Socket

IO模型

一个输入操作通常包括两个阶段：

1. 等待数据准备好；
2. 从内核向进程复制数据；

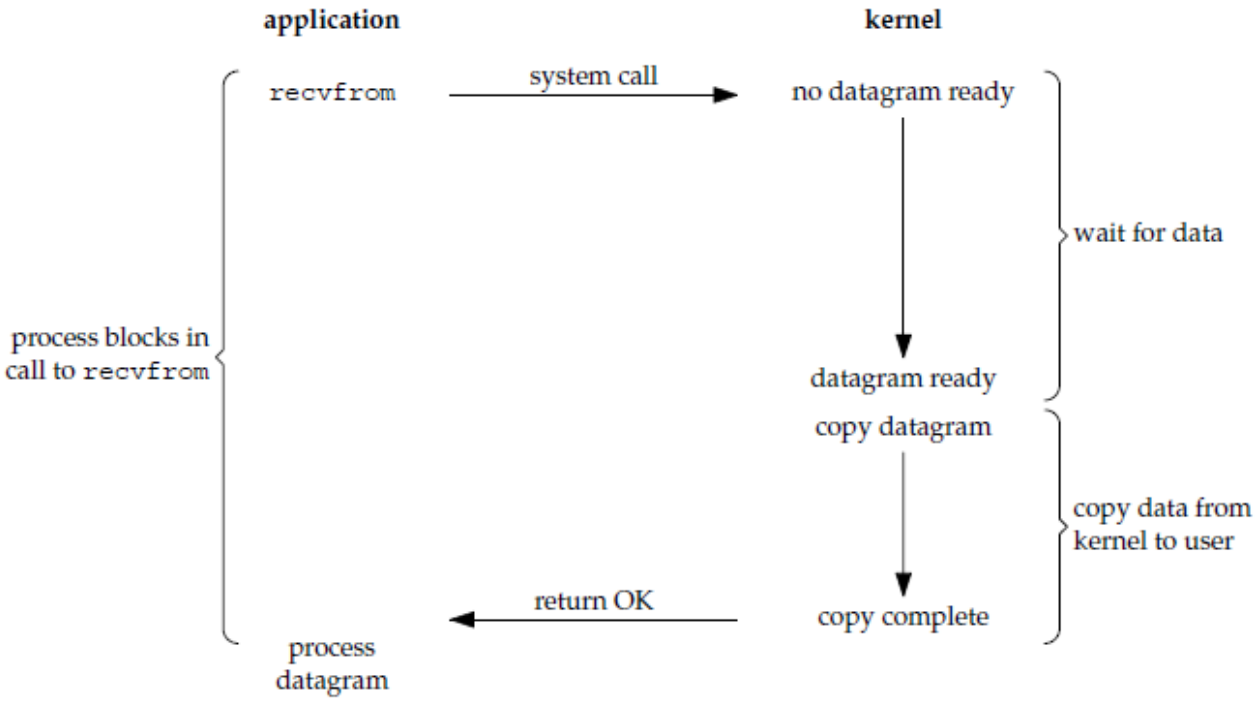
对于一个套接字上的输入操作，第一步通常涉及等待数据从网络中到达。当所等待数据到达时，它被复制到内核的某个缓冲区；第二步就是将数据从内核缓冲区复制到进程缓冲区；

Unix有5种IO模型：

1. 阻塞式IO；
2. 非阻塞式IO；
3. IO复用；
4. 信号驱动式；
5. 异步IO；

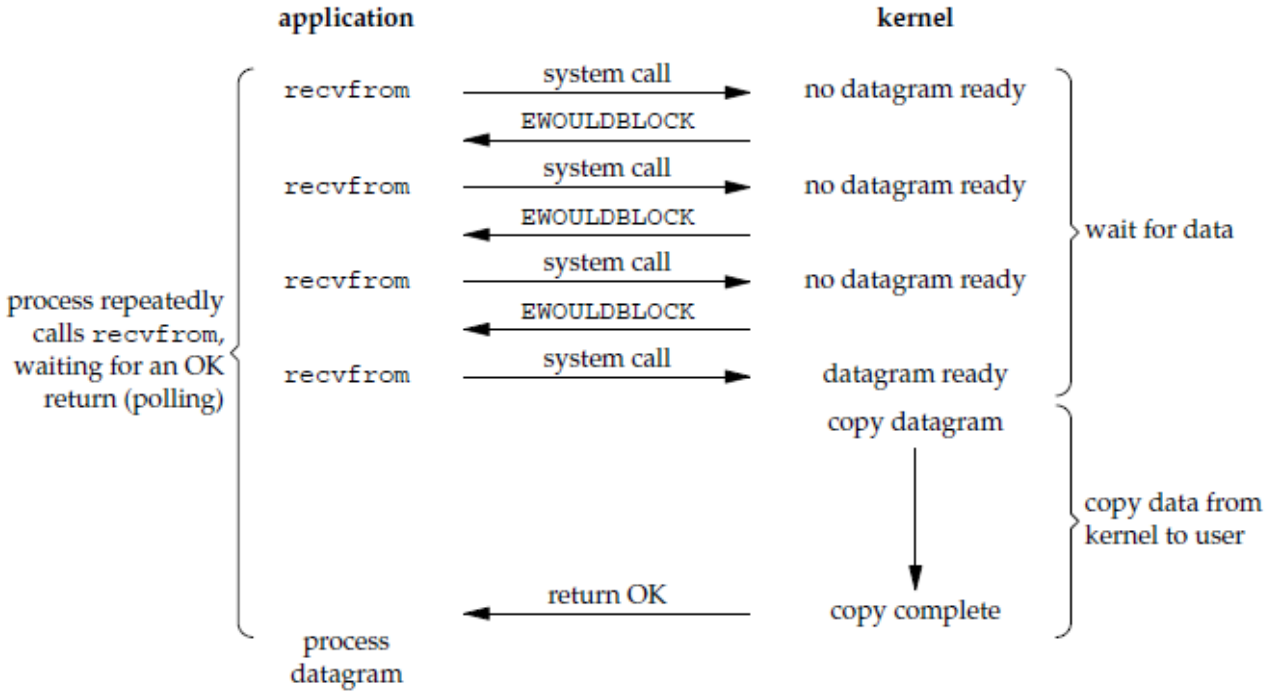
阻塞式IO 应用进程被阻塞，知道数据从内核缓冲区复制到进程缓冲区中才返回；

在阻塞过程中国，其他应用还在执行，不意味着整个操作系统的阻塞。其他应用进程还可以执行，所以不消耗CPU时间，利用率会比较高；



非阻塞式IO 应用进程执行系统调用，内核返回一个错误码。应用进程继续执行，但是需要不断的执行系统调用来获知IO是否完成，这种方式称之为轮询；

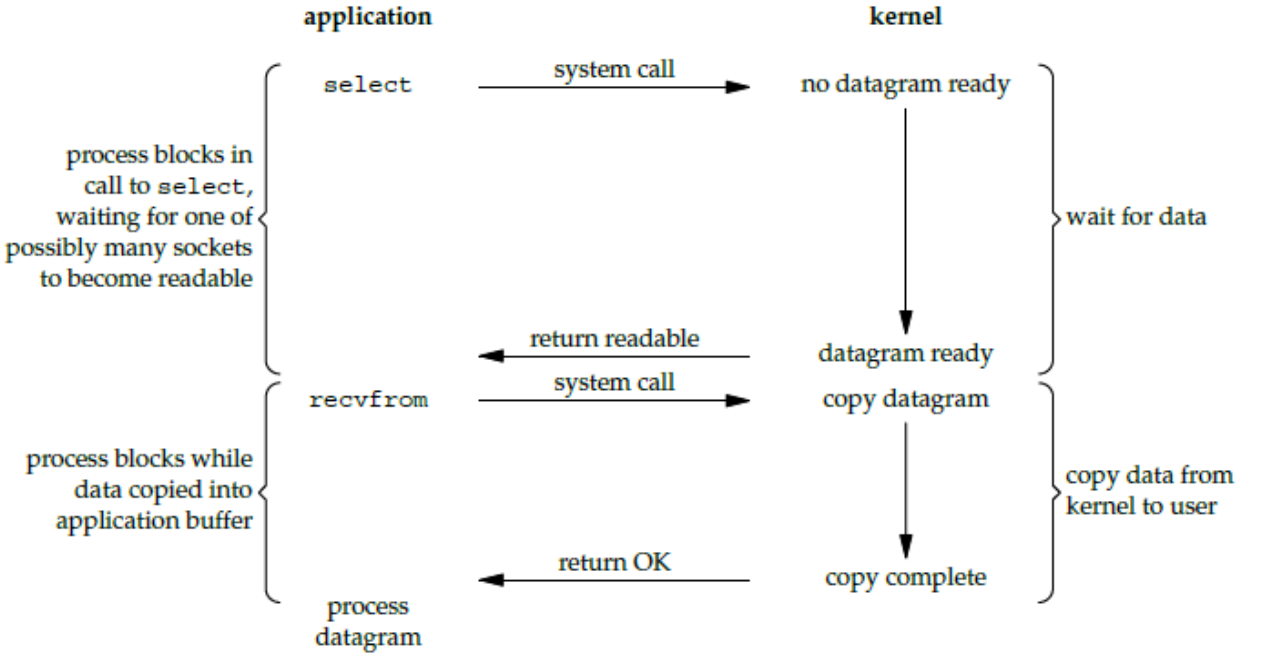
CPU要一直处理系统调用，因此这种模型的CPU利用率比较低；



IO复用 使用select或者poll等待数据，等待多个套接字中的任何一个变为可读。这个过程汇报阻塞，当其中一个套接字可读时返回，之后再使用recvfrom吧数据从内核复制到进程中；

它可以让单个进程具有处理多个I/O事件的能力。又被称为 Event Driven I/O，即事件驱动I/O。

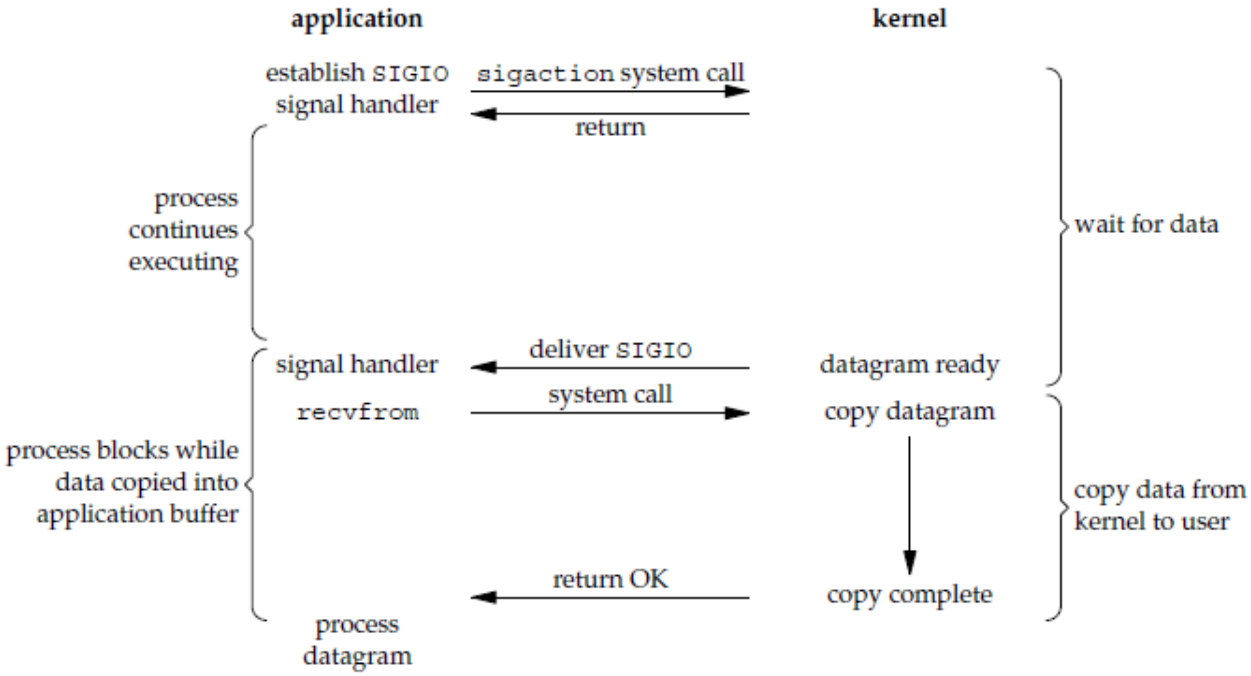
如果没有IO复用，每一个socket连接都需要创建一个线程去处理，有了IO复用就不会占用很多的进程创建和切换的开销。



信号驱动IO 应用进程使用sigaction系统调用，内核立即返回，应用进程可以继续执行，也就是说等待数据的阶段应用进程是非阻塞的。内核在数据到达时向应用进程发送SIGIO信号，应用进程收到之后在信号处理程序中调

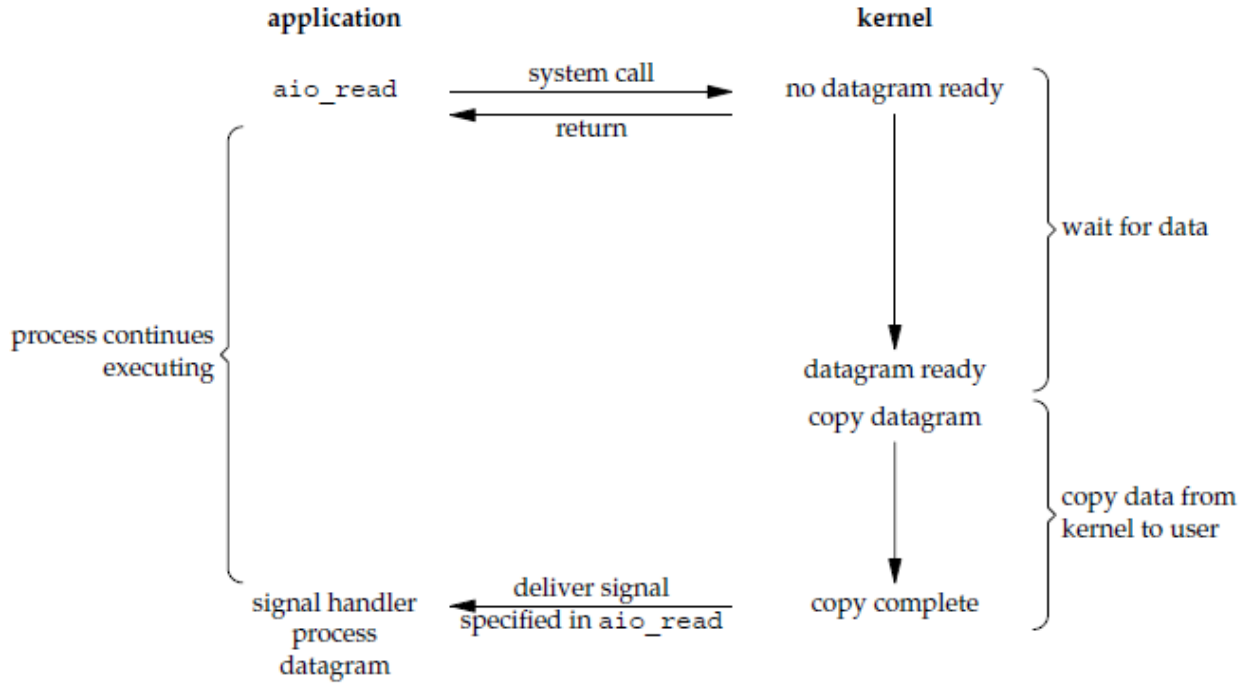
用recvfrom将数据从内核复制到应用进程中。

相比于非阻塞式 I/O 的轮询方式，信号驱动 I/O 的 CPU 利用率更高。



异步IO 应用进程执行aio_read系统调用之后，应用进程可以继续执行，不会被阻塞，内核会在所有操作完成之后，向应用进程发送信号。

异步IO和信号驱动IO的区别在于，异步IO的信号是通知应用进程IO完成，而信号驱动IO的信号是通知应用进程可以开始IO；

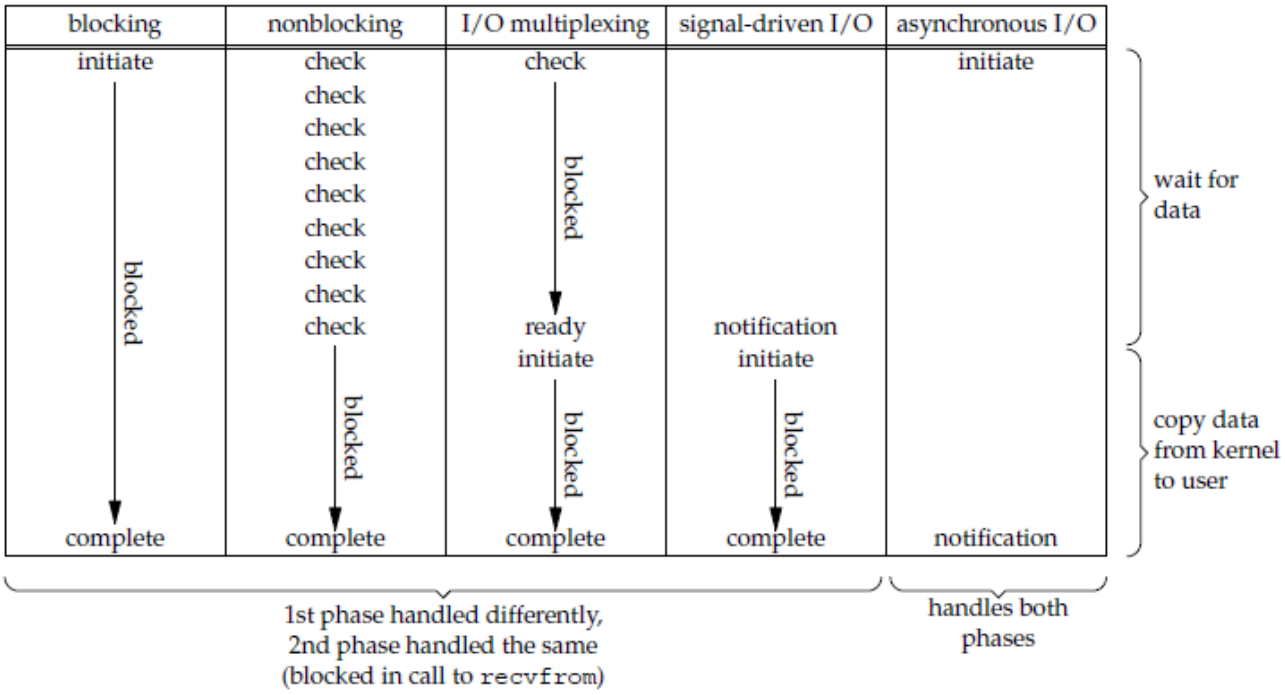


五大IO模型比较

同步IO：将数据从内核缓冲区中复制到应用进程缓冲区的阶段（第二阶段），应用进程会阻塞； 异步IO：第二阶段应用进程不会阻塞；

同步IO包括阻塞式IO，非阻塞式IO，IO复用和信号驱动IO，他们的主要区别就是在第一个阶段；

非阻塞IO，信号IO和异步IO在第一阶段不会阻塞；



IO复用

select/poll/epoll都是IO多路复用发具体实现，select出现的最早，之后是poll再是epoll；

select 允许应用程序监视一组文件描述符，等待一个或者多个描述符成为就绪状态，从而完成 I/O 操作；

fd_set 使用数组实现，数组大小使用 FD_SETSIZE 定义，所以只能监听少于 FD_SETSIZE 数量的描述符。有三种类型的描述符类型：readset、writeset、exceptset，分别对应读、写、异常条件的描述符集合。

timeout 为超时参数，调用 select 会一直阻塞直到有描述符的事件到达或者等待的时间超过 timeout。

成功调用返回结果大于 0，出错返回结果为 -1，超时返回结果为 0。

poll poll 的功能与 select 类似，也是等待一组描述符中的一个成为就绪状态。

应用场景

1. select 应用场景 select 的 timeout 参数精度为微秒，而 poll 和 epoll 为毫秒，因此 select 更加适用于实时性要求比较高的场景，比如核反应堆的控制。

select 可移植性更好，几乎被所有主流平台所支持。

2. poll 应用场景 poll 没有最大描述符数量的限制，如果平台支持并且对实时性要求不高，应该使用 poll 而不是 select。

3. epoll 应用场景 只需要运行在 Linux 平台上，有大量的描述符需要同时轮询，并且这些连接最好是长连接。

需要同时监控小于 1000 个描述符，就没有必要使用 epoll，因为这个应用场景下并不能体现 epoll 的优势。

需要监控的描述符状态变化多，而且都是非常短暂的，也没有必要使用 epoll。因为 epoll 中的所有描述符都存储在内核中，造成每次需要对描述符的状态改变都需要通过 epoll_ctl() 进行系统调用，频繁系统调用降低效率。并且 epoll 的描述符存储在内核，不容易调试。