在前面TCP并发服务器中，客户端在和服务端建立连接后先用fgets获取标准输入，再用write、read收发消息，这两个步骤都属于I/O模型使用，前者是标准的输入输出I/O，后者套接字使用的是网络I/O。当服务端断开连接时客户端由于阻塞在标准输I/O导致不能获取网络I/O的状态，应该有一种机制使得内核一旦发现一个或多个I/O条件就绪时能通知讲程序。这个能力就是I/O复用，主要有select、poll支持。

首先介绍基本的 概念，再介绍五种IO模型，最后比较分析这五种模型。

**1、I/O模型介绍**

**1.1 概念**

为更好了解IO模型，先需要理解同步、异步、阻塞、非阻塞几个概念。

（1）**同步**

在发出一个\*调用\*时，在没有得到结果之前，该\*调用\*就不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由\*调用者\*主动等待这个\*调用\*的结果。

（2）**异步**

和同步对应。当一个异步调用发出后，调用者不能理立刻得到结果。被调用者（实际处理这个调用的对象）在处理完成后，通过状态、通知通知调用者，或通过回调函数来通知调用者

（3）**阻塞**

阻塞调用是指调用结果返回之前，当前线程会被挂起，不占用系统资源。调用线程只有在得到结果之后才会返回。

（4）**非阻塞**

和阻塞的概念相对应，指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。

同步和异步关注的是消息通信机制，阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态。以“你打电话到书店找书”为例，拨通电话后询问老板是否有某一本书：

**同步**：老板会说“稍等，我查一下”，等老板查好(几秒，或者几天)了，就告诉你结果（返回结果）。

**异步：**老板说“我查好了，再打电话告诉你”，之后挂断电话。老板通过回电方式来回调。

**阻塞：**老板去查看，你一直拿着电话，一直等到老板查询完告诉结果，中途不干任何其他事。

**非阻塞**：老板去查看，电放放一边，你先离开一段时间处理其他事，之后再过来拿起电话问一次结果。

**1.2、五种IO模型**

在了解了同步与异步、阻塞与非阻塞概念后，我们来讲讲linux的五种IO模型：

1)阻塞I/O（blocking I/O）

2)非阻塞I/O （nonblocking I/O）

3) I/O复用(select 和poll) （I/O multiplexing）

4)信号驱动I/O （signal driven I/O (SIGIO)）

5)异步I/O （asynchronous I/O (the POSIX aio\_functions)）

其中前4种都是同步，最后一种才是异步。由signal driven IO（信号驱动IO）在实际中并不常用，所以主要介绍其余四种IO Model。

对于一个network IO (这里我们以read举例)，它会涉及到两个系统对象，一个是调用这个IO的process (or thread)，另一个就是系统内核(kernel)。当一个read操作发生时，该操作会经历两个阶段：

#1）等待数据准备 (Waiting for the data to be ready)

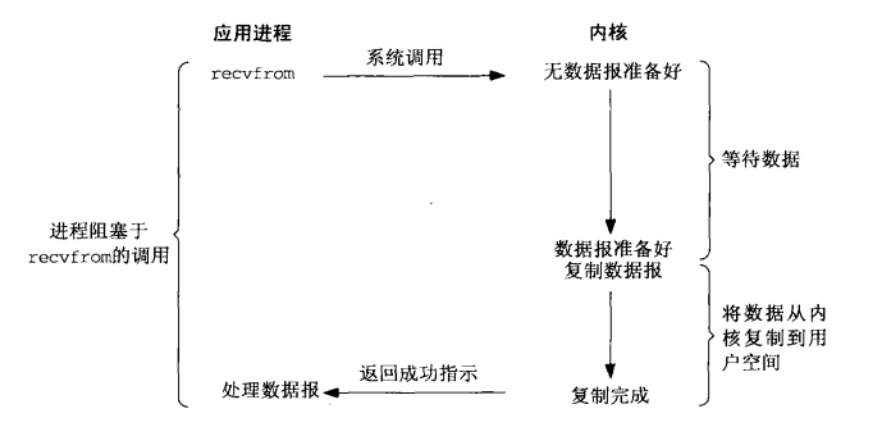
#2）将数据从内核拷贝到进程中(Copying the data from the kernel to the process)

记住这两点很重要，因为以上IO模型的区别就是在两个阶段上各有不同的情况。

**2、I/O模型**

**2.1 阻塞I/O（blocking I/O）**

最流行的I/O模型就是阻塞式I/O模型，linux下套接字默认也是阻塞的。以数据报读为例：



当用户进程调用了recvfrom这个系统调用，内核就开始了IO的第一个阶段：准备数据。对于network io来说，很多时候数据在一开始还没有到达（比如，还没有收到一个完整的UDP包），这个时候内核就要等待足够的数据到来。

而在用户进程这边，整个进程会被阻塞。当kernel一直等到数据准备好了，它就会将数据从内核中拷贝到用户内存，然后内核返回结果，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（等待数据和拷贝数据两个阶段）都被block了。

实际上，除非特别指定，几乎所有的IO接口 ( 包括socket接口 ) 都是阻塞型的，只有当该系统调用获得结果或者超时出错时才返回。

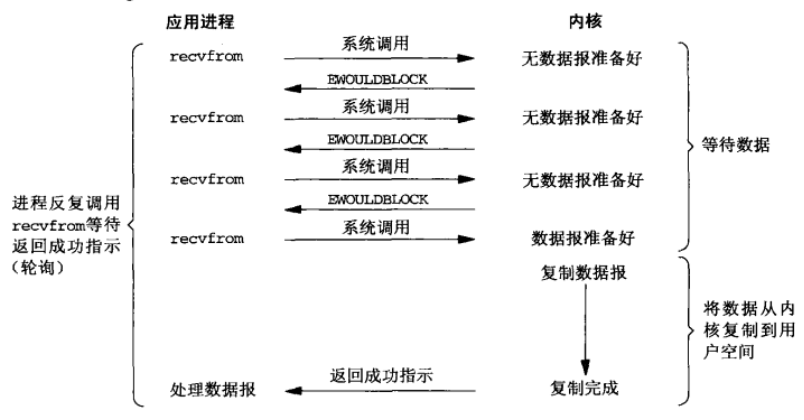
这给网络编程带来了一个很大的问题，如在调用recv(1024)的同时，线程将被阻塞，在此期间，线程将无法执行任何运算或响应任何的网络请求。

有一个简单的解决方案。正如前面例子所示，在服务端使用多线程让每个客户端都拥有独立的线程进行处理，任何一个客户端连接线程阻塞不会影响其他的连接。但这个方案存在问题，当遇到成百上千的请求时，多线程或多进程的调度都会严重占据系统资源，降低系统响应效率，且容易进入假死状态。另外，使用线程池方案能在一定程序上缓解资源调度问题，但线程池的数量也是由有上限的。

总之，多线程模型可以方便高效的解决小规模的服务请求，但面对大规模的服务请求，多线程模型也会遇到瓶颈，可以用非阻塞接口来尝试解决这个问题。

**2.2 非阻塞I/O（non-blocking I/O）**

将套接字设置成non-blocking模式，使得在调用recvfrom读取数据时，立即返回一个结果。



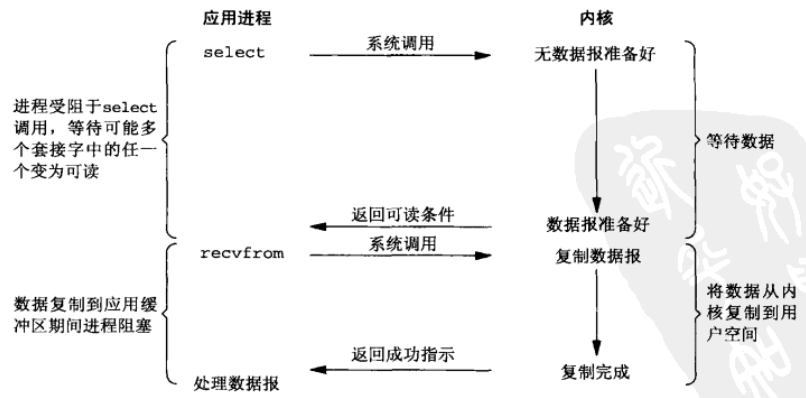
前三次调用recvfrom时数据未准备好，内核立即返回EWOULDBLOCK错误；第四次调用时已有数据准备好，数据被复制到用户内存(**这一阶段仍然是阻塞的**)，之后recvfrom函数返回。

非阻塞的recvform系统调用被调用之后，进程并没有被阻塞，内核马上返回给进程一个error。进程在返回后，可以做其他事情，之后再次调用recvfrom，不断循环上述步骤直到数据准备好。这个过程被就是轮询（pooling）。

在非阻塞式IO中，用户进程其实是需要不断的主动询问kernel数据准备好了没有。虽然能够在等待任务完成的时间里做其他事，但是循环调用系统调用提高CPU占用率，并且任务完成的响应延迟增大导致整体的数据吞吐量降低。这种方式主要是起到检测“操作是否完成”，但是系统提供了更高效的方式，例如select()多路复用可以一次检测多个连接是否就绪。

**2.3 I/O复用（IO multiplexing）**

使用select()或者poll()，阻塞在这两个函数上，而不是真正的I/O系统上。



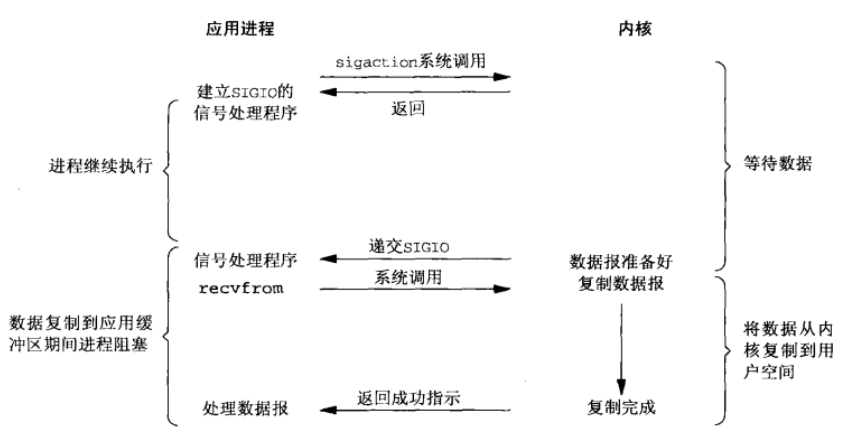
进程阻塞于select()或者poll()，内核会轮询器负责的所有socket，当任何一个socket上的数据准备好，select()就会返回。之后再调用recvfrom，将数据从内核中拷贝到用户内存。

相对于阻塞IO方式， IO复用显得并没有多大优势，甚至因为有两个系统调用而性能更差，其优势在于其能同时处理多个连接。

在多线程中使用调用阻塞IO系统调用的方式，与IO复用模型使用及其相似。但是**使用IO复用模型的只需要单线程执行，占用资源少**。同样地，IO复用也存在问题：当需要探测的句柄值或socket值较大时，select()函数就需要消耗大量时间去轮询；不同操作系统都提供了更高效的方式，例如linux提供了epoll，BSD提供了kqueue，Solaris提供了/dev/poll，跨平台较难。

**2.4 信号驱动I/O**

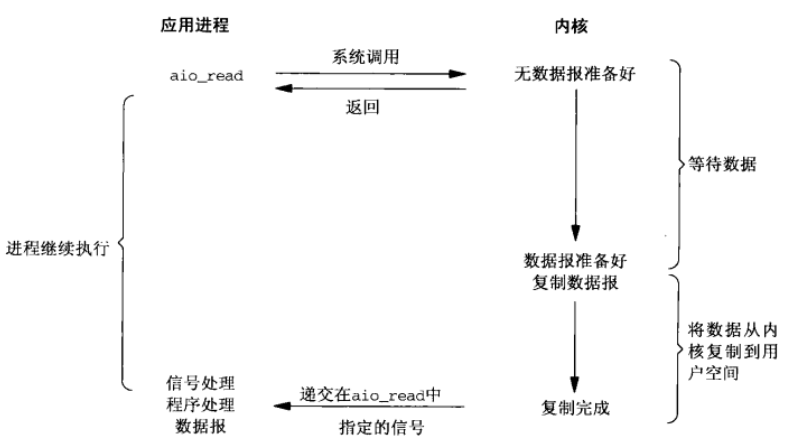
使用信号让内核在数据就绪是发送SIGIO信号通知应用进程，这种模型称为信号驱动I/O模型。这种方式使用较少。



先开启套接字的信号驱动功能，并传递一个用于接收系统通知的回调函数给sigaction系统调用，这个系统调用将立即返回。当数据准备好时，内核会产生一个SIGIO信号，我们可以在信号处理函数中调用recvform接收数据或者通知主线程读取数据。其优势在于等到数据的过程不是阻塞的，主循环可以继续执行，只要等到来自信号处理的函数的通知。

**2.5 异步I/O（Asynchronous I/O）**

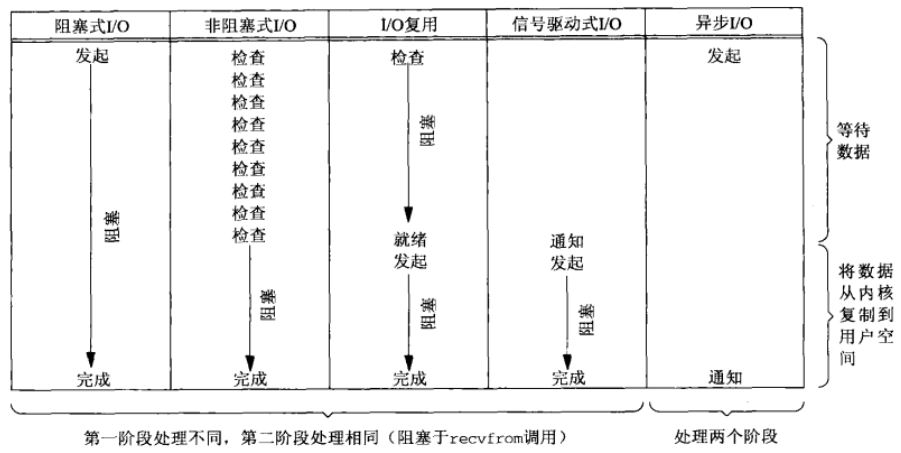
异步I/O模型的运行机制是，告知内核执行某个操作，并让内核在整个操作（包括数据从内核空间复制到用户内存）完成后通知应用进程。与信号驱动I/O相比的区别在于：**信号驱动IO是让内核通知应用进程何时可以启动一个IO操作，而异步IO是让内核通知应用进程IO操作何时完成**。



调用aio\_read函数，需要传递描述符、缓冲区地址、缓冲区带下和文件偏移，并告诉内核整个操作完成时如何通知应用进程。当应用进程调用aio\_read系统调用后，立即返回，不被阻塞。当数据准备好，内核将数据从内核空间拷贝到用户内存中后，会通知应用进程整个操作已经执行完成。

**3、模型比较分析**

如下图，五种IO模型中，前四种主要区别在第一阶段，第二阶段都是相同的：从内核空间拷贝数据到用户内存空间，进程阻塞于recvfrom。相反，异步IO模型在这两个阶段都要处理，从而不同于其他模型。



阻塞于非阻塞的区别很明显，调用block IO时，前者一直block进程知道操作完成，而后者不论数据是都就绪都可以直接返回。前四种模型都是同步的，只有第五种是异步的。尽管前四种中的non block模式没有“被block”，但是他们真正的IO操作是recvfrom系统调用是阻塞的。另外，非阻塞IO和异步IO都是能立即返回，但是非阻塞IO需要应用进程主动去轮询检查数据是否就绪进而再从内核空间拷贝数据到用户控件，而异步IO中应用进程既不需要轮询也不需要拷贝数据，仅仅等待接收整个操作完成的通知。