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；

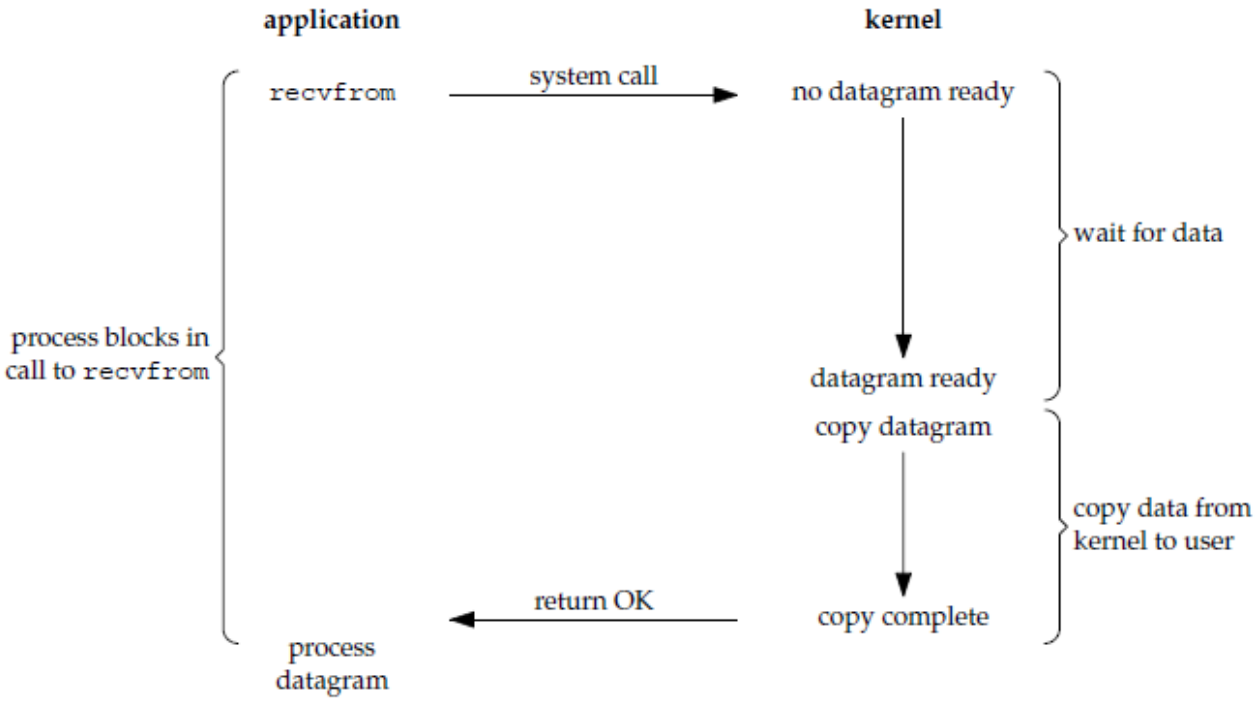
对于一个套接字上的输入操作，第一步通常涉及等待数据从网络中到达。当所等待数据到达时，它被复制到内核的某个缓冲区；第二步就是将数据从内核缓冲区复制到进程缓冲区；

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；

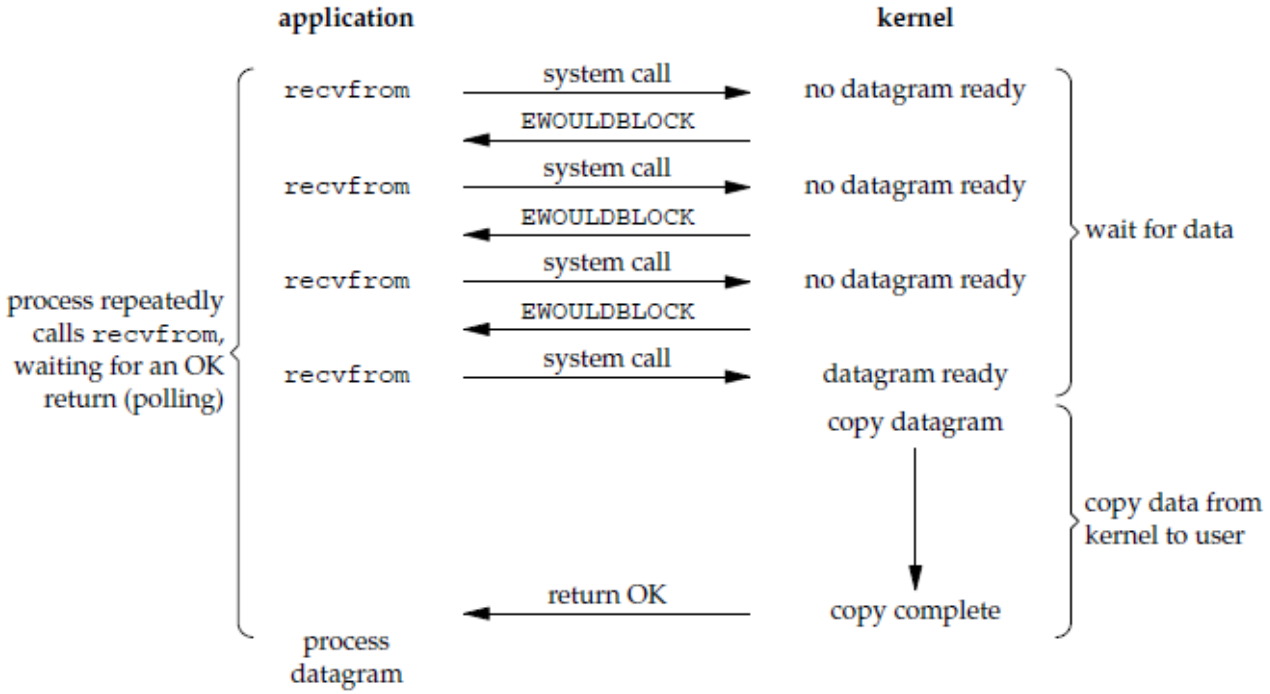
阻塞式IO 应用进程被阻塞，知道数据从内核缓冲区复制到进程缓冲区中才返回；

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；

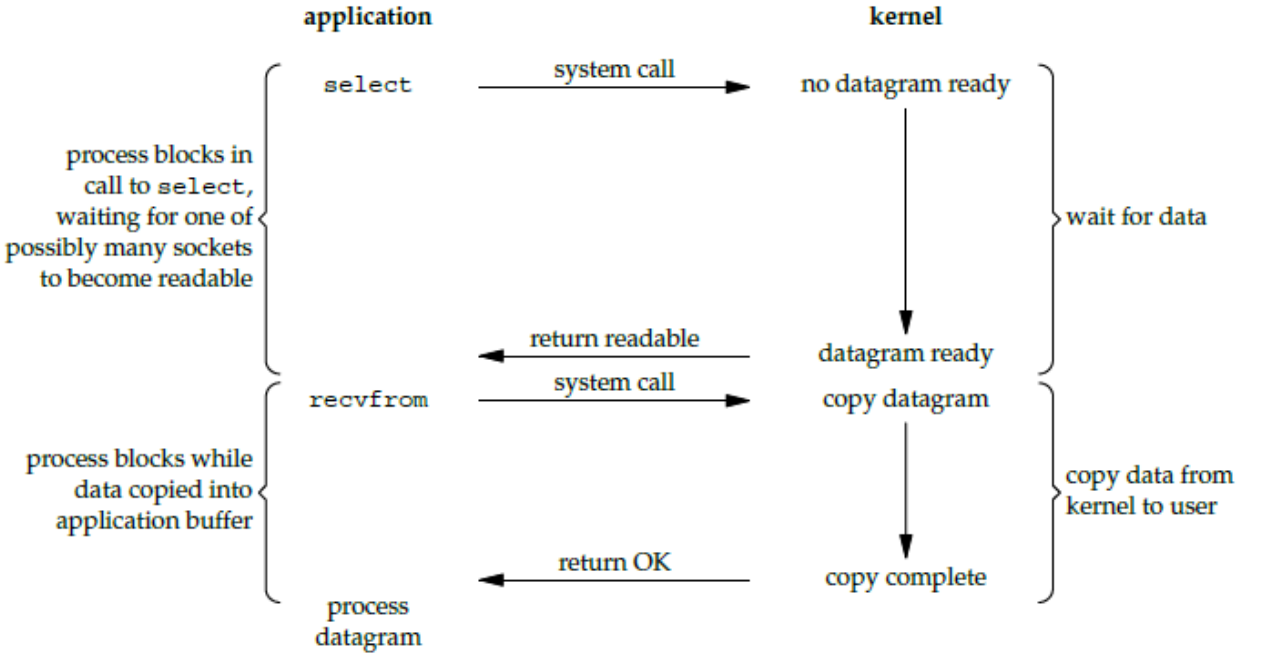
CPU要一直处理系统调用，因此这种模型的CPU利用率比较低；



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。这个过程会被阻塞，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。

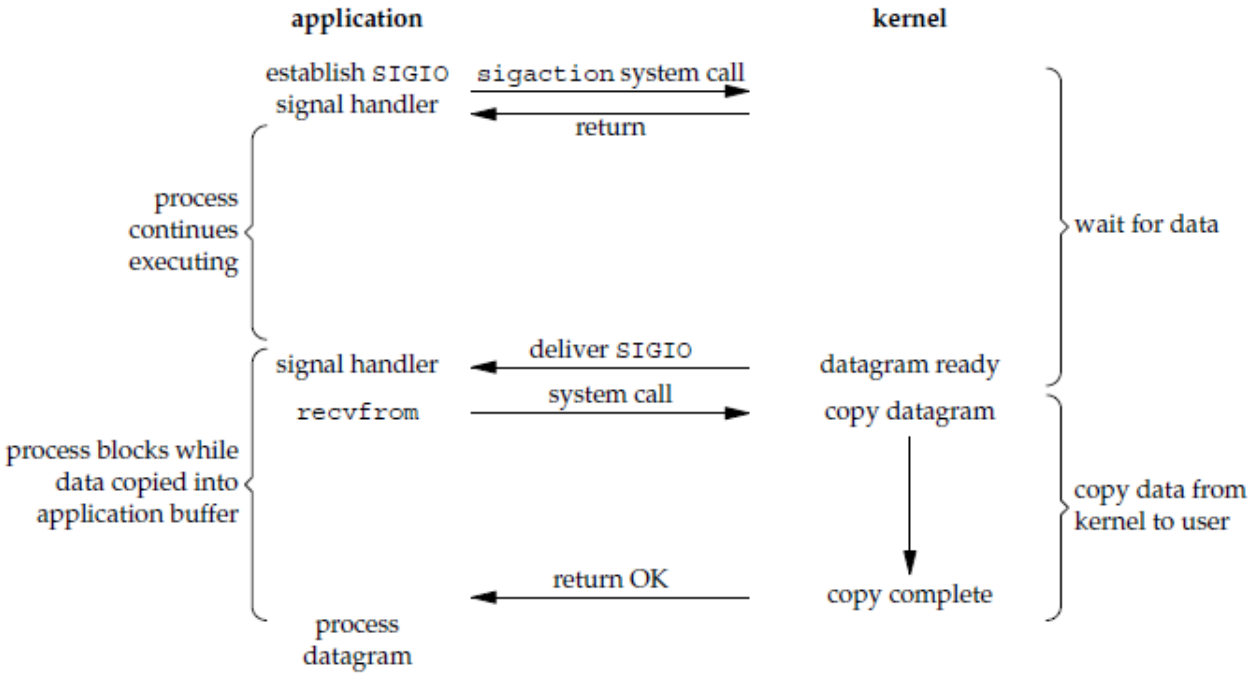
如果没有IO复用，每一个socket连接都需要创建一个线程去处理，有了IO复用就不会占用很多的进程创建和切换的开销。



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

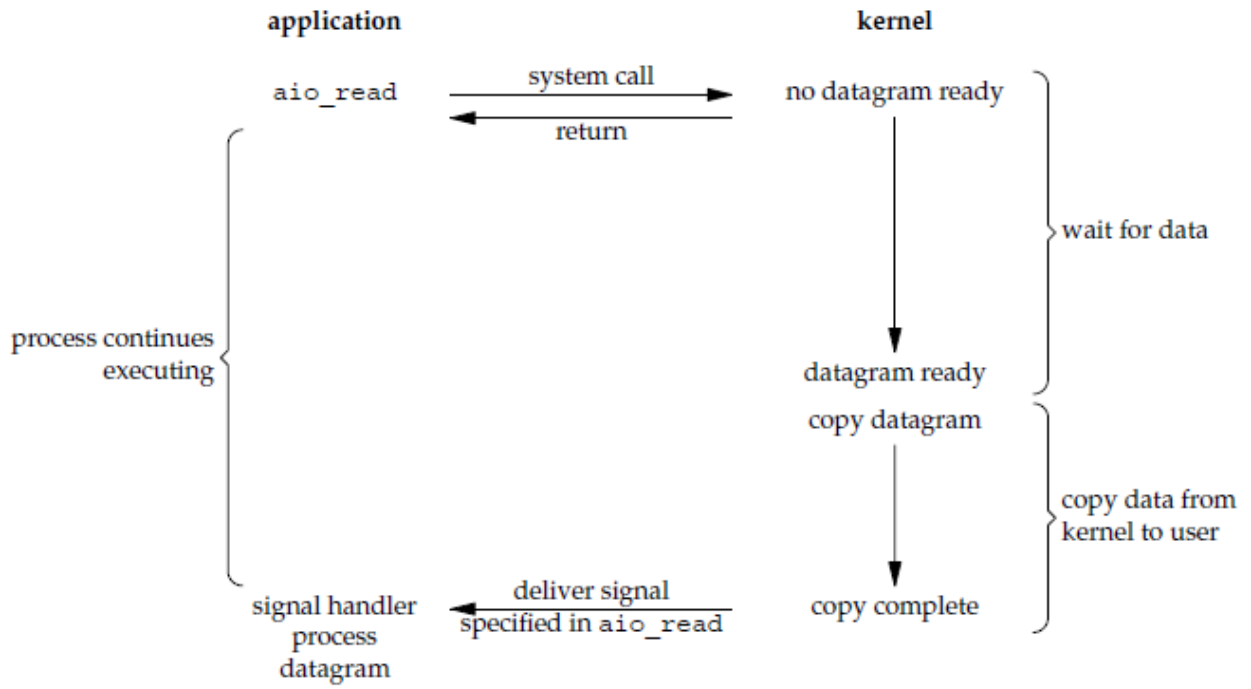
用recvfrom将数据从内核复制到应用进程中。

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；

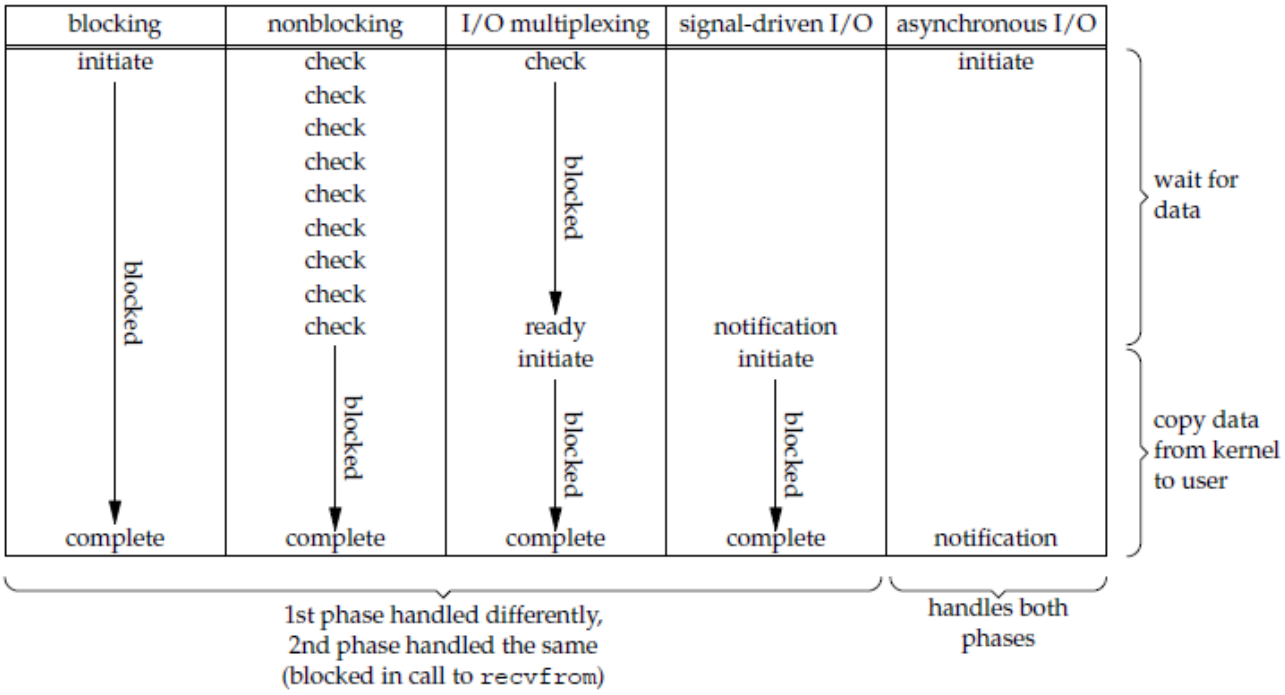


五大IO模型比较

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多路复用的具体实现，select出现最早，poll，再是epoll；

select

```
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
timeval *timeout);
```

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，fd_set使用数组实现，数组大小使用FD_SETSIZE定义；

timeout为超时参数，调用select会一直阻塞知道有描述符的事件或者等待事件超过timeout。

成功调用返回结果大于0，出错返回-1，超时返回结果为0；

poll

poll使用链表实现；

比较

1. 功能

select和poll的功能基本相同，但是在一些实现细节上不同；

1. select会修改描述符，而poll不会；
2. select的描述符类型使用数组实现，FD_SETSIZE大小默认为1024，因此只能监听1024个描述符。如果要监听更多描述符的话，需要修改FD_SETSIZE之后重新编译，而poll的描述符类型使用链表实现，没有描述符数量的限制；
3. poll提供了更多的事件类型，并且对描述符的利用率比select更高；
4. 如果一个线程对某个描述符调用了select或者poll，另一个线程关闭了该描述符，会导致调用结果不确定；

2. 速度

select和poll速度都比较慢；

1. select和poll每次调用都需要将全部的描述符从应用程序缓冲区复制到内核缓冲区；
2. select和poll的返回结果中没有声明那些描述符已经准备好，所以如果返回值大于0时，应用进程都需要使用轮询的方式找到IO完成的描述符；

3. 可移植性

几乎所有的系统都支持select，只用比较新的系统支持poll；

epoll

epoll_ctl()用于向内核注册新的描述符或者是改变某个文件描述符的状态。已注册的描述符在内核中会被维护在一颗红黑树上，通过回调函数内核会将IO准备好的描述符加入到一个链表中管理，进程调用epoll_wait()便可以得到事件完成的描述符；

从上面的描述可以看出，epoll只需要将描述符从进程缓冲区向内核缓冲区拷贝一次，并且进程不需要通过轮询来获得事件完成的描述符；

epoll仅适用于Linux OS

epoll比select和poll更加灵活并且没有描述符数量限制；

epoll对多线程编程更友好，一个线程调用了epoll_wait()另一个线程关闭了同一个描述符也不会产生像select和poll的不确定情况；

工作模式

epoll的描述符事件有两种触发模式：LT和ET。

1. LT模式

当epoll_wait()检测到描述符事件到达时，将此事件通知进程，进程不立即处理该事件，下次调用epoll_wait()会再次通知进程。是默认的一种模式，并且同时支持Blocking和No-Blocking。（epoll_wait会将在等待的文件描述符返给进程，但是进程处理不处理都行，但是线程LT的epoll_wait还会继续通知）

2. ET模式

和LT模式不同的是，通知之后进程必须立即处理事件，下次再调用epoll_wait时不会再等到事件到达地通知了。

很大程度上减少了epoll事件被重复触发的次数，因此效率比LT模式高，只支持No-Blocking，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死；

应用场景

1. select应用场景 select的timeout参数精确到1ns，而poll和epoll为1ms，因此select更加实用与实时性要求更高的场景；

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，如果平台支持并且实时度要求不高的话，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，并不能体现出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中所有的描述符都存储在内核中，造成每次都需要对描述符的状态改变都需要通过epoll_ctl进行系统调用，频繁的系统调用减低了效率，并且epoll的描述符存储在内核不容易调试；