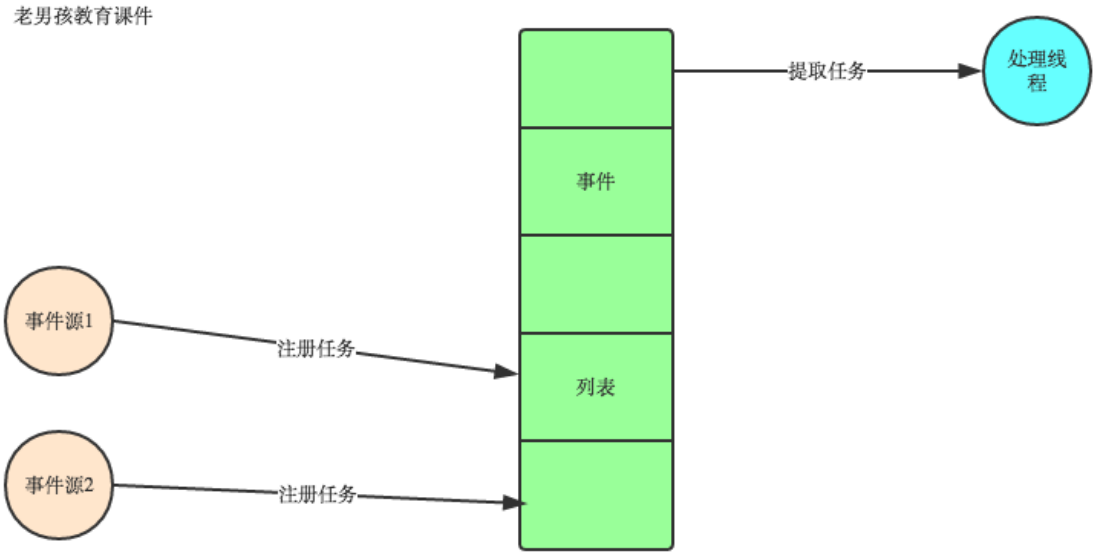
# 事件驱动和异步 IO

通常，我们写服务器处理模型的程序时，有以下几种模型：
（1）每收到一个请求，创建一个新的进程，来处理该请求；
（2）每收到一个请求，创建一个新的线程，来处理该请求；
（3）每收到一个请求，放入一个事件列表，让主进程通过非阻塞 I/O 方式来处理请求
上面的几种方式，各有千秋，
第（1）中方法，由于创建新的进程的开销比较大，所以，会导致服务器性能比较差,但实现比较简单。
第（2）种方式，由于要涉及到线程的同步，有可能会面临死锁等问题。
第（3）种方式，在写应用程序代码时，逻辑比前面两种都复杂。
综合考虑各方面因素，一般普遍认为第（3）种方式是大多数网络服务器采用的方式

## 看图说话讲事件驱动模型

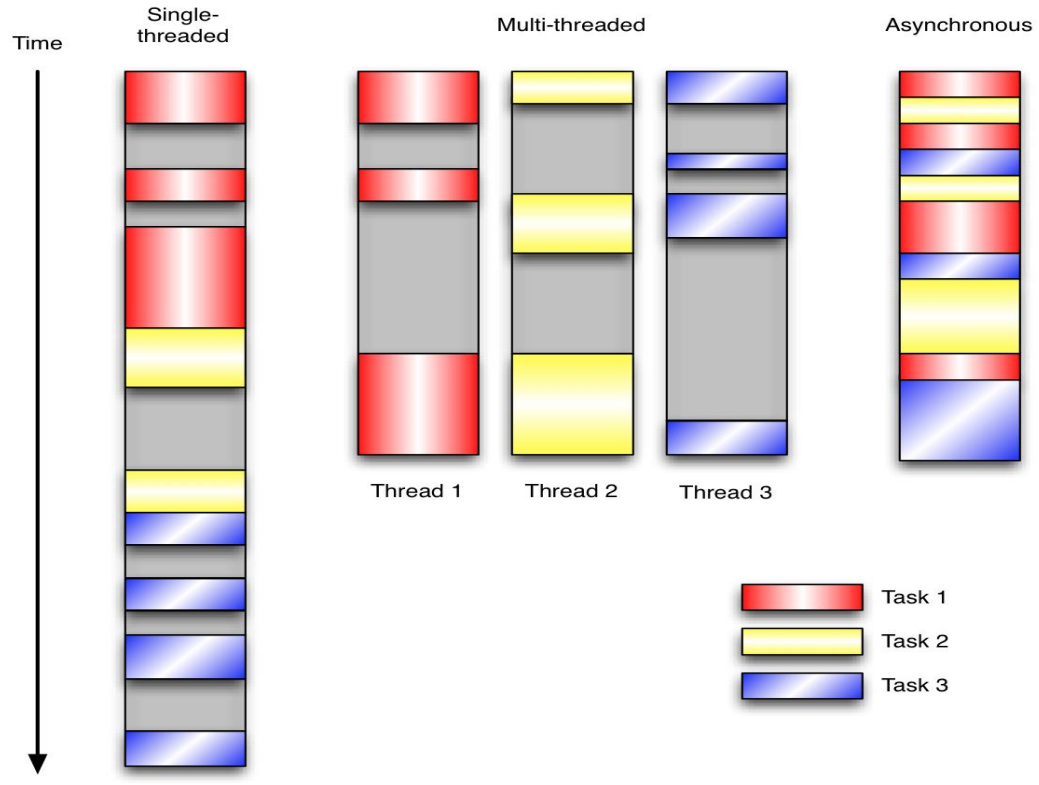
在 UI 编程中，常常要对鼠标点击进行相应，首先如何获得鼠标点击呢？
**方式一：创建一个线程，该线程一直循环检测是否有鼠标点击，那么这个方式有以下几个缺点：**
1．CPU 资源浪费，可能鼠标点击的频率非常小，但是扫描线程还是会一直循环检测，这会造成很多的 CPU 资源浪费；如果扫描鼠标点击的接口是阻塞的呢？
2．如果是堵塞的，又会出现下面这样的问题，如果我们不但要扫描鼠标点击，还要扫描键盘是否按下，由于扫描鼠标时被堵塞了，那么可能永远不会去扫描键盘；
3．如果一个循环需要扫描的设备非常多，这又会引来响应时间的问题；
所以，该方式是非常不好的。

**方式二：就是事件驱动模型**
目前大部分的 UI 编程都是事件驱动模型，如很多 UI 平台都会提供 onClick()事件，这个事件就代表鼠标按下事件。事件驱动模型大体思路如下：
1．有一个事件（消息）队列；
2．鼠标按下时，往这个队列中增加一个点击事件（消息）；
3．有个循环，不断从队列取出事件，根据不同的事件，调用不同的函数，如 onClick()、onKeyDown()等；
4．事件（消息）一般都各自保存各自的处理函数指针，这样，每个消息都有独立的处理函数；




事件驱动编程是一种编程范式，这里程序的执行流由外部事件来决定。它的特点是包含一个事件循环，当外部事件发生时使用回调机制来触发相应的处理。另外两种常见的编程范式是（单线程）同步以及多线程编程。

让我们用例子来比较和对比一下单线程、多线程以及事件驱动编程模型。下图展示了随着时间的推移，这三种模式下程序所做的工作。这个程序有 3 个任务需要完成，每个任务都在等待 I/O 操作时阻塞自身。阻塞在 I/O 操作上所花费的时间已经用灰色框标示出来了。



在单线程同步模型中，任务按照顺序执行。如果某个任务因为 I/O 而阻塞，其他所有的任务都必须等待，直到它完成之后它们才能依次执行。这种明确的执行顺序和串行化处理的行为是很容易推断得出的。如果任务之间并没有互相依赖的关系，但仍然需要互相等待的话这就使得程序不必要的降低了运行速度。

在多线程版本中，这 3 个任务分别在独立的线程中执行。这些线程由操作系统来管理，在多处理器系统上可以并行处理，或者在单处理器系统上交错执行。这使得当某个线程阻塞在某个资源的同时其他线程得以继续执行。与完成类似功能的同步程序相比，这种方式更有效率，但程序员必须写代码来保护共享资源，防止其被多个线程同时访问。多线程程序更加难以推断，因为这类程序不得不通过线程同步机制如锁、可重入函数、线程局部存储或者其他机制来处理线程安全问题，如果实现不当就会导致出现微妙且令人痛不欲生的 bug。

在事件驱动版本的程序中，3 个任务交错执行，但仍然在一个单独的线程控制中。当处理 I/O 或者其他昂贵的操作时，注册一个回调到事件循环中，然后当 I/O 操作完成时继续执行。回调描述了该如何处理某个事件。事件循环轮询所有的事件，当事件到来时将它们分配给等待处理事件的回调函数。这种方式让程序尽可能的得以执行而不需要用到额外的线程。事件驱动型程序比多线程程序更容易推断出行为，因为程序员不需要关心线程安全问题。

当我们面对如下的环境时，事件驱动模型通常是一个好的选择：

1. 程序中有许多任务，而且...
2. 任务之间高度独立（因此它们不需要互相通信，或者等待彼此）而且...
3. 在等待事件到来时，某些任务会阻塞。

当应用程序需要在任务间共享可变的数据时，这也是一个不错的选择，因为这里不需要采用同步处理。

网络应用程序通常都有上述这些特点，这使得它们能够很好的契合事件驱动编程模型。

**此处要提出一个问题，就是，上面的事件驱动模型中，只要一遇到 IO 就注册一个事件，然后主程序就可以继续干其它的事情了，只到 io 处理完毕后，继续恢复之前中断的任务，这本质上是怎么实现的呢？哈哈，下面我们就来一起揭开这神秘的面纱。。。。**

最初的问题：怎么确定 IO 操作完了切回去呢？**通过回调函数**

# 二、Select\Poll\Epoll 异步 IOIO => 多路复用

前面是用协程实现的 IO 阻塞自动切换，那么协程又是怎么实现的，在原理是是怎么实现的。如何去实现事件驱动的情况下 IO 的自动阻塞的切换，这个学名叫什么呢？ => IO 多路复用

比如 socketserver，多个客户端连接，单线程下实现并发效果，就叫多路复用。

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。

# 1、阻塞 IO，非阻塞 IO，同步 IO,异步 IO 介绍

在进行解释之前，首先要说明几个概念：

用户空间和内核空间

进程切换

进程的阻塞

文件描述符

缓存 I/O

**用户空间与内核空间**

现在操作系统都是采用虚拟存储器，那么对 32 位操作系统而言，它的寻址空间（虚拟存储空间）为 4G（2 的 32 次方）。

操作系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底层硬件设备的所有权限。

为了保证用户进程不能直接操作内核（kernel），保证内核的安全，操心系统将虚拟空间划分为两部分，一部分为内核空间，一部分为用户空间。

针对 linux 操作系统而言，将最高的 1G 字节（从虚拟地址 0xC0000000 到 0xFFFFFFFF），供内核使用，称为内核空间，而将较低的 3G 字节（从虚拟地址 0x00000000 到 0xBFFFFFFF），供各个进程使用，称为用户空间。

**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.保存处理机上下文，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.恢复处理机上下文。

注：总而言之就是很耗资源 **进程的阻塞**

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，也就是说，**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。**用户空间没法直接访问内核空间的，内核态到用户态的数据拷贝

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。

# 2、IO 模式

刚才说了，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。所以说，当一个 read 操作发生时，它会经历两个阶段：

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)

正式因为这两个阶段，linux 系统产生了下面五种网络模式的方案。 阻塞 I/O（blocking IO）

非阻塞 I/O（nonblocking IO）

I/O 多路复用（ IO multiplexing）

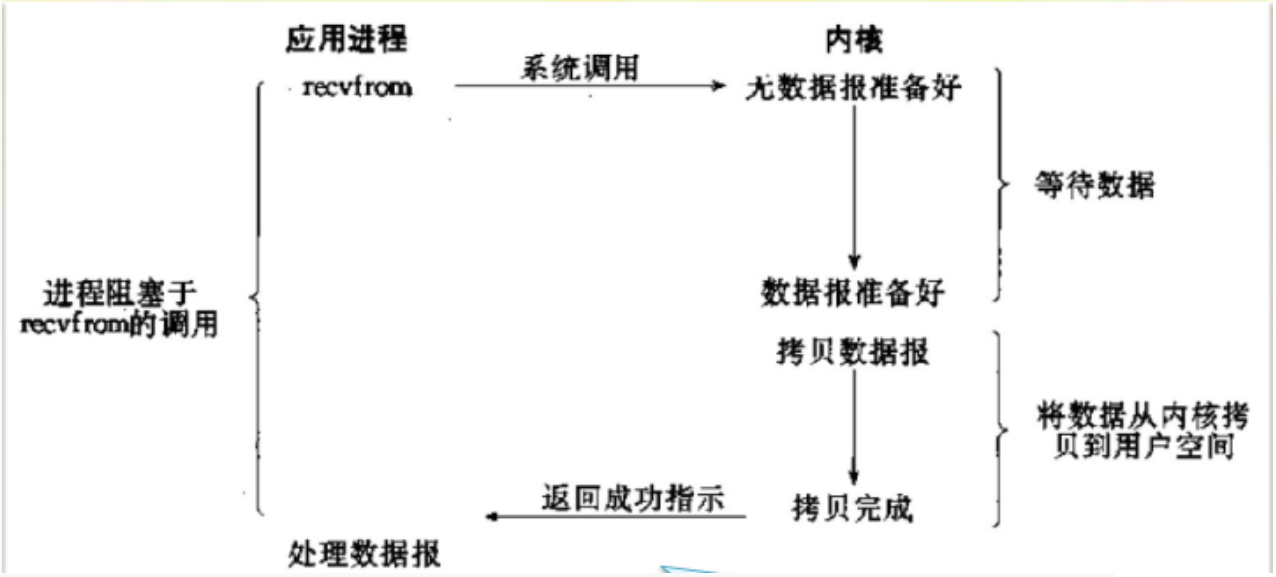
信号驱动 I/O（ signal driven IO）

异步 I/O（asynchronous IO）

注：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。

## 1）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：



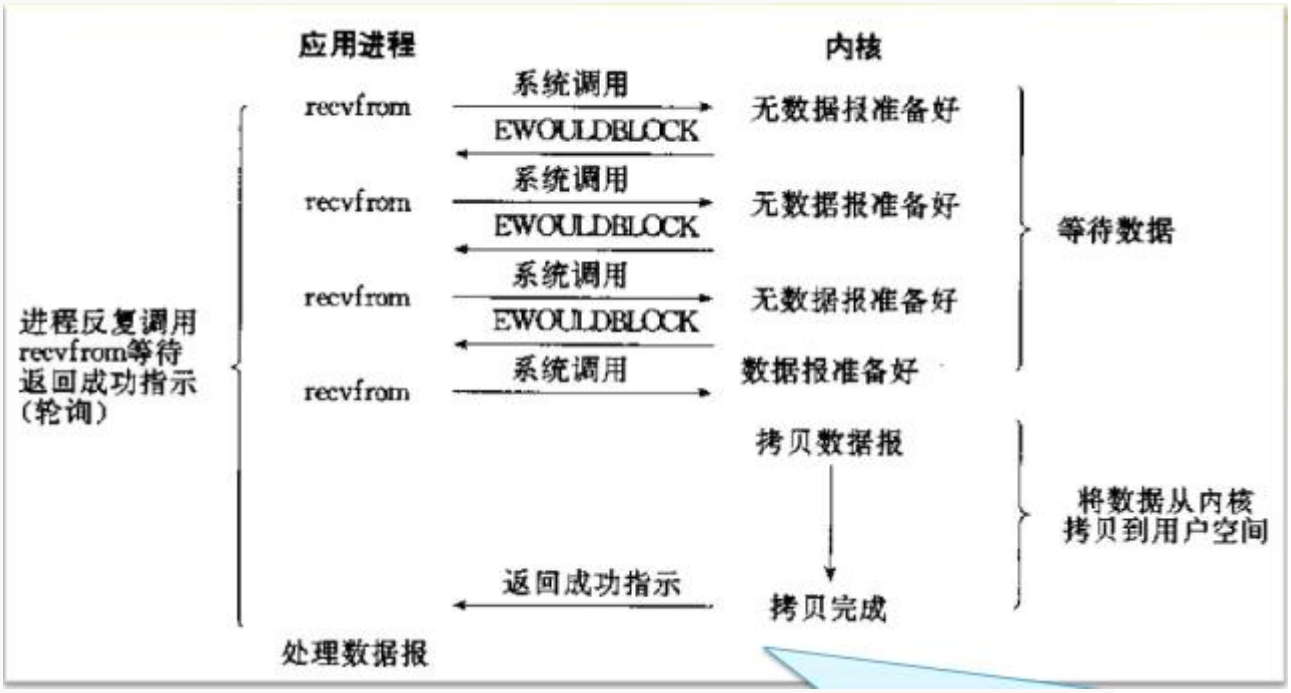

当用户进程调用了 recvfrom 这个系统调用，kernel 就开始了 IO 的第一个阶段：准备数据（对于网络 IO 来说，很多时候数据在一开始还没有到达。比如，还没有收到一个完整的 UDP 包。这个时候 kernel 就要等待足够的数据到来）。这个过程需要等待，也就是说数据被拷贝到操作系统内核的缓冲区中是需要一个过程的。而在用户进程这边，整个进程会被阻塞（当然，是进程自己选择的阻塞）。当 kernel 一直等到数据准备好了，它就会将数据从 kernel 中拷贝到用户内存，然后 kernel 返回结果，用户进程才解除 block 的状态，重新运行起来。

**所以，阻塞 IO 的特点就是在 IO 执行的两个阶段都被阻塞了。**

## 2）非阻塞 I/O（nonblocking 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：

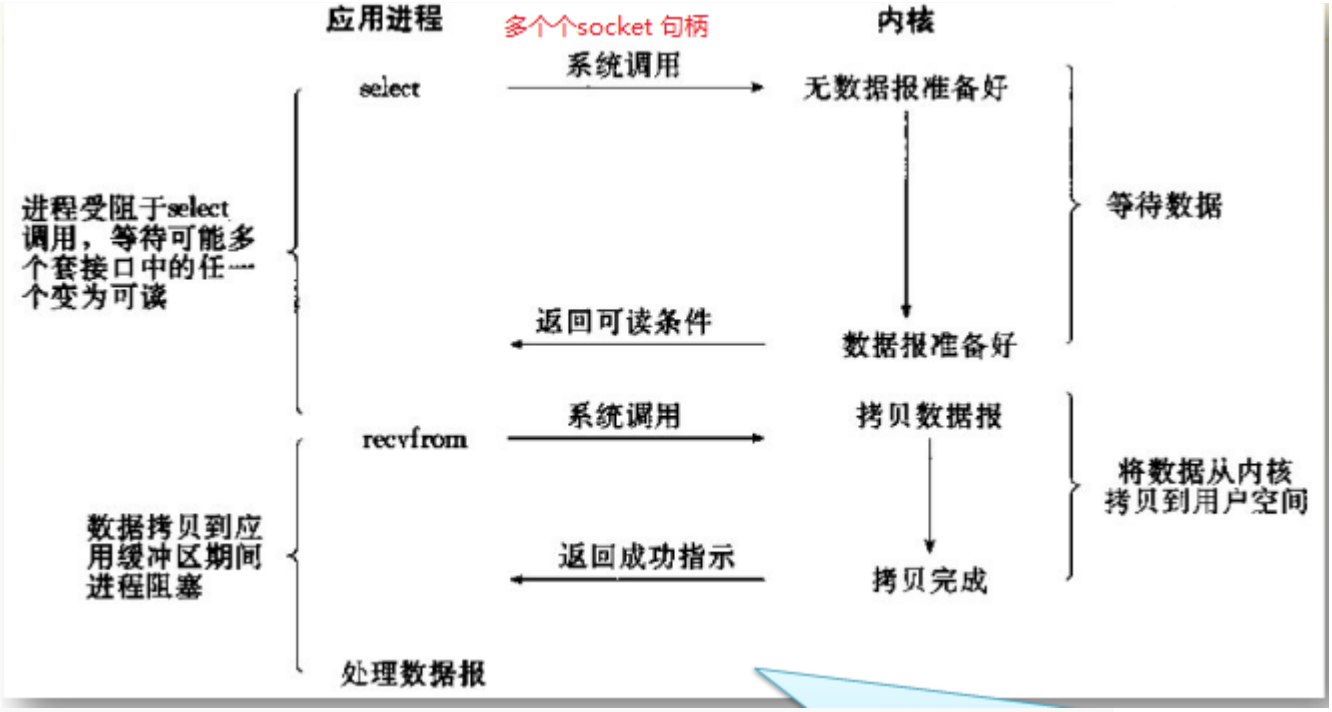



当用户进程发出 read 操作时，如果 kernel 中的数据还没有准备好，那么它并不会 block 用户进程，而是立刻返回一个 error。从用户进程角度讲 ，它发起一个 read 操作后，并不需要等待，而是马上就得到了一个结果。用户进程判断结果是一个 error 时，它就知道数据还没有准备好，于是它可以再次发送 read 操作。一旦 kernel 中的数据准备好了，并且又再次收到了用户进程的 system call，那么它马上就将数据拷贝到了用户内存，然后返回。

**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。**已经可以实现多并发，从内核态拷贝到用户态还有堵塞

## 3）I/O 多路复用（ IO multiplexing）

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

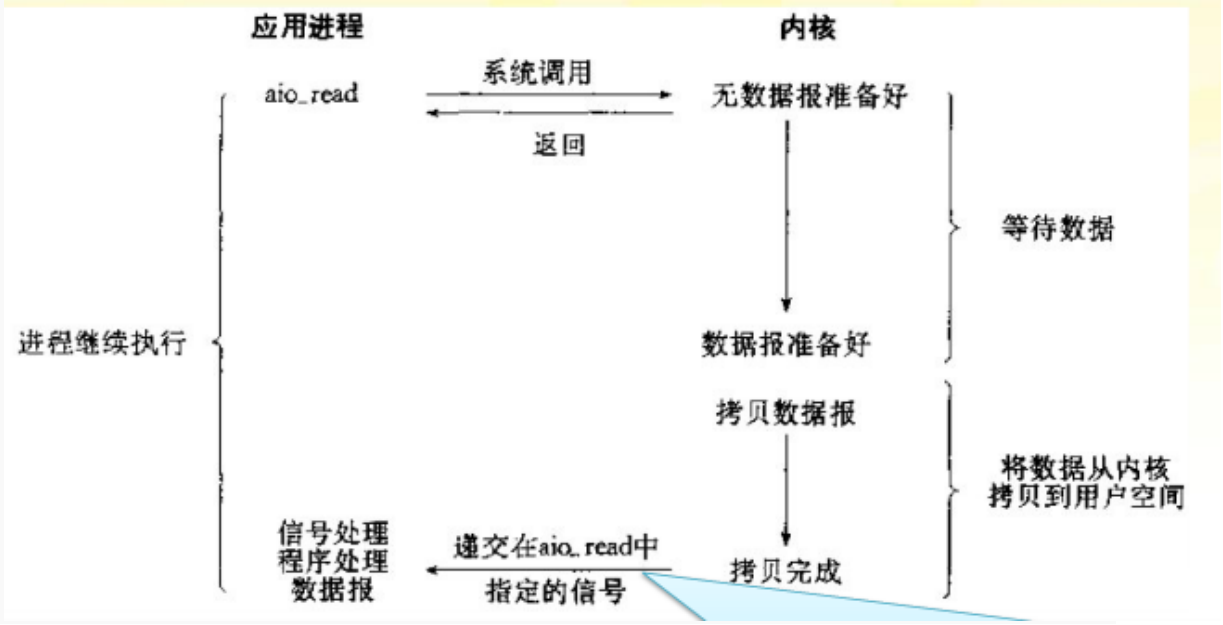



**当用户进程调用了 select，那么整个进程会被 block**，而同时，kernel 会“监视”所有 select 负责的 socket，当任何一个 socket 中的数据准备好了，select 就会返回。这个时候用户进程再调用 read 操作，将数据从 kernel 拷贝到用户进程。

**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。**

## 4）异步 I/O（asynchronous IO）

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



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。
如同：网购之后，直接可以干别的了，之后快递就送到家门口了。

# 3、总结

**阻塞 IO 和非阻塞 IO 的区别：**
调用阻塞会一直阻塞住对应的进程直到操作完成。
非阻塞 IO 在 kernel 还准备数据的情况下会立刻返回。
**同步 IO 和异步 IO 的区别：**
同步 IO 做”IO 操作”的时候会将进程阻塞，阻塞 IO、非阻塞 IO、IO 多路复用都是同步 IO。异步则不一样，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，进程完全没有被阻塞。

# 4、select poll epoll IO 多路复用介绍

**首先列一下，sellect、poll、epoll 三者的区别**
**select**
select 最早于 1983 年出现在 4.2BSD 中，它通过一个 select()系统调用来监视多个文件描述符的数组，当 select()返回后，该数组中就绪的文件描述符便会被内核修改标志位，使得进程可以获得这些文件描述符从而进行后续的读写操作。

select 目前几乎在所有的平台上支持

select 的一个缺点在于单个进程能够监视的文件描述符的数量存在最大限制，在 Linux 上一般为 1024，不过可以通过修改宏定义甚至重新编译内核的方式提升这一限制。

另外，select()所维护的存储大量文件描述符的数据结构，随着文件描述符数量的增大，其复制的开销也线性增长。同时，由于网络响应时间的延迟使得大量 TCP 连接处于非活跃状态，但调用 select()会对所有 socket 进行一次线性扫描，所以这也浪费了一定的开销。
**poll**
它和 select 在本质上没有多大差别，但是 poll 没有最大文件描述符数量的限制。

一般也不用它，相当于过渡阶段。

**epoll**

直到 Linux2.6 才出现了由内核直接支持的实现方法，那就是 epoll。被公认为 Linux2.6 下性能最好的多路 I/O 就绪通知方法。windows 不支持。

没有最大文件描述符数量的限制。

比如 100 个连接，有两个活跃了，epoll 会告诉用户这两个两个活跃了，直接取就 ok 了，而 select 是循环一遍。

（了解）epoll 可以同时支持水平触发和边缘触发（Edge Triggered，只告诉进程哪些文件描述符刚刚变为就绪状态，它只说一遍，如果我们没有采取行动，那么它将不会再次告知，这种方式称为边缘触发），理论上边缘触发的性能要更高一些，但是代码实现相当复杂。

另一个本质的改进在于 epoll 采用基于事件的就绪通知方式。在 select/poll 中，进程只有在调用一定的方法后，内核才对所有监视的文件描述符进行扫描，而 epoll 事先通过 epoll_ctl() 来注册一个文件描述符，一旦基于某个文件描述符就绪时，内核会采用类似 callback 的回调机制，迅速激活这个文件描述符，当进程调用 epoll_wait()时便得到通知。

所以市面上上见到的所谓的异步 IO，比如 nginx、Tornado、等，我们叫它异步 IO，实际上是 IO 多路复用。

异步 IO 模块，3.0 里才有，叫 asyncio

# 5、select IO 多路复用代码实例

select 模拟一个 socket server，注意 socket 必须在非阻塞情况下才能实现 IO 多路复用。接下来通过例子了解 select 是如何通过单进程实现同时处理多个非阻塞的 socket 连接的。

服务端

```
?
1 import select
2 import socket
3 import queue
4
5 server = socket.socket()
6 server.bind(('localhost',9000))
7 server.listen(1000)
8
```

```
server.setblocking(False)  # 设置成非阻塞模式，accept 和 recv 都非阻塞
# 这里如果直接 server.accept() ，如果没有连接会报错，所以有数据才调他们
# BlockIOError: [WinError 10035] 无法立即完成一个非阻塞性套接字操作。
msg_dic = {}
inputs = [server,]  # 交给内核、select 检测的列表。
# 必须有一个值，让 select 检测，否则报错提供无效参数。
# 没有其他连接之前，自己就是个 socket，自己就是个连接，检测自己。活动了说明有链接
outputs = []  # 你往里面放什么，下一次就出来了

while True:
    readable, writeable, exceptional = select.select(inputs, outputs, inputs)  # 定义检测
    #新来连接检测列表          异常（断开）
    # 异常的也是 inputs 是：  检测那些连接的存在异常
    print(readable,writeable,exceptional)
    for r in readable:
        if r is server:  # 有数据，代表来了一个新连接
            conn, addr = server.accept()
            print("来了个新连接",addr)
            inputs.append(conn)  # 把连接加到检测列表里，如果这个连接活动了，就说明数据来了
            # inputs = [server.conn] # 【conn】只返回活动的连接，但怎么确定是谁活动了
            # 如果 server 活动，则来了新连接，conn 活动则来数据
            msg_dic[conn] = queue.Queue()  # 初始化一个队列，后面存要返回给这个客户端的数据
        else:
            try:
                data = r.recv(1024)  # 注意这里是 r，而不是 conn，多个连接的情况
                print("收到数据",data)
                # r.send(data) # 不能直接发，如果客户端不收，数据就没了
                msg_dic[r].put(data)  # 往里面放数据
                outputs.append(r)  # 放入返回的连接队列里
            except ConnectionResetError as e:
                print("客户端断开了",r)
                if r in outputs:
                    outputs.remove(r) #清理已断开的连接
                inputs.remove(r) #清理已断开的连接
                del msg_dic[r] ##清理已断开的连接

    for w in writeable:  # 要返回给客户端的连接列表
        data_to_client = msg_dic[w].get()  # 在字典里取数据
        w.send(data_to_client)  # 返回给客户端
```

```
53          outputs.remove(w)  # 删除这个数据，确保下次循环的时候不返回这个已经处理
54完的连接了。
```

```
    for e in exceptional:  # 如果连接断开，删除连接相关数据
        if e in outputs:
            outputs.remove(e)
        inputs.remove(e)
        del msg_dic[e]
```

客户端

?

```
1  import socket
2  client = socket.socket()
3
4  client.connect(('localhost', 9000))
5
6  while True:
7      cmd = input('>>> ').strip()
8      if len(cmd) == 0 : continue
9      client.send(cmd.encode('utf-8'))
10     data = client.recv(1024)
11     print(data.decode())
12
13 client.close()
```

# 6、selectors 模块

selectors 已经把 ipoll、select 封装好了，默认用 epoll，如果机器上不支持 epoll，比如 window 是不支持 epoll，就用 select。

服务端

?

```
1 import selectors
2 import socket
3
4 sel = selectors.DefaultSelector()
5
```

```
6 def accept(sock, mask):
7     conn, addr = sock.accept()  # 开始连接
8     print('accepted', conn, 'from', addr)
9     conn.setblocking(False)  # 连接设为非阻塞模式
10    sel.register(conn, selectors.EVENT_READ, read)  # 把 conn 注册到 sel 对象里
11    # 新连接回调 read 函数
12
13def read(conn, mask):
14    data = conn.recv(1024)  # 接收数据
15    if data:
16        print('echoing', repr(data), 'to', conn)
17        conn.send(data)  # Hope it won't block
18    else:
19        print('closing', conn)
20        sel.unregister(conn)  # 取消注册
21        conn.close()
22
23sock = socket.socket()
24sock.bind(('localhost', 9000))
25sock.listen(100)
26sock.setblocking(False)
27sel.register(sock, selectors.EVENT_READ, accept)  # 注册事件
28#       sock 注册过来                    新连接调用这个函数
29
30while True:
31    events = sel.select()  # 有可能调用 epoll，也有可能调用 select，看系统支持
32    # 默认是阻塞，有活动连接，就返回活动的列表
33    for key, mask in events:
34        callback = key.data  # 掉 accept 函数
35        callback(key.fileobj, mask)  # key.fileobj = 文件句柄 （相当于上个例子中检测的自己）
```