原创

# python之IO多路复用

忘情OK 关注

2017-02-06 15:08:47 775人阅读 2人评论

同步IO和异步IO，阻塞IO和非阻塞IO分别是什么，到底有什么区别?

不同的人在不同的上下文下给出的答案是不同的。所以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。

在进行解释之前，首先要说明几个概念：

进程切换

进程的阻塞

文件描述符

缓存 I/O

**进程切换**

为了控制进程的执行，内核必须有能力挂起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行。这种行为被称为进程切换。

因此可以说，任何进程都是在操作系统内核的支持下运行的，是与内核紧密相关的。

从一个进程的运行转到另一个进程上运行，这个过程中经过下面这些变化：

1. 保存处理器上下文，包括程序计数器和其他寄存器
2. 更新PCB信息
3. 把进程的PCB移入相应的队列，如就绪、在某事件阻塞等队列
4. 选择另一个进程执行，并更新其PCB
5. 更新内存管理的数据结构
6. 恢复处理器上下文

进程控制块**PCB**（Processing Control Block），是操作系统核心中一种数据结构，主要表示进程状态。

PCB的作用是使一个在多道程序环境下不能独立运行的程序（含数据），成为一个能独立运行的基本单位或与其它进程并发执行的进程。

或者说，OS是根据PCB来对并发执行的进程进行控制和管理的。

PCB通常是系统内存占用区中的一个连续存区，它存放着操作系统用于描述进程情况及控制进程运行所需的全部信息

正在执行的进程，由于期待的某些事件未发生，如请求系统资源失败、等待某种操作的完成、新数据尚未到达或无新工作做等，则由系统自动执行阻塞原语(Block)，使自己由运行状态变为阻塞状态。可见，进程的阻塞是进程自身的一种主动行为，也因此只有处于运行态的进程（获得CPU），才可能将其转为阻塞状态。当进程进入阻塞状态，是不占用CPU资源的。

**文件描述符fd**

文件描述符（File descriptor）是计算机科学中的一个术语，是一个用于表述指向文件的引用的抽象化概念。

文件描述符在形式上是一个非负整数。实际上，它是一个索引值，指向内核为每一个进程所维护的该进程打开文件的记录表。

当程序打开一个现有文件或者创建一个新文件时，内核向进程返回一个文件描述符。

在程序设计中，一些涉及底层的程序编写往往会围绕着文件描述符展开。但是文件描述符这一概念往往只适用于UNIX、Linux这样的操作系统。

**缓存 I/O**

缓存 I/O 又被称作标准 I/O，大多数文件系统的默认 I/O 操作都是缓存 I/O。

在 Linux 的缓存 I/O 机制中，操作系统会将 I/O 的数据缓存在文件系统的页缓存（ page cache ）中。

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区拷贝到应用程序的地址空间。

缓存 I/O 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，这些数据拷贝操作所带来的 CPU 以及内存开销是非常大的。

对于一次IO访问（以read举例），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区拷贝到应用程序的地址空间。

一个IO（如read）操作会经历以下两个阶段：

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)

因为有了这两个阶段，linux系统产生了下面五种网络模式的方案。

1.阻塞 I/O（blocking IO）

2.非阻塞 I/O（nonblocking IO）

3.I/O 多路复用（ IO multiplexing）

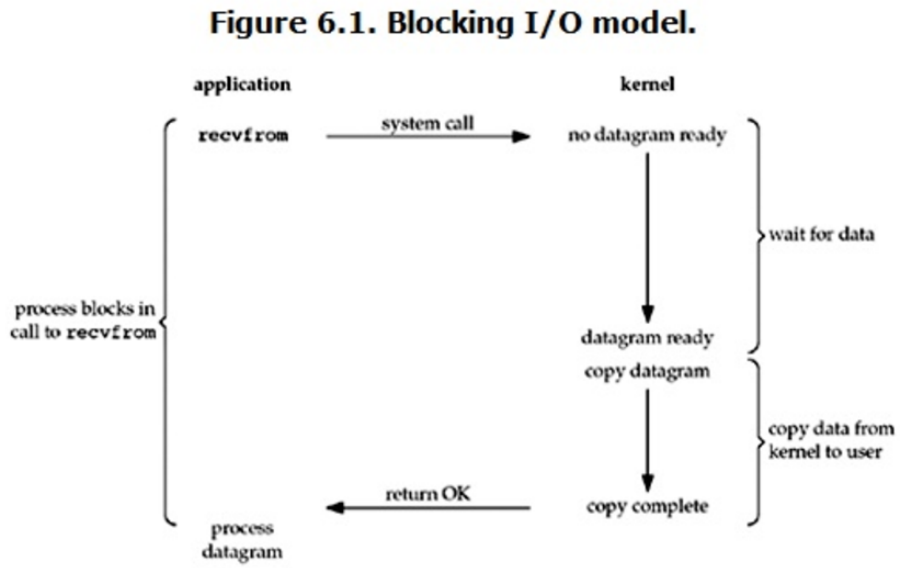
4.信号驱动 I/O（ signal driven IO）

5.异步 I/O（asynchronous IO）

由于signal driven IO（信号驱动）在实际中并不常用，所以 里只提及剩下的四种IO Model。

**阻塞 I/O（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：

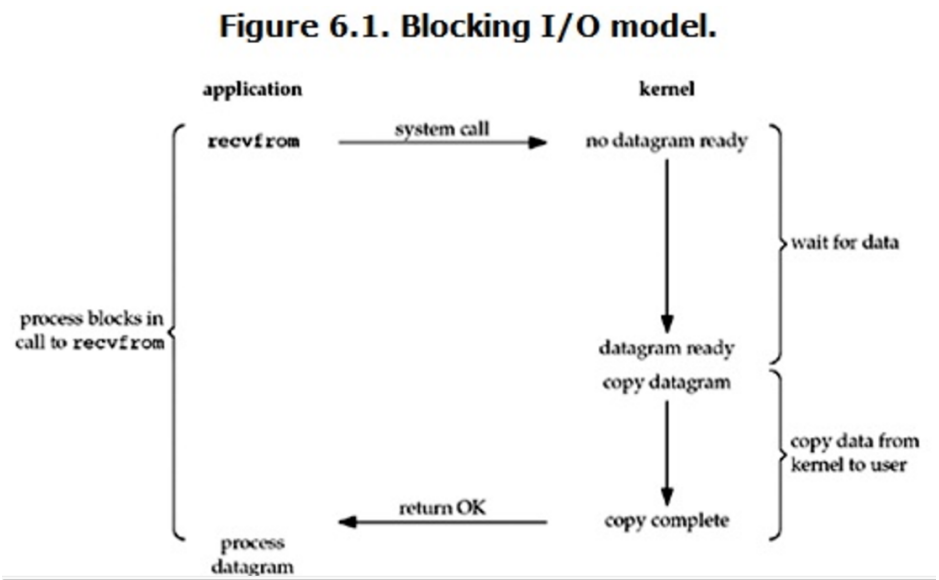



当用户进程调用了recvfrom这个系统调用，kernel就开始了IO的第一个阶段：准备数据（对于网络IO来说，很多时候数据在一开始还没有到达。比如，还没有收到一个完整的UDP包。这个时候kernel就要等待足够的数据到来）。这个过程需要等待，也就是说数据被拷贝到操作系统内核的缓冲区中是需要一个过程的。而在用户进程这边，整个进程会被阻塞（当然，是进程自己选择的阻塞）。当kernel一直等到数据准备好了，它就会将数据从kernel中拷贝到用户内存，然后kernel返回结果，用户进程才解除block的状态，重新运行起来。

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

**非阻塞 I/O（nonblocking IO）**

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




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。

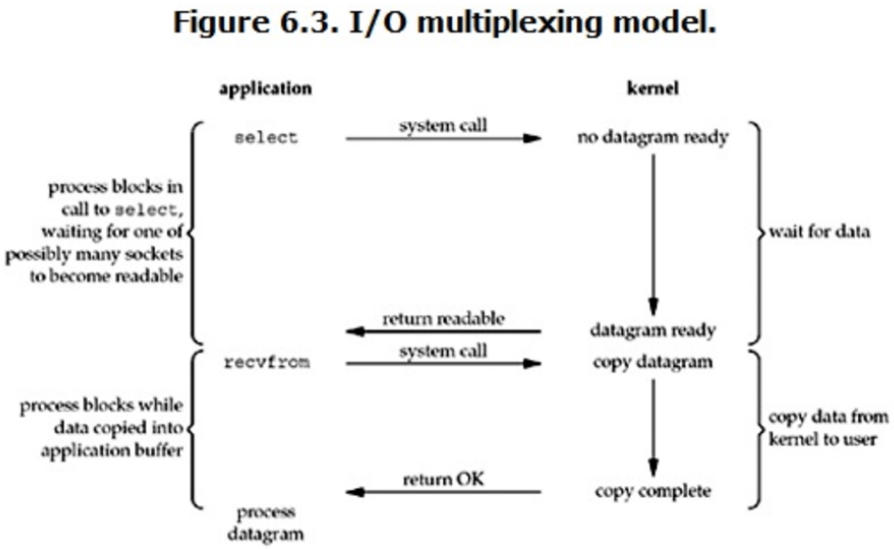
所以，nonblocking IO的特点是用户进程需要不断的主动询问kernel数据好了没有。

IO multiplexing就是我们说的select，poll，epoll，有些地方也称这种IO方式为event driven IO。

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，poll，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。




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。

所以，I/O 多路复用的特点是通过一种机制使一个进程能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，select()函数就可以返回。

IO多路复用和阻塞IO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)，而阻塞IO只调用了一个system call (recvfrom)。但是，用select的优势在于它可以同时处理多个连接。

如果处理的连接数不是很高的话，使用select/epoll的web server不一定比使用多线程+阻塞IO的web server性能更好，可能延迟还更大。

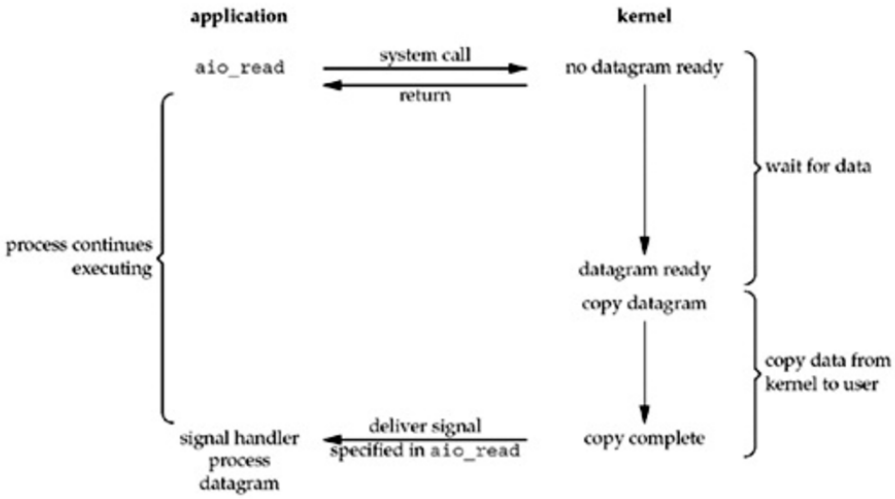
select/epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。

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。

**异步 I/O（asynchronous IO）**

**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。

**blocking和non-blocking的区别**

调用blocking IO会一直block住对应的进程直到操作完成

调用non-blocking IO在kernel还准备数据的情况下会立刻返回

**synchronous IO和asynchronous IO的区别**

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。

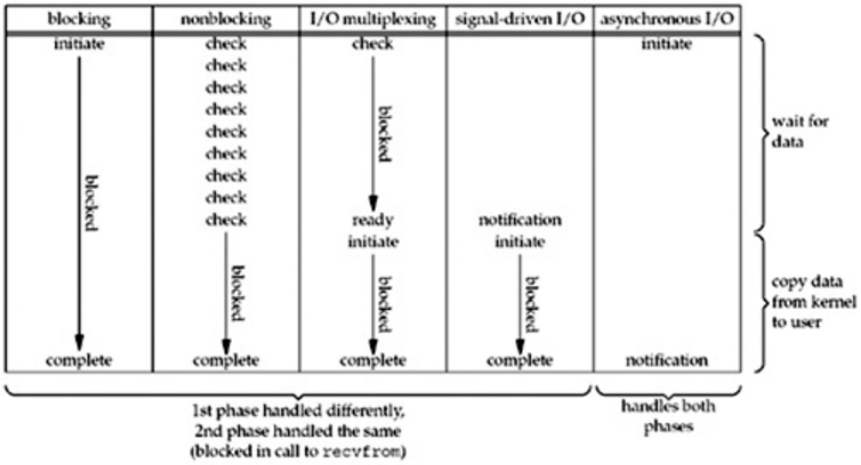
之前所说的blocking IO，non-blocking IO，IO multiplexing都属于synchronous IO。

有人会说，non-blocking IO并没有被block啊。 这里需要格外注意，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.**




通过上面的图片，可以发现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。

**I/O 多路复用之select、poll、epoll详解，请往这走**



IO 复用 多路 Python

1 收藏 分享

上一篇：python之协程 下一篇：python之IO多路复用（二）...

忘情OK

83篇文章，18W+人气，22粉丝

关注

提问和评论都可以，用心的回复会被更多人看到和认可

Ctrl+Enter 发布 取消 发布

**2条评论** 按时间正序 | 按时间倒序

bwhtie


1 0 2 分享

忘情OK 关注