

## 【23期】请你谈谈关于IO同步、异步、阻塞、非阻塞的区别

---

同步 (synchronous) IO和异步 (asynchronous) IO，阻塞 (blocking) IO和非阻塞 (non-blocking) IO分别是什么，到底有什么区别？

这个问题其实不同的人给出的答案都可能不同，比如wiki，就认为asynchronous IO和non-blocking IO是一个东西。这其实是因为不同的人的知识背景不同，并且在讨论这个问题的时候上下文(context)也不相同。所以，为了更好的回答这个问题，我先限定一下本文的上下文。

本文讨论的背景是Linux环境下的network IO。

本文最重要的参考文献是Richard Stevens的“UNIX® Network Programming Volume 1, Third Edition: The Sockets Networking”，6.2节“I/O Models”，Stevens在这节中详细说明了各种IO的特点和区别，如果英文够好的话，推荐直接阅读。

Stevens的文风是有名的深入浅出，所以不用担心看不懂。本文中的流程图也是截取自参考文献。

Stevens在文章中一共比较了五种IO Model：

- blocking IO
- nonblocking IO
- IO multiplexing
- signal driven IO
- asynchronous IO

由于signal driven IO在实际中并不常用，所以我这只提及剩下的四种IO Model。

再说一下IO发生时涉及的对象和步骤。

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，它会经历两个阶段：

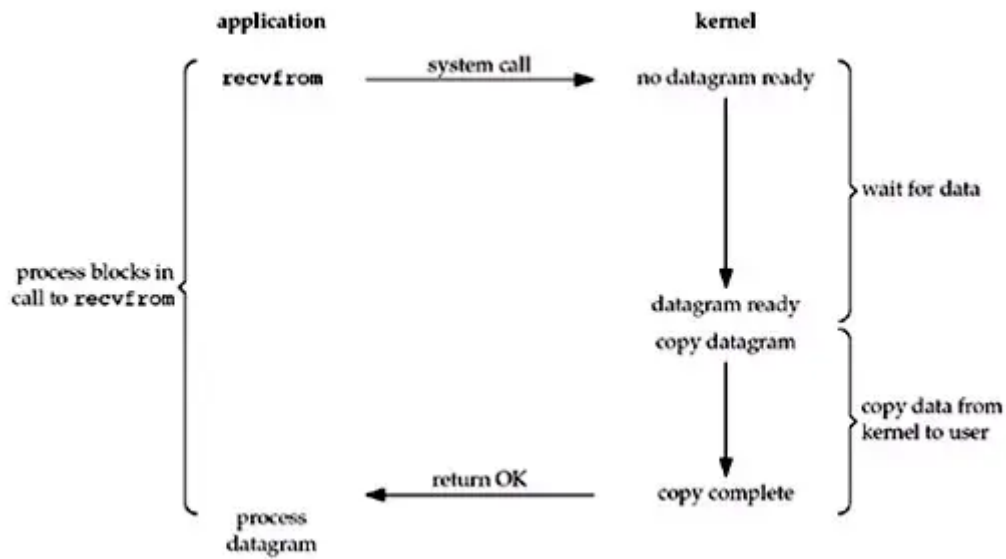
- 等待数据准备 (Waiting for the data to be ready)
- 将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)

记住这两点很重要，因为这些IO Model的区别就是在两个阶段上各有不同的情况。

### blocking IO

在linux中，默认情况下所有的socket都是blocking，一个典型的读操作流程大概是这样：

**Figure 6.1. Blocking I/O model.**



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。

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kernel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。

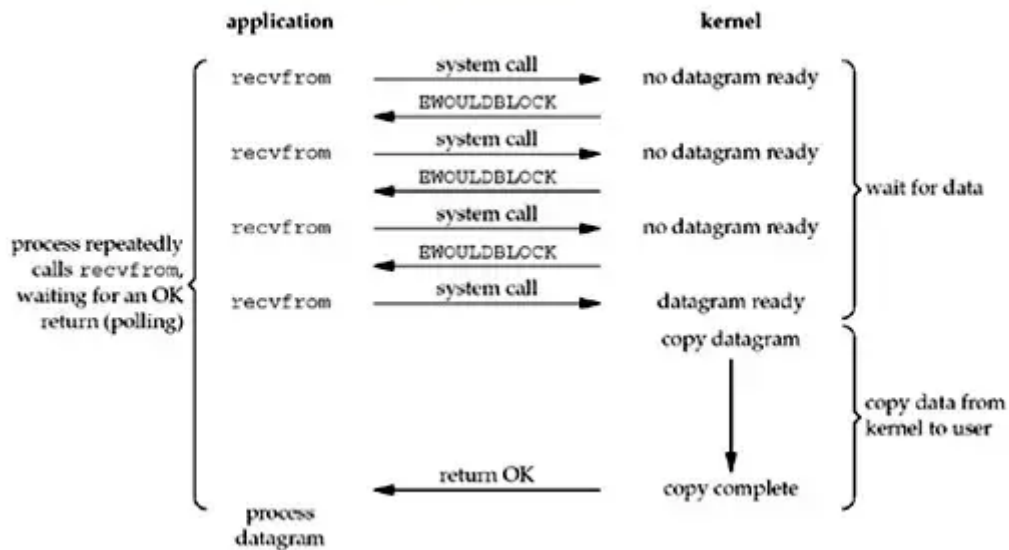
当kernel一直等到数据准备好了，它就会将数据从kernel中拷贝到用户内存，然后kernel返回结果，用户进程才解除block的状态，重新运行起来。

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

## non-blocking IO

linux下，可以通过设置socket使其变为non-blocking。当对一个non-blocking socket执行读操作时，流程是这个样子：

Figure 6.2. Nonblocking I/O model.



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，那么它并不会block用户进程，而是立刻返回一个error。从用户进程角度讲，它发起一个read操作后，并不需要等待，而是马上就得到了一个结果。

用户进程判断结果是一个error时，它就知道数据还没有准备好，于是它可以再次发送read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。

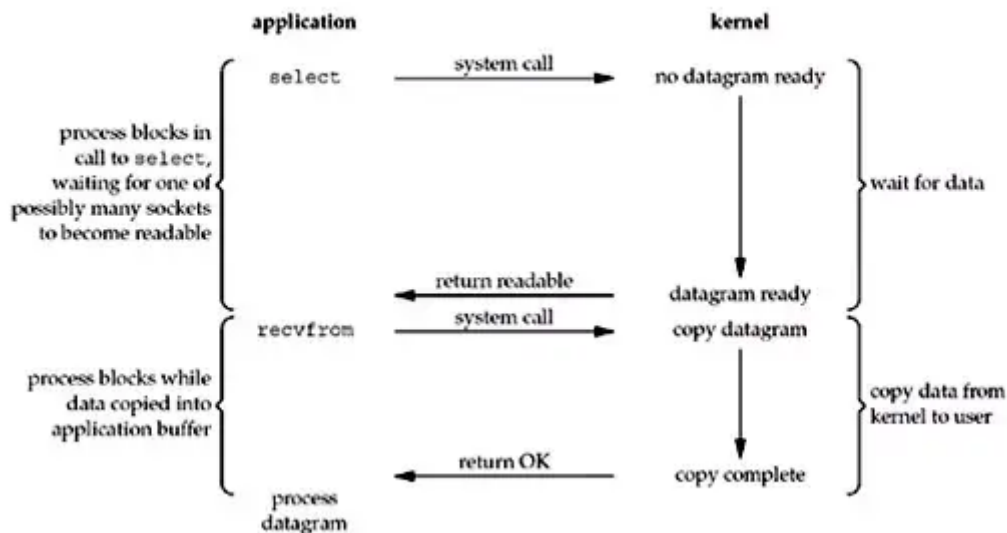
所以，用户进程其实是需要不断的主动询问kernel数据好了没有。

## IO multiplexing

IO multiplexing这个词可能有点陌生，但是如果我说select，epoll，大概就都能明白了。有些地方也称这种IO方式为event driven IO。我们都知道，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。

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：

Figure 6.3. I/O multiplexing model.



当用户进程调用了select，那么整个进程会被block，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。这个时候用户进程再调用read操作，将数据从kernel拷贝到用户进程。

这个图和blocking IO的图其实并没有太大的不同，事实上，还更差一些。因为这里需要使用两个system call (select 和 recvfrom)，而blocking IO只调用了—个system call (recvfrom)。但是，用select的优势在于它可以同时处理多个connection。

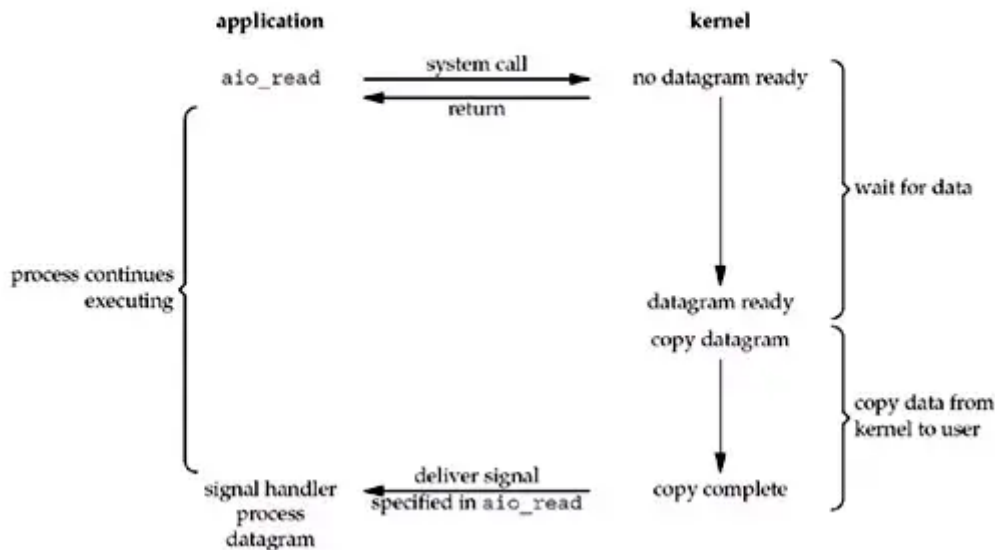
多说一句。所以，如果处理的连接数不是很高的话，使用select/epoll的web server不一定比使用multi-threading + blocking IO的web server性能更好，可能延迟还更大。select/epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。

在IO multiplexing Model中，实际中，对于每一个socket，一般都设置成为non-blocking，但是，如上图所示，整个用户的process其实是一直被block的。只不过process是被select这个函数block，而不是被socket IO给block。

## Asynchronous I/O

linux下的asynchronous IO其实用得很少。先看一下它的流程：

Figure 6.5. Asynchronous I/O model.



用户进程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从kernel的角度，当它受到一个asynchronous read之后，首先它会立刻返回，所以不会对用户进程产生任何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，当这一切都完成之后，kernel会给用户进程发送一个signal，告诉它read操作完成了。

到目前为止，已经将四个IO Model都介绍完了。现在回过头来回答最初的那几个问题：**blocking和non-blocking的区别在哪，synchronous IO和asynchronous IO的区别在哪。**

先回答最简单的这个：blocking vs non-blocking。前面的介绍中其实已经很明确的说明了这两者的区别。调用blocking IO会一直block住对应的进程直到操作完成，而non-blocking IO在kernel还准备数据的情况下会立刻返回。

在说明synchronous IO和asynchronous IO的区别之前，需要先给出两者的定义。Stevens给出的定义（其实是POSIX的定义）是这样子的：

A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes;

An asynchronous I/O operation does not cause the requesting process to be blocked;

两者的区别就在于synchronous IO做”IO operation”的时候会将process阻塞。按照这个定义，之前所述的blocking IO，non-blocking IO，IO multiplexing都属于synchronous IO。

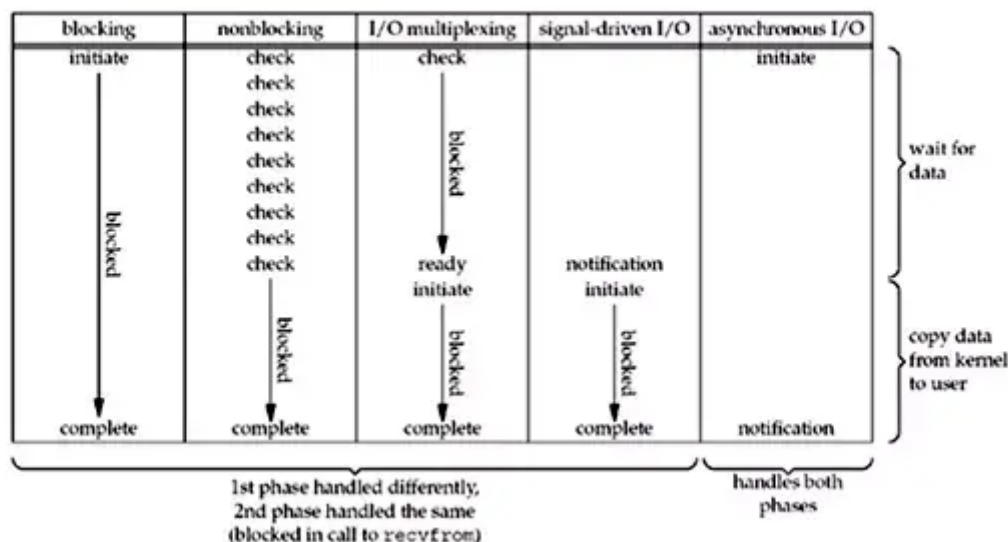
有人可能会说，non-blocking IO并没有被block啊。这里有个非常“狡猾”的地方，定义中所指的”IO operation”是指真实的IO操作，就是例子中的recvfrom这个system call。non-blocking IO在执行recvfrom这个system call的时候，如果kernel的数据没有准备好，这时候不会block进程。

但是，当kernel中数据准备好的时候，recvfrom会将数据从kernel拷贝到用户内存中，这个时候进程是被block了，在这段时间内，进程是被block的。而asynchronous IO则不一样，当进程发起IO操作之后，就直

接返回再也不理睬了，直到kernel发送一个信号，告诉进程说IO完成。在这整个过程中，进程完全没有被block。

各个IO Model的比较如图所示：

Figure 6.6. Comparison of the five I/O models.



经过上面的介绍，会发现non-blocking IO和asynchronous IO的区别还是很明显的。在non-blocking IO中，虽然进程大部分时间都不会被block，但是它仍然要求进程去主动的check，并且当数据准备完成以后，也需要进程主动的再次调用recvfrom来将数据拷贝到用户内存。

而asynchronous IO则完全不同。它就像是用户进程将整个IO操作交给了他人（kernel）完成，然后他人做完后发信号通知。在此期间，用户进程不需要去检查IO操作的状态，也不需要主动的去拷贝数据。

最后，再举几个不是很恰当的例子来说明这四个IO Model:

有A，B，C，D四个人在钓鱼：

- A用的是最老式的鱼竿，所以呢，得一直守着，等到鱼上钩了再拉杆；
- B的鱼竿有个功能，能够显示是否有鱼上钩，所以呢，B就和旁边的MM聊天，隔会再看看有没有鱼上钩，有的话就迅速拉杆；
- C用的鱼竿和B差不多，但他想了一个好办法，就是同时放好几根鱼竿，然后守在旁边，一旦有显示说鱼上钩了，它就将对应的鱼竿拉起来；
- D是个有钱人，干脆雇了一个人帮他钓鱼，一旦那个人把鱼钓上来了，就给D发个短信。

出处：<https://dwz.cn/gg7wFJTh>

