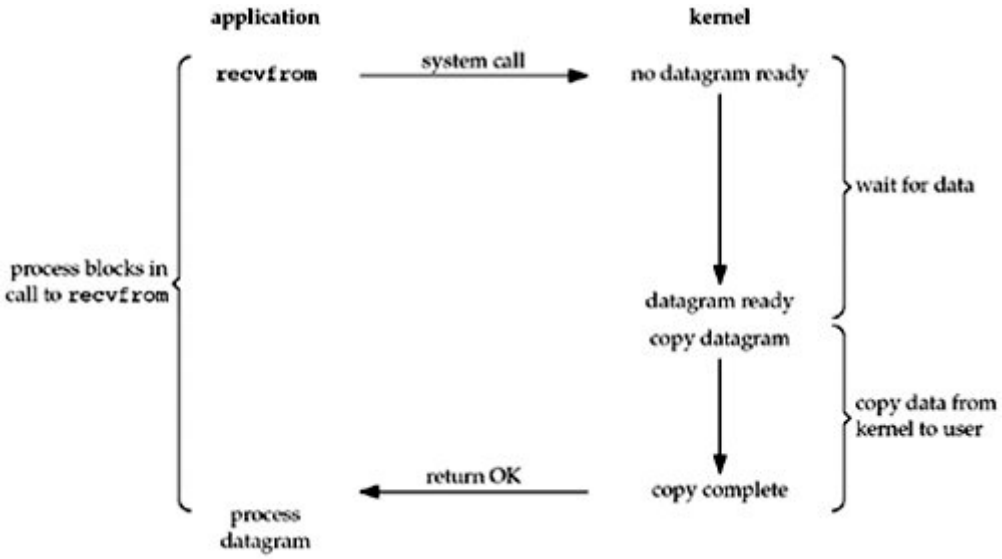
常见的I/O模型及其区别

首先，介绍几种常见的I/O模型及其区别，如下：《Unix网络编程》

- blocking I/O
- nonblocking I/O
- I/O multiplexing (`select` and `poll`)
- signal driven I/O (`SIGIO`)
- asynchronous I/O (the POSIX `aio_` functions)

读数据的例子

blocking I/O 这个不用多解释吧，阻塞套接字。下图是它调用过程的图示：



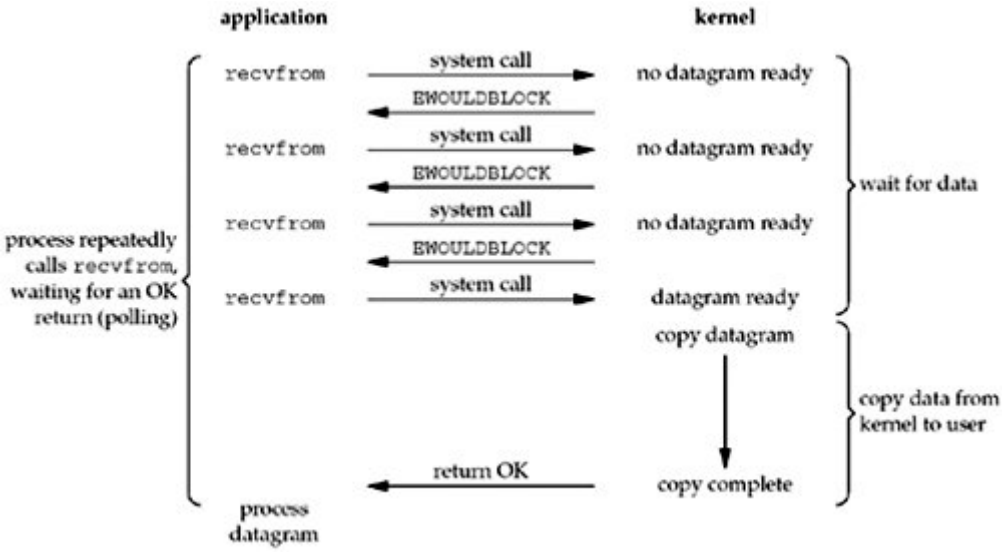
重点解释下上图，下面例子都会讲到。首先application调用 recvfrom()转入kernel，注意kernel有2个过程，wait for data和copy data from kernel to user。直到最后copy complete后，recvfrom()才返回。此过程一直是阻塞的。

示例代码：

while ((n=read(STDIN_FILENO, buf, BUFSIZ)) > 0) if (write (STDOUT_FILENO, buf, n) != n) err_sys (write error ") ;

从应用程序的角度来说，read 调用可能会延续很长时间。实际上，在内核执行读操作和其他工作时，应用程序的确会被阻塞，也就是说应用程序不能做其它事情了。

nonblocking I/O： 与blocking I/O对立的，非阻塞套接字，调用过程图如下：



可以看见，如果直接操作它，那就是个轮询。。直到内核缓冲区有数据。

对于一个给定的描述符有两种方法对其指定非阻塞I / O： (1) 如果是调用o p e n以获得该描述符，则可指定O _ N O N B L O C K标志。 (2) 对于已经打开的一个描述符，则可调用f c n t l打开O _ N O N B L O C K文件状态标志。 对于非阻塞I/O，read发现没有数据可读，则简单的返回-EAGAIN("try it agin")，而不是阻塞当前进程。来看一个非阻塞I/O的例子：

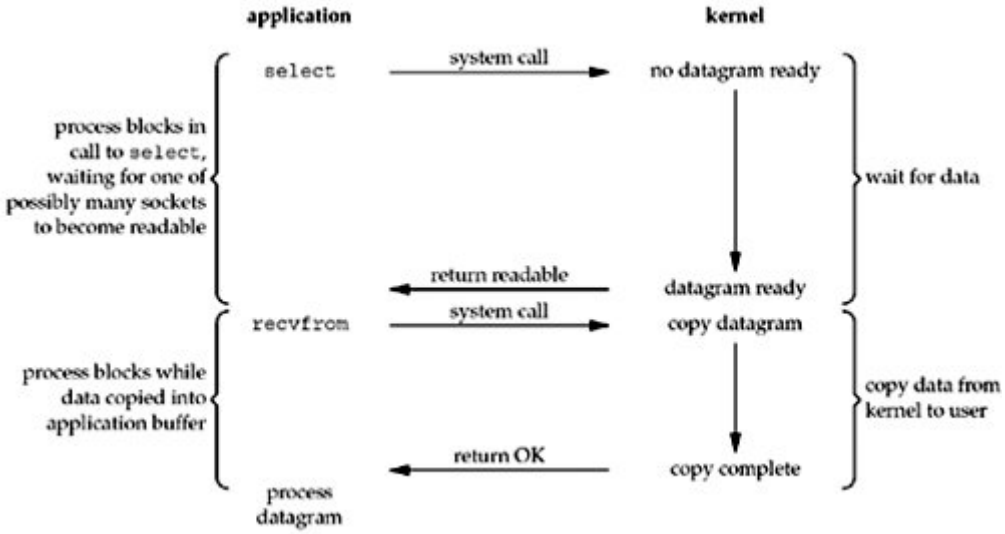
//nbtest.c

#include stdio.h #include unistd.h #include fcntl.h #include stdlib.h #include errno.h buffer[]; main(argc, argv) { delay , n, m ; (argc) delayatoi(argv[]); fcntl(, F_SETFL, fcntl(,F_GETFL) O_NONBLOCK); stdin fcntl(, F_SETFL, fcntl(,F_GETFL) O_NONBLOCK); stdout

() { n read(, buffer,); (n) m write(, buffer, n); ((n m) (errno EAGAIN)) ; sleep(delay); } perror(n stdin : stdout); exit(); }

我们用strace来跟踪一下程序执行的结果：可以清楚的看到read读取失败的情况。实际上，该方式需要应用程序以一种轮询的方式来实现数据读取，多次无谓的系统调用会加大系统开销，影响应整个系统的吞吐量。

I/O multiplexing (select and poll) 最常见的I/O复用模型，select。



select先阻塞，有活动套接字才返回。与**blocking I/O**相比，select会有两次系统调用，但是select能处理多个套接字。

Linux中，poll、epoll和select这三个函数可以用来实现 I/O多路转接。它们的本质上是相同的：每个允许一个进程来决定它是否可读或者写一个或多个文件而不阻塞. 这些调用也可阻塞进程直到任何一个给定集合的文件描述符可用来读或写. 因此, 它们常常用在必须使用多输入输出流的应用程序。 3.1、poll函数

应将events成员设置为如下所示值的一个或几个。通过这些值告诉内核我们对该描述符关心的是什么。返回时，内核设置revents成员，以说明对该描述符发生了什么事件。 （注意，poll没有更改events成员）。events和revents的取值：

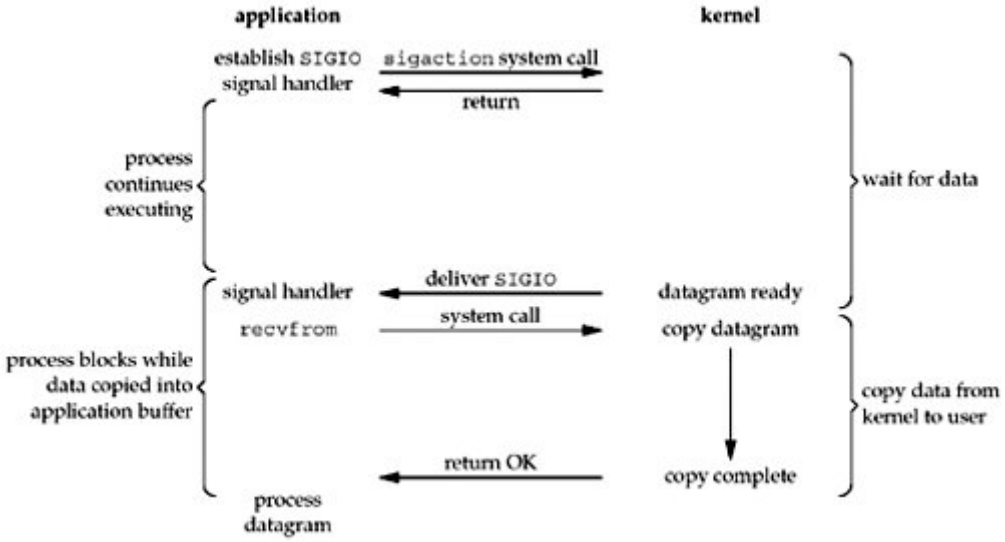
名　称	对events的输入	从revents得到的结果	说　明
POLLIN	•	•	可读除高优级外的数据，不阻塞
POLLRDNORM	•	•	可读普通（优先波段0 数据，不阻塞
POLLRDBAND	•	•	可读0优先波段数据，不阻塞
POLLPRI	•	•	可读高优先级数据，不阻塞
POLLOUT	•	•	可与普通数据，不阻塞
POLLWRNORM	•	•	与POLLOUT相同
POLLWRBAND	•	•	可写非0优先波段数据，不阻塞
POLLERR		•	已出错
POLLHUP		•	已挂起
POLLNVAL		•	此描述符并不引用一打开文件

头四行测试可读性，接着三行测试可写性，最后三行则是异常条件。最后三行是 由内核在返回时设置的。即使在 events字段中没有指定这三个值，如果相应条件发生，则在revents中也返回它们。当一个描述符被挂断后（POLLUP），就不能再写向该描述符。但是仍可能从该描述符读取到数据。 poll的最后一个参数说明我们想要等待多少时间。有三种不同的情形：• timeout == -1永远等待。常数INFTIM定义在<stropts.h>,其值通常是 - 1。当所指定 的描述符中的一个已准备好，或捕捉到一个信号则返回。如果捕捉到一个信号，则p o l l返回 - 1，errno设置为EINTR。 • timeout == 0 不等待。测试所有描述符并立即返回。这是得到很多个描述符的状态而不阻塞p o l l函数的轮询方法。 • timeout > 0 等待timeout毫秒。当指定的描述符之一已准备好，或指定的时间值已超过时立即返回。如果已超时但是还没有一个描述符准备好，则返回值是 0。 （如果系统不提供毫秒分辨率，则timeout值取整到最近的支持值）。

3.2、例子

select(或poll)的调用仍然会阻塞进程，与一般典型的I/O不一样的它是等待事件通知。但是它引入了超时机制，可以让应用程序有权力避免过长时间等待；另一方面，如果应用程序需要读写多个文件，该方式可以一显身手

signal driven I/O (SIGIO) 只有UNIX系统支持，感兴趣的课查阅相关资料

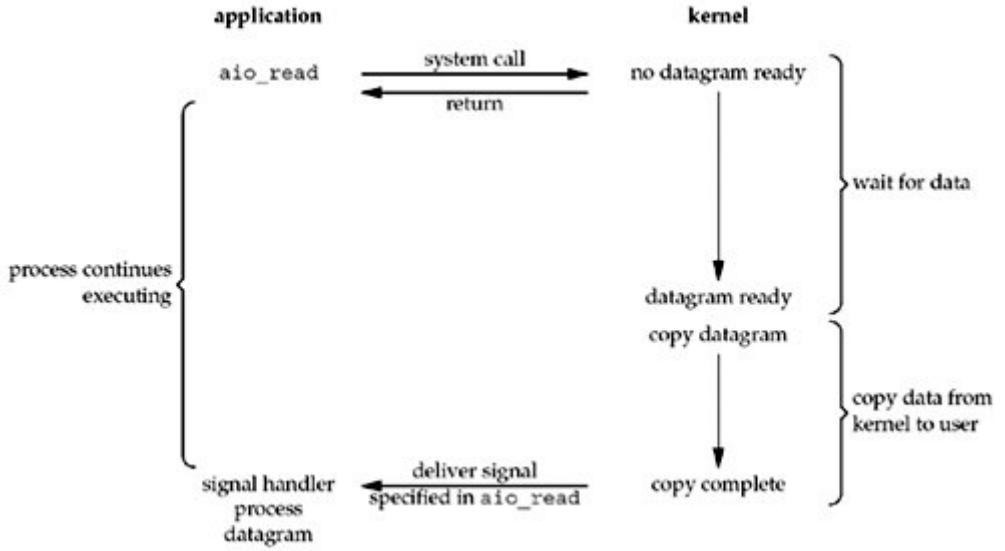


与**I/O multiplexing (select and poll)**相比，它的优势是，免去了select的阻塞与轮询，当有活跃套接字时，由注册的handler处理。

asynchronous I/O (the POSIX aio_functions)

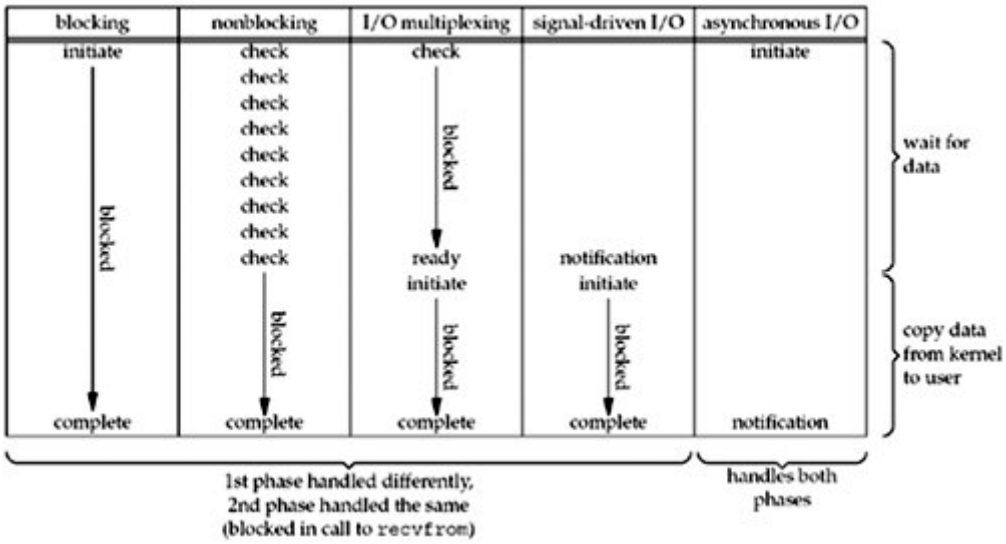
很少有LInux系统支持，windows的IOCP则是此模型

Linux 异步 I/O (AIO)，即异步非阻塞I/O，是 Linux 内核中提供的一个相当新的增强。它是 2.6 版本内核的一个标准特性，但是我们在 2.4 版本内核的补丁中也可以找到它。AIO 背后的基本思想是允许进程发起很多 I/O 操作，而不用阻塞或等待任何操作完成。稍后或在接收到 I/O 操作完成的通知时，进程就可以检索 I/O 操作的结果。



完全异步的I/O复用机制，因为纵观上面其它四种模型，至少都会在由kernel copy data to appliction时阻塞。而该模型是当copy完成后才通知application，可见是纯异步的。好像只有windows的完成端口是这个模型，效率也很出色。

下面是以上五种模型的比较



可以看出，越往后，阻塞越少，理论上效率也是最优。

==========分割线==========

5种模型的比较清晰了，剩下的就是把select,epoll,iocp,kqueue按号入座那就OK了。

select和iocp分别对应第3种与第5种模型，那么epoll与kqueue呢？其实也于select属于同一种模型，只是更高级一些，可以看作有了第4种模型的某些特性，如callback机制。

那么，为什么epoll,kqueue比select高级?

答案是，他们无**轮询**。因为他们用callback取代了。想想看，当套接字比较多的时候，每次select()都要通过遍历FD_SETSIZE个Socket来完成调度,不管哪个Socket是活跃的,都遍历一遍。这会浪费很多CPU时间。如果能给套接字注册某个回调函数，当他们活跃时，自动完成相关操作，那就避免了轮询，这正是epoll与kqueue做的。

windows or Linux （IOCP or kqueue/epoll）?

诚然，Windows的IOCP非常出色，目前很少有支持**asynchronous I/O**的系统，但是由于其系统本身的局限性，大型服务器还是在UNIX下。而且正如上面所述，kqueue/epoll 与 IOCP相比，就是多了一层从内核copy数据到应用层的阻塞，从而不能算作**asynchronous I/O类。**但是，这层小小的阻塞无足轻重，kqueue与epoll已经做得很优秀了。

提供一致的接口，IO Design Patterns

实际上，不管是哪种模型，都可以抽象一层出来，提供一致的接口，广为人知的有ACE,Libevent这些，他们都是跨平台的，而且他们自动选择最优的I/O复用机制，用户只需调用接口即可。说到这里又得说说2个设计模式，**Reactor** and **Proactor。**有一篇经典文章http://www.artima.com/articles/io_design_patterns.html值得阅读，Libevent是**Reactor**模型，ACE提供**Proactor**模型。实际都是对各种I/O复用机制的封装。

Java nio包是什么I/O机制?

我曾天真的认为java nio封装的是IOCP。。现在可以确定，目前的java本质是select()模型，可以检查/jre/bin/nio.dll得知。至于java服务器为什么效率还不错。。我也不得而知，可能是设计得比较好吧。。-_-。

==========分割线==========

JAVA运用EPoll来进行NIO处理的方法

Event Poll

Epoll是Linux内核为处理大批量句柄而作了改进的poll。要使用epoll只需要这三个系统调用：epoll_create(2)， epoll_ctl(2)， epoll_wait(2)。它是在2.5.44内核中被引进的(epoll(4) is a new API introduced in Linux kernel 2.5.44)，在2.6内核中得到广泛应用，例如LightHttpd

JDK 6.0 以及JDK 5.0 update 9 的 nio支持epoll （**仅限 Linux 系统**），对并发idle connection会有大幅度的性能提升，这就是很多网络服务器应用程序需要的。

启用的方法如下：

```
-Djava.nio.channels.spi.SelectorProvider=sun.nio.ch.EPollSelectorProvider
```

例如在 Linux 下运行的 Tomcat 使用 NIO Connector ，那么启用 epoll 对性能的提升会有帮助。

而 Tomcat 要启用这个选项的做法是在 catalina.sh 的开头加入下面这一行

CATALINA_OPTS='-Djava.nio.channels.spi.SelectorProvider=sun.nio.ch.EPollSelectorProvider'

一 epoll - epoll的优点

- 支持一个进程打开大数目的socket描述符(FD)

select 最不能忍受的是一个进程所打开的FD是有一定限制的，由FD_SETSIZE设置，默认值是2048。对于那些需要支持的上万连接数目的IM服务器来说显然太少了。这时候你一是可以选择修改这个宏然后重新编译内核，不过资料也同时指出这样会带来网络效率的下降，二是可以选择多进程的解决方案(传统的Apache方案)，不过虽然linux上面创建进程的代价比较小，但仍旧是不可忽视的，加上进程间数据同步远比不上线程间同步的高效，所以也不是一种完美的方案。不过 epoll则没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。

- IO效率不随FD数目增加而线性下降

传统的select/poll另一个致命弱点就是当你拥有一个很大的socket集合，不过由于网络延时，任一时间只有部分的socket是"活跃"的，但是select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是epoll不存在这个问题，它只会对"活跃"的socket进行操作---这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。那么，只有"活跃"的socket才会主动的去调用 callback函数，其他idle状态socket则不会，在这点上，epoll实现了一个"伪"AIO，因为这时候推动力在os内核。在一些 benchmark中，如果所有的socket基本上都是活跃的---比如一个高速LAN环境，epoll并不比select/poll有什么效率，相反，如果过多使用epoll_ctl,效率相比还有稍微的下降。但是一旦使用idle connections模拟WAN环境,epoll的效率就远在select/poll之上了。

- 使用mmap加速内核与用户空间的消息传递。

这点实际上涉及到epoll的具体实现了。无论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就很重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。而如果你想我一样从2.5内核就关注epoll的话，一定不会忘记手工 mmap这一步的。

- 内核微调

这一点其实不算epoll的优点了，而是整个linux平台的优点。也许你可以怀疑linux平台，但是你无法回避linux平台赋予你微调内核的能力。比如，内核TCP/IP协议栈使用内存池管理sk_buff结构，那么可以在运行时期动态调整这个内存pool(skb_head_pool)的大小--- 通过echo XXXX>/proc/sys/net/core/hot_list_length完成。再比如listen函数的第2个参数(TCP完成3次握手的数据包队列长度)，也可以根据你平台内存大小动态调整。更甚至在一个数据包面数目巨大但同时每个数据包本身大小却很小的特殊系统上尝试最新的NAPI网卡驱动架构。

二 epoll - epoll的使用

令人高兴的是，2.6内核的epoll比其2.5开发版本的/dev/epoll简洁了许多，所以，大部分情况下，强大的东西往往是简单的。唯一有点麻烦是epoll有2种工作方式:LT和ET。

LT(level triggered)是缺省的工作方式，并且同时支持block和no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。传统的select/poll都是这种模型的代表.

ET (edge-triggered)是高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了(比如，你在发送，接收或者接收请求，或者发送接收的数据少于一定量时导致了一个EWOULDBLOCK 错误）。但是请注意，如果一直不对这个fd作IO操作(从而导致它再次变成未就绪)，内核不会发送更多的通知(only once)，不过在TCP协议中，ET模式的加速效用仍需要更多的benchmark确认。

epoll只有epoll_create,epoll_ctl,epoll_wait 3个系统调用， 具体用法请参考http://www.xmailserver.org/linux-patches/nio-improve.html 在http://www.kegel.com/rn/有一个完整的例子。

epoll 或者 kqueue 的原理是什么?

首先我们来定义流的概念，一个流可以是文件，socket，pipe等等可以进行I/O操作的内核对象。

但是流中还没有数据 很明显一般人不会用第二种做法，不仅显很无脑，浪费话费不说，还占用了快递员大量的时间。大部分程序也不会用第二种做法，因为第一种方法经济而简单，经济是指消耗很少的CPU时间，如果线程睡眠了，就掉出了系统的调度队列，暂时不会去瓜分CPU宝贵的时间片了。

这四个情形涵盖了四个I/O事件，缓冲区满，缓冲区空，缓冲区非空，缓冲区非满（注都是说的内核缓冲区，且这四个术语都是我生造的，仅为解释其原理而造）。这四个I/O事件是进行阻塞同步的根本。（如果不能理解"同步"是什么概念，请学习操作系统的锁，信号量，条件变量等任务同步方面的相关知识）。

为了避免CPU空转，可以引进了一个代理（一开始有一位叫做select的代理，后来又有一位叫做poll的代理，不过两者的本质是一样的）。这个代理比较厉害，可以同时观察许多流的I/O事件，在空闲的时候，，当有一个或多个流有I/O事件时，就从阻塞态中醒来，于是我们的程序就会轮询一遍所有的流（于是我们可以把"忙"字去掉了）。代码长这样:while true {select(streams[])for i in streams[] {if i has dataread until unavailable}}于是，如果没有I/O事件产生，我们的程序就会阻塞在select处。但是依然有个问题，我们从select那里仅仅知道了，有I/O事件发生了，但却并不知道是那几个流（可能有一个，多个，甚至全部），我们只能所有流，找出能读出数据，或者写入数据的流，对他们进行操作。但是使用select，我们有O(n)的无差别轮询复杂度，同时处理的流越多，每一次无差别轮询时间就越长。再次**说了这么多，终于能好好解释epoll了**epoll可以理解为event poll，不同于忙轮询和无差别轮询，epoll之会把哪个流发生了怎样的I/O事件通知我们。此时我们对这些流的操作都是有意义的。（复杂度降低到了O(k)，k为产生I/O事件的流的个数，也有认为O(1)的[更新 1]）在讨论epoll的实现细节之前，先把epoll的相关操作列出[更新 2]:

- epoll_create 创建一个epoll对象，一般epollfd = epoll_create()
- epoll_ctl （epoll_add/epoll_del的合体），往epoll对象中增加/删除某一个流的某一个事件 比如 epoll_ctl(epollfd, EPOLL_CTL_ADD, socket, EPOLLIN);//有缓冲区内有数据时epoll_wait返回 epoll_ctl(epollfd, EPOLL_CTL_DEL, socket, EPOLLOUT);//缓冲区可写入时epoll_wait返回
- epoll_wait(epollfd,...)等待直到注册的事件发生

（注：当对一个非阻塞流的读写发生缓冲区满或缓冲区空，write/read会返回-1，并设置errno=EAGAIN。而epoll只关心缓冲区非满和缓冲区非空事件）。

epoll的原理就是： 你把要监控读写的文件交给内核（epoll_add) 设置你关心的事件（epoll_ctl），比如读事件 然后等（epoll_wait），此时，如果没有哪个文件有你关心的事件，则休眠，直到有事件，被唤醒 然后返回那些事件

==========分割线==========

总结一些重点：

1. 只有IOCP是asynchronous I/O，其他机制或多或少都会有一点阻塞。
2. select低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善
3. epoll, kqueue是Reacor模式，IOCP是Proactor模式。
4. Java nio 包是select ??模型
5. Java nio2包是epoll(Linux),windows(IOCP)

==========分割线==========

ACE:

“重量级的C++ I/O框架，用面向对象实现了一些I/O策略和其它有用的东西，特别是它的Reactor是用OO方式处理非阻塞I/O，而Proactor是用OO方式处理异步I/O的(In particular, his Reactor is an OO way of doing nonblocking I/O, and Proactor is an OO way of doing asynchronous I/O).”

从很多实际使用来看，ACE是一个很值得学习的网络框架，但由于它过于重量级，导致使用起来并不方便。

ACE中提出了两种网络模式：Proactor和Reactor。

ASIO:

“C++的I/O框架，逐渐成为Boost库的一部分。it’s like ACE updated for the STL era。”

支持select、epoll、IOCP等IO模型；

libevent:

由Niels Provos用C编写的一个轻量级的I/O框架。它支持kqueue和select、poll和epoll。

1.4.11版还不支持windows的IOCP，但已经有很多开发者自己修改源码，把IOCP合并进去。

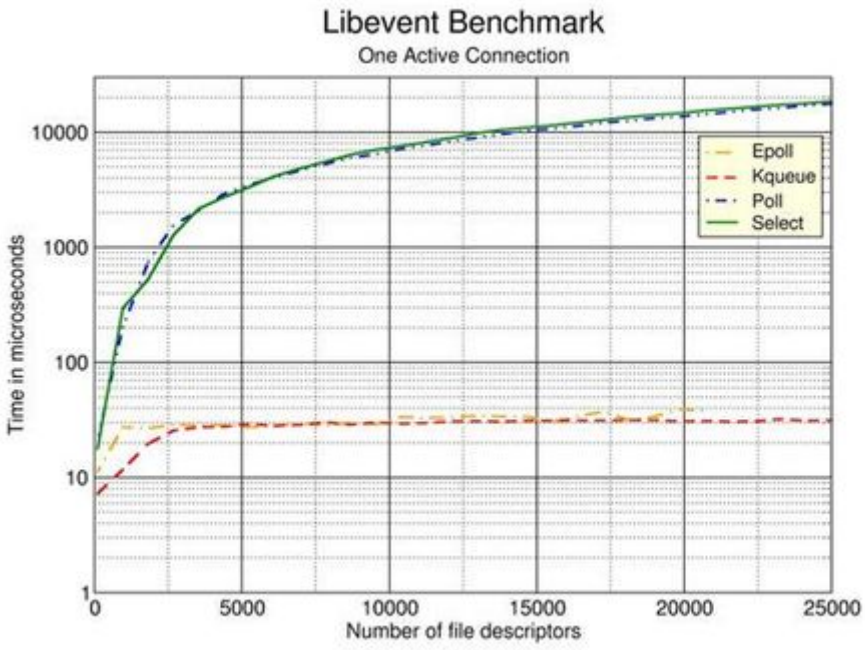
==========分割线==========

C10K问题与解决之道

一台服务器如何同时处理一万个以上的客户端，这就是著名的C10K问题；这个问题曾经困扰过很多服务器的架构师，但这种困扰随着时间的推移早已成为了过去。

网络的众多模型中，有些适合简单处理，有些适合复杂应用，而C10K问题考验的则是网络框架的并发和大连接处理能力，异步I/O虽然也是为并发和大连接设计，但目前aio并不支持socket，所以目前最适合的解决方案是I/O复用。

I/O复用最初在解决C10K问题时也并不出色，随着连接数的增加，处理单个socket请求所花费的事件也迅速增加，但kqueue和epoll的出世后，I/O复用成为了C10K问题的首选方案，下图是libevent给出的测试数据：



从图中可以看出，select、poll随着需要处理的连接数（横轴）增多，处理单个连接的时间明显变长，而epoll和kqueue的表现则非常优秀。

网络编程：Reactor与Proactor的概念

1、标准定义

两种I/O多路复用模式：Reactor和Proactor

一般地,I/O多路复用机制都依赖于一个事件多路分离器(Event Demultiplexer)。分离器对象可将来自事件源的I/O事件分离出来，并分发到对应的read/write事件处理器(Event Handler)。开发人员预先注册需要处理的事件及其事件处理器（或回调函数）；事件分离器负责将请求事件传递给事件处理器。两个与事件分离器有关的模式是Reactor和Proactor。Reactor模式采用同步IO，而Proactor采用异步IO。

在Reactor中，事件分离器负责等待文件描述符或socket为读写操作准备就绪，然后将就绪事件传递给对应的处理器，最后由处理器负责完成实际的读写工作。

而在Proactor模式中，处理器--或者兼任处理器的事件分离器，只负责发起异步读写操作。IO操作本身由操作系统来完成。传递给操作系统的参数需要包括用户定义的数据缓冲区地址和数据大小，操作系统才能从中得到写出操作所需数据，或写入从socket读到的数据。事件分离器捕获IO操作完成事件，然后将事件传递给对应处理器。比如，在windows上，处理器发起一个异步IO操作，再由事件分离器等待IOCompletion事件。典型的异步模式实现，都建立在操作系统支持异步API的基础之上，我们将这种实现称为“系统级”异步或“真”异步，因为应用程序完全依赖操作系统执行真正的IO工作。

举个例子，将有助于理解Reactor与Proactor二者的差异，以读操作为例（类操作类似）。 **在Reactor中实现读：**

- 注册读就绪事件和相应的事件处理器 - 事件分离器等待事件 - 事件到来，激活分离器，分离器调用事件对应的处理器。 - 事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。 **在Proactor中实现读：**

- 处理器发起异步读操作（注意：操作系统必须支持异步IO）。在这种情况下，处理器无视IO就绪事件，它关注的是完成事件。 - 事件分离器等待操作完成事件 - 在分离器等待过程中，操作系统利用并行的内核线程执行实际的读操作，并将结果数据存入用户自定义缓冲区，最后通知事件分离器读操作完成。 - 事件分离器呼唤处理器。 - 事件处理器处理用户自定义缓冲区中的数据，然后启动一个新的异步操作，并将控制权返回事件分离器。

可以看出，两个模式的相同点，都是对某个IO事件的事件通知(即告诉某个模块，这个IO操作可以进行或已经完成)。在结构上，两者也有相同点：demultiplexor负责提交IO操作(异步)、查询设备是否可操作(同步)，然后当条件满足时，就回调handler；不同点在于，异步情况下(Proactor)，当回调handler时，表示IO操作已经完成；同步情况下(Reactor)，回调handler时，表示IO设备可以进行某个操作(can read or can write)。

2、通俗理解

使用Proactor框架和Reactor框架都可以极大的简化网络应用的开发，但它们的重点却不同。

Reactor框架中用户定义的操作是在实际操作之前调用的。比如你定义了操作是要向一个SOCKET写数据，那么当该SOCKET可以接收数据的时候，你的操作就会被调用；而Proactor框架中用户定义的操作是在实际操作之后调用的。比如你定义了一个操作要显示从SOCKET中读入的数据，那么当读操作完成以后，你的操作才会被调用。

Proactor和Reactor都是并发编程中的设计模式。在我看来，他们都是用于派发/分离IO操作事件的。这里所谓的IO事件也就是诸如read/write的IO操作。"派发/分离"就是将单独的IO事件通知到上层模块。两个模式不同的地方在于，Proactor用于异步IO，而Reactor用于同步IO。

3、备注

其实这两种模式在ACE(网络库)中都有体现；如果要了解这两种模式，可以参考ACE的源码，ACE是开源的网络框架，非常值得一学。