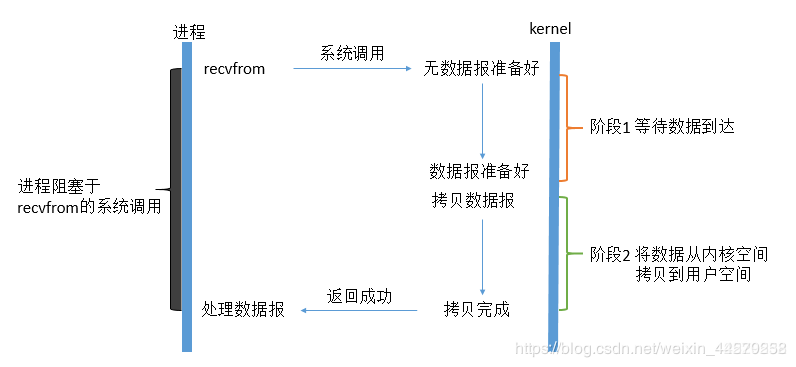
3、IO多路复用

4、信号量

5、异步IO

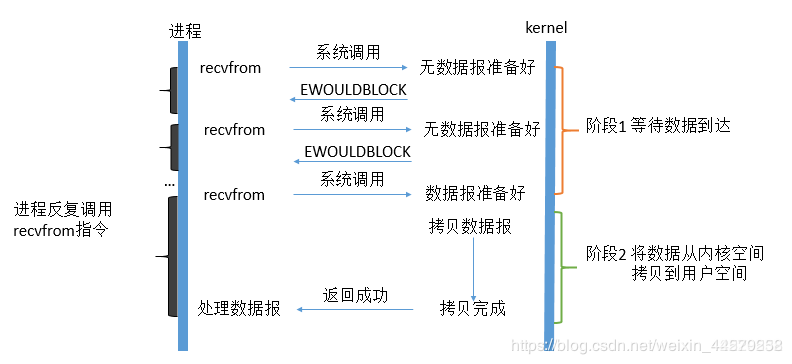


## 阻塞IO（BIO）



recvfrom函数为系统调用函数，从图中可以看出，从进行系统调用到拷贝数据到应用进程缓冲区完成，整段时间都是被阻塞的，在这个过程中要么正确到达，要么系统调用被打断；直到数据报被拷贝到用户进程后，用户才接触阻塞状态，这里的用户进程是自己进行阻塞，拷贝也是有用户进行完成。在等待数据待处理数据的两个阶段，整个进程都是被阻塞的，不能处理别的网络IO，调用应用程序处于一种不在消费CPU而只是简单等待响应的状态。

## 非阻塞IO

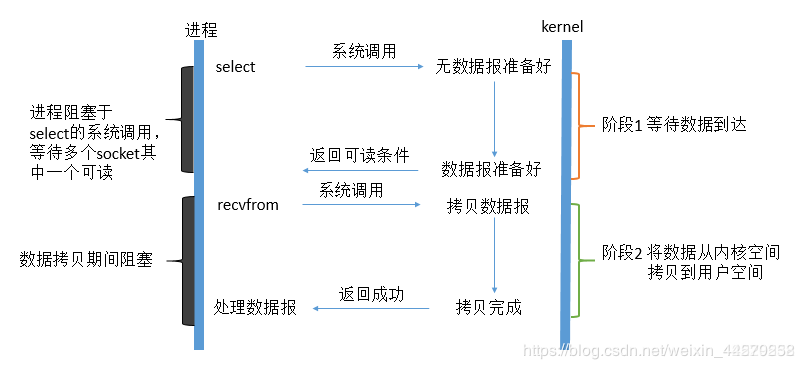


从图中可以看出，当用户进程发出read操作时，如果kernel中的数据还没有准备好，，那么他并不会block用户进程，而是立马返回一个error，从用户角度来讲，他发起一个read操作后，并不需要等待，而是马上等到一个结果，用户进程判断是一个error时，他就知道数据换没有准备好，于是它再次发送read操作，一旦kermel中的数据准备好了，并且再次收到了用户进程的read，那么他此时就会将数据拷贝到用户内存，然后返回。

所以用户的第一个阶段不是阻塞的，需要不断的主动问kernel数据好了没；第二阶段依然总体是阻塞的。

## IO多路复用（NIO）

select/epoll的好处就在于单个process就可以同时处理多个网络连接的IO。



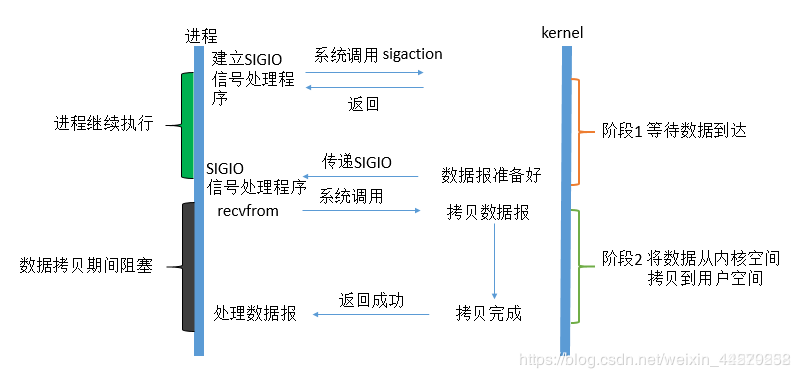
IO复用和同步阻塞本质一样，不过利用了新的select系统调用，由内核来负责本来是请求进程该做的轮询操作，看似不非阻塞IO还多了一个系统调用开销，不过因为支持多路IO才算提高了效率。

也就是一个可以监听多个。

它的基本原理就是select /epoll这个function会不断的轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程。它的流程如图：

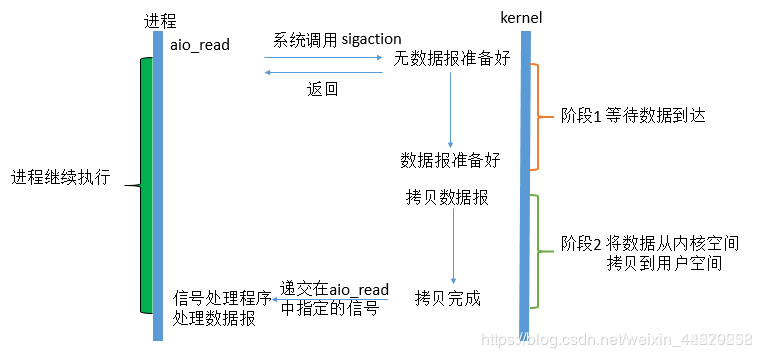
当用户线程调用select，那么整个进程会被阻塞，而同时，kernel会“监视”所有select负责的socket，当任何一个socket中的数据准备好了，select就会返回，这个时候用户进程在调用read操作，将数据kernel拷贝到用户进程。

## 信号驱动IO：



首先开启套接字的信号驱动式IO功能，并且通过sigaction(信号处理程序) 系统调用安装一个信号处理函数 ，该函数调用将立即返回，当前进程没有被阻塞 ，继续工作；当数据报准备好的时候，内核为该进程产生SIGIO 的信号，随后既可以在信号处理函数中调用recvfrom 读取数据报，并且通知主循环数据已经准备好等待处理；也可以直接通知主循环让它读取数据报；(其实就是一个待读取的通知和待处理的通知)，基本不会用到。

## 异步IO（AIO）



多线程和多进程的模型虽然解决了并发的问题，但是系统不能无限的增加线程，由于系统的切换线程的开销恒大，所以，一旦线程数量过多，CPU的时间就花在线程的切换上，正真运行代码的时间就会减少，结果导致性能严重下降

由于我们要解决的问题是CPU高速执行能力和IO设备的龟速严重不匹配，多线程和多进程只是解决这一个问题的一种方法。

另一种解决IO问题的方法是异步IO，当代码需要执行一个耗时的IO操作时，他只发出IO指令，并不等待IO结果然后就去执行其他代码，一段时间后，当IO返回结果是，在通知CPU进行处理

我们调用aio\_read函数，给内核传递描述符，缓冲区指针，缓冲区大小，和文件偏移量，并且告诉内核当整个操作完成时如何通知我们，该函数调用后，立即返回，不会被阻塞

另一方面：从kernel的角度，当他收到一个aio\_read之后，首先它立即返回，所以不会对用户进程产生block，然后kernel会等待数据准备完成，然后将数据拷贝到用户内存（copy由内核完成），当着一切完成后，kernel会给用户进程发送一个singal或者执行下一个基于线程回调函数来完成此次IO处理过程，告诉他read操作完成

## ****异步IO的拷贝是有内核完成， 其他几种IO都是由用户进程完成****

## 以上五种IO模型的通俗理解

活动：演唱会

角色一：满满

角色二：举办方-》售票员

角色三：黄牛

角色四：送票快递员

同步阻塞IO：

满满从家到售票点买票，售票员告诉满满，票明天才能卖。满满直接在售票点等到明天买票，然后回家。

非阻塞IO:

满满从家到演唱会现场向售票员买票，但票还未出来，然后满满就走了，去干别的事情，过了几个小时再来询问票是否出来，还没出来继续干别的事情。重复以上操作，直到票可以买。

IO复用： JAVA->selector / linux->select,poll,epoll

满满想买演唱会的票，打电话给黄牛(select)帮留一张票，票出来后，小明是需要花费时间去售票点买票(阻塞)。

信号IO:

满满想买演唱会门票，给举办方打电话，帮我留意票，可以售票了给我打个电话(打完就返回结果，等待kernel信号通知)，我自己来买票。票出来后，满满亲自去售票点买票

异步IO:

满满要看演唱会，给举办方打电话，可以售票了让送票快递员帮我把票送家里，满满就不用自己去专门买票了