人云思云 发布于 人云思云
2015年08月07日 · 116.2k 人阅读

# Linux IO模式及 select、poll、epoll详解

> 注：本文是对众多博客的学习和总结，可能存在理解错误。请带着怀疑的眼光，同时如果有错误希望能指出。

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。

## 一 概念说明

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**，具体的可以参考这篇文章：进程切换

## 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。

**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。

# 二 IO模式

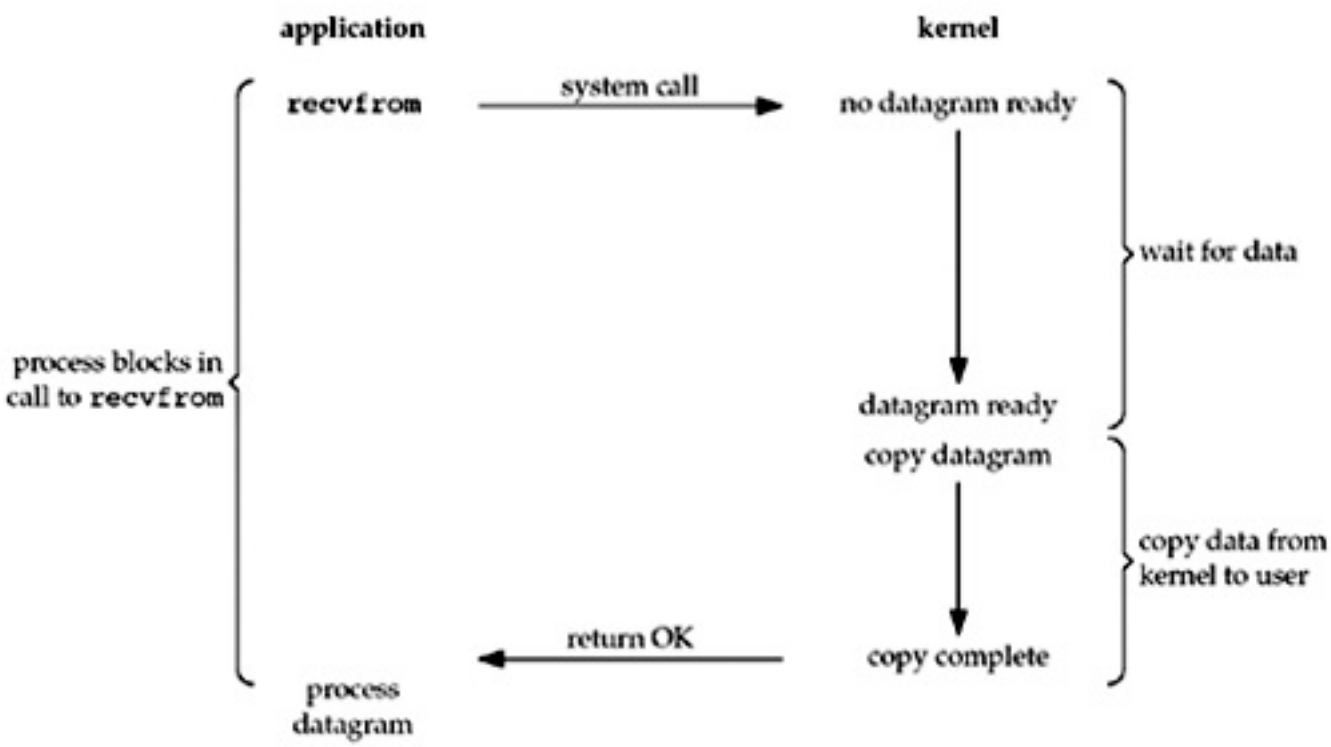
刚才说了，对于一次IO访问（以read举例），数据会先被拷贝到操作系统内核的缓冲区中，然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间。所以说，当一个read操作发生时，它会经历两个阶段：
1. 等待数据准备 (Waiting for the data to be ready)
2. 将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)

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。

## 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：

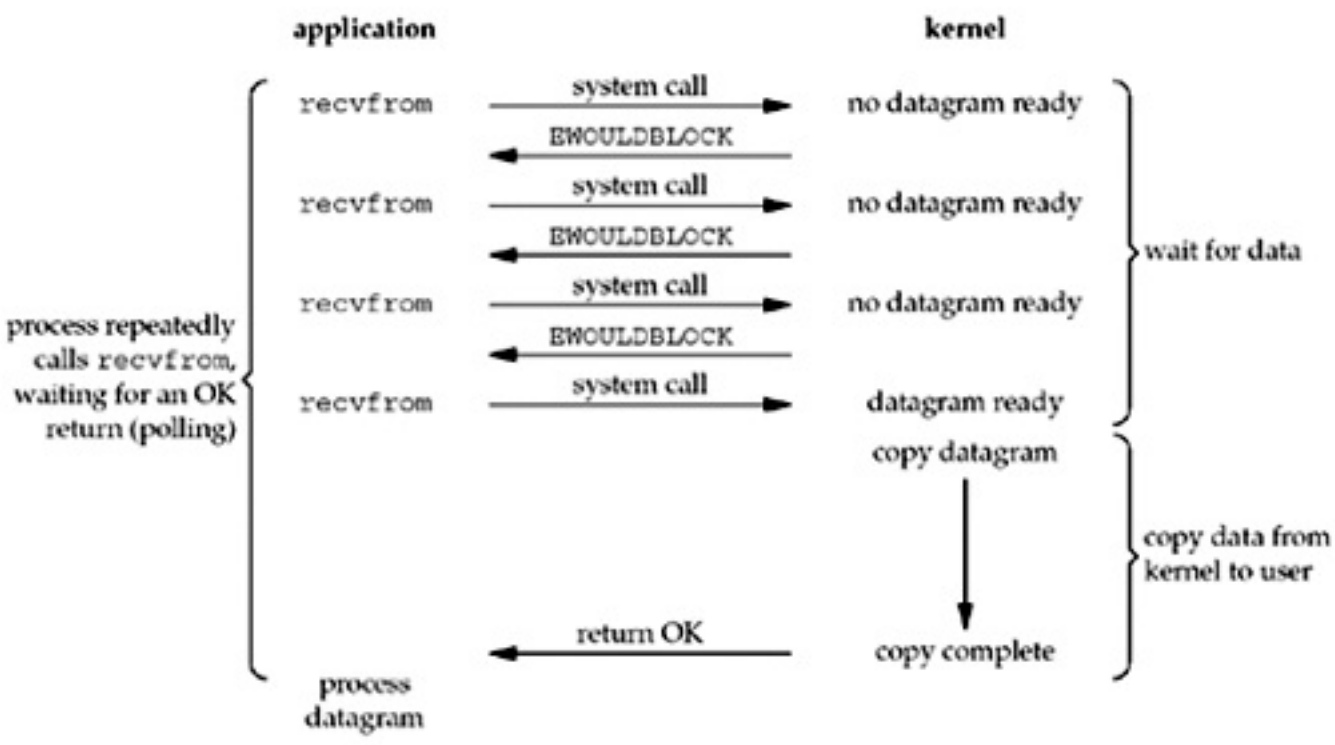



当用户进程调用了recvfrom这个系统调用，kernel就开始了IO的第一个阶段：准备数据（对于网络IO来说，很多时候数据在一开始还没有到达。比如，还没有收到一个完整的UDP包。这个时候kernel就要等待足够的数据到来）。这个过程需要等待，也就是说数据被拷贝到操作系统内核的缓冲区中是需要一个过程的。而在用户进程这边，整个进程会被阻塞（当然，是进程自己选择的阻塞）。当kernel一直等到数据准备好了，它就会将数据从kernel中拷贝到用户内存，然后kernel返回结果，用户进程才解除block的状态，重新运行起来。

> 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：

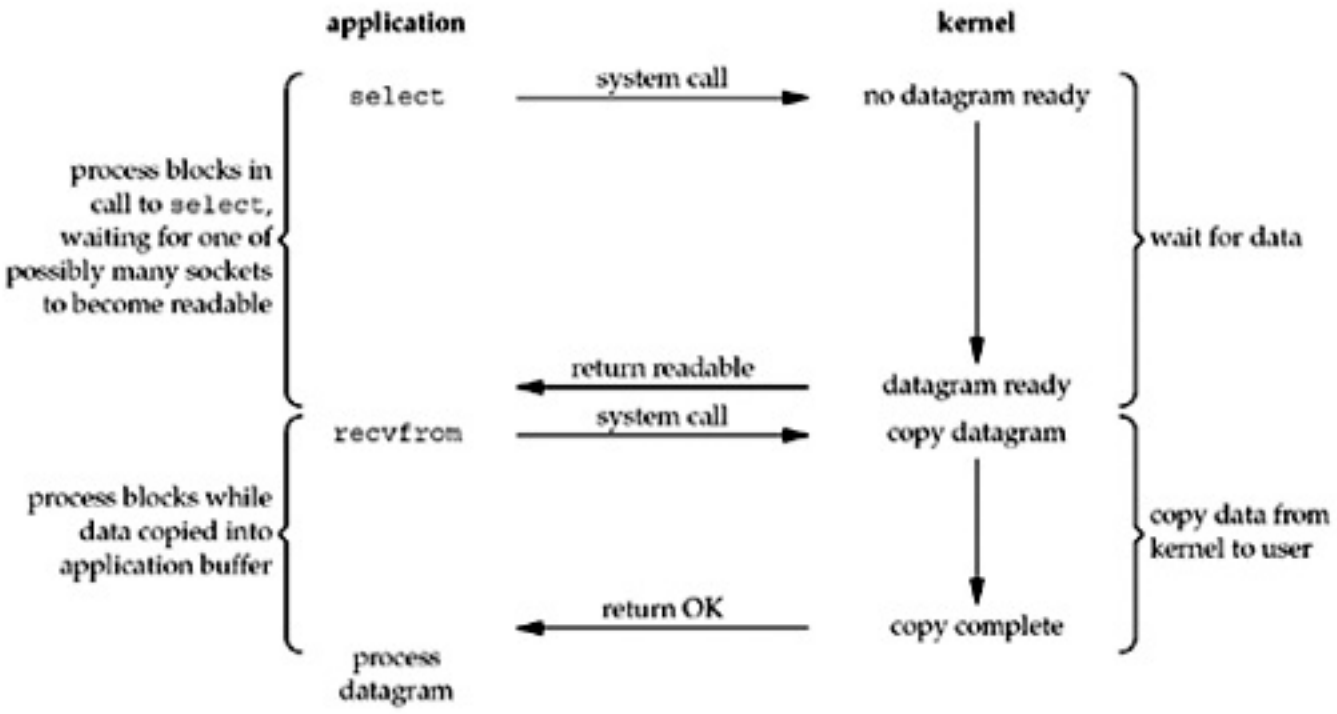



当用户进程发出read操作时，如果kernel中的数据还没有准备好，那么它并不会block用户进程，而是立刻返回一个error。从用户进程角度讲 ，它发起一个read操作后，并不需要等待，而是马上就得到了一个结果。用户进程判断结果是一个error时，它就知道数据还没有准备好，于是它可以再次发送read操作。一旦kernel中的数据准备好了，并且又再次收到了用户进程的system call，那么它马上就将数据拷贝到了用户内存，然后返回。

> 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。

## 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。

这个图和blocking IO的图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blocking 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connection。

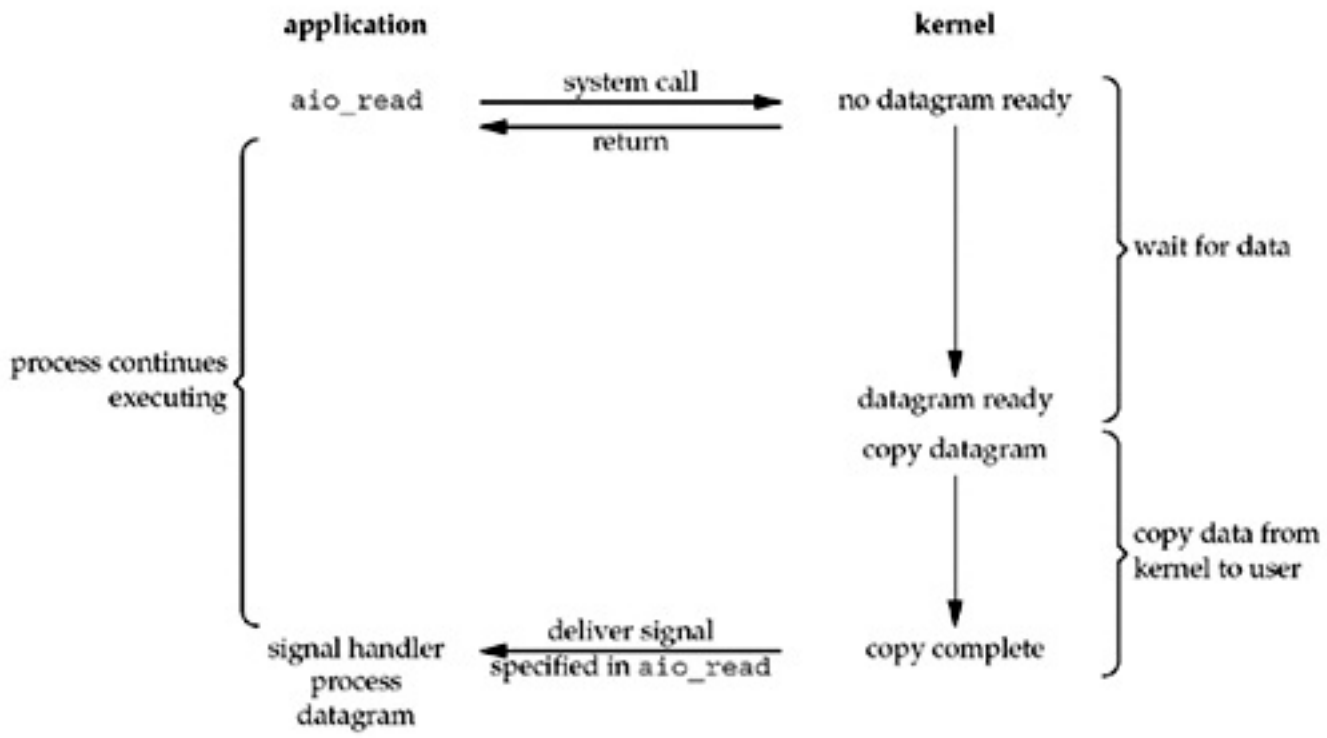
所以，如果处理的连接数不是很高的话，使用select/epoll的web server不一定比使用multi-threading + blocking IO的web server性能更好，可能延迟还更大。select/epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）

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）

inux下的asynchronous IO其实用得很少。先看一下它的流程：

**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。

# 总结

## 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，而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

在说明synchronous IO和asynchronous IO的区别之前，需要先给出两者的定义。POSIX的定义是这样子的：
- 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;

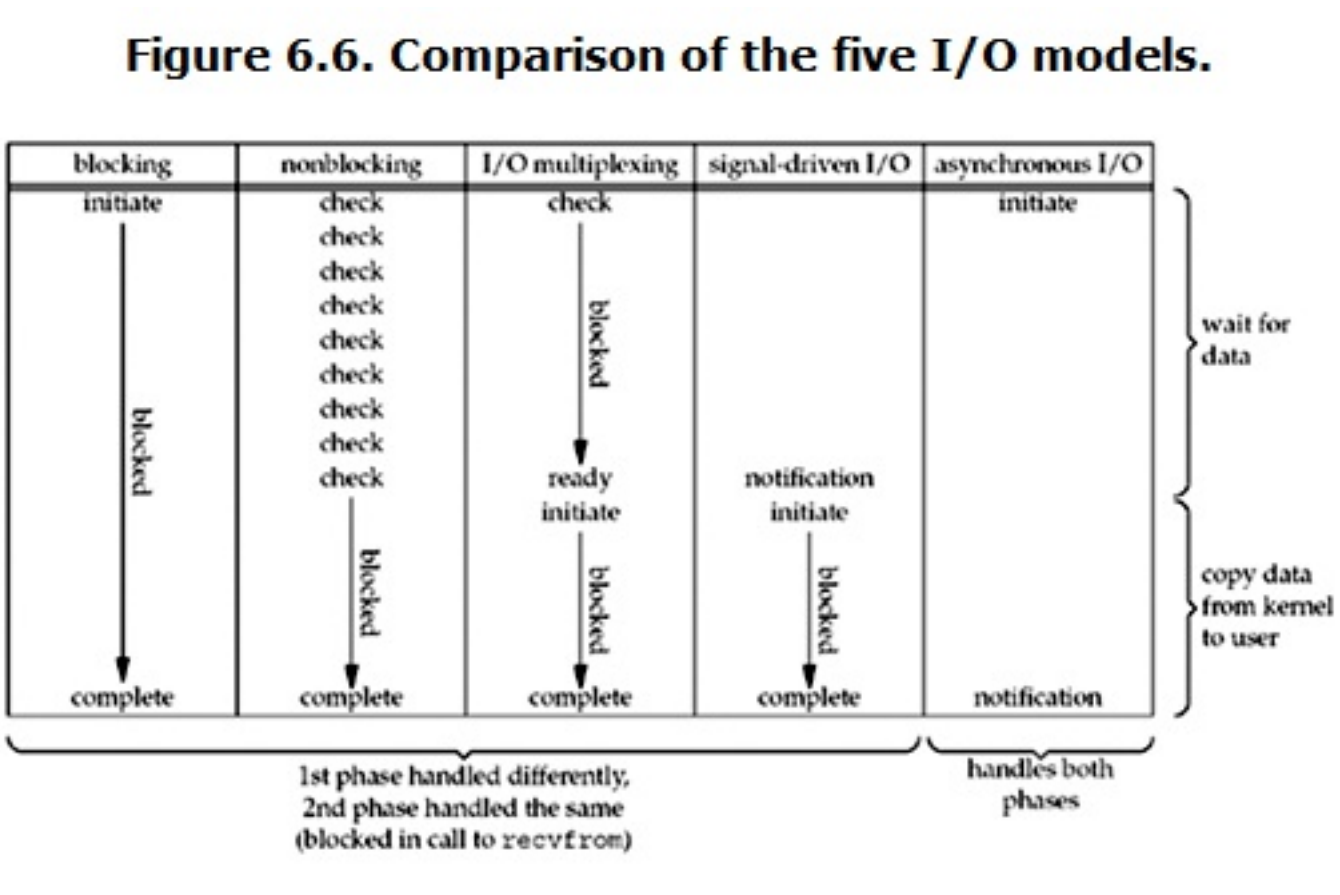
两者的区别就在于synchronous IO做”IO operation”的时候会将process阻塞。按照这个定义，之前所述的blocking IO，non-blocking IO，IO multiplexing都属于synchronous IO。

有人会说，non-blocking IO并没有被block啊。这里有个非常“狡猾”的地方，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：




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。

# 三 I/O 多路复用之select、poll、epoll详解

select，poll，epoll都是IO多路复用的机制。I/O多路复用就是通过一种机制，一个进程可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。（这里啰嗦下）

## select

```
int select (int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct tim
```

select 函数监视的文件描述符分3类，分别是writefds、readfds、和exceptfds。调用后select函数会阻塞，直到有描述副就绪（有数据 可读、可写、或者有except），或者超时（timeout指定等待时间，如果立即返回设为null即可），函数返回。当select函数返回后，可以 通过遍历fdset，来找到就绪的描述符。

select目前几乎在所有的平台上支持，其良好跨平台支持也是它的一个优点。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，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，但 是这样也会造成效率的降低。

## poll

```
int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

不同与select使用三个位图来表示三个fdset的方式，poll使用一个 pollfd的指针实现。

```
struct pollfd {
    int fd; /* file descriptor */
    short events; /* requested events to watch */
    short revents; /* returned events witnessed */
};
```

pollfd结构包含了要监视的event和发生的event，不再使用select“参数-值”传递的方式。同时，pollfd并没有最大数量限制（但是数量过大后性能也是会下降）。 和select函数一样，poll返回后，需要轮询pollfd来获取就绪的描述符。

> 从上面看，select和poll都需要在返回后， 通过遍历文件描述符来获取已经就绪的socket 。事实上，同时连接的大量客户端在一时刻可能只有很少的处于就绪状态，因此随着监视的描述符数量的增长，其效率也会线性下降。

## epoll

epoll是在2.6内核中提出的，是之前的select和poll的增强版本。相对于select和poll来说，epoll更加灵活，没有描述符限制。epoll使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。

## 一 epoll操作过程

epoll操作过程需要三个接口，分别如下：

```
int epoll_create(int size); //创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
```

1. int epoll_create(int size);

创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大，这个参数不同于select()中的第一个参数，给出最大监听的fd+1的值，参数size并不是限制了epoll所能监听的描述符最大个数，只是对内核初始分配内部数据结构的一个建议。

当创建好epoll句柄后，它就会占用一个fd值，在linux下如果查看/proc/进程id/fd/，是能够看到这个fd的，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

2. int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);

函数是对指定描述符fd执行op操作。

- epfd：是epoll_create()的返回值。

- op：表示op操作，用三个宏来表示：添加EPOLL_CTL_ADD，删除EPOLL_CTL_DEL，修改EPOLL_CTL_MOD。分别添加、删除和修改对fd的监听事件。

- fd：是需要监听的fd（文件描述符）

- epoll_event：是告诉内核需要监听什么事，struct epoll_event结构如下：

```
struct epoll_event {
  __uint32_t events;  /* Epoll events */
  epoll_data_t data;  /* User data variable */
};

//events可以是以下几个宏的集合:
EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；
EPOLLOUT：表示对应的文件描述符可以写；
EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；
EPOLLERR：表示对应的文件描述符发生错误；
EPOLLHUP：表示对应的文件描述符被挂断；
EPOLLET：  将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的
EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这
```

3. int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);

等待epfd上的io事件，最多返回maxevents个事件。

参数events用来从内核得到事件的集合，maxevents告之内核这个events有多大，这个maxevents的值不能大于创建epoll_create()时的size，参数timeout是超时时间（毫秒，0会立即返回，-1将不确定，也有说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回0表示已超时。

## 二 工作模式

epoll对文件描述符的操作有两种模式：**LT（level trigger）**和**ET（edge trigger）**。LT模式是默认模式，LT模式与ET模式的区别如下：

**LT模式**：当epoll_wait检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用epoll_wait时，会再次响应应用程序并通知此事件。

**ET模式**：当epoll_wait检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用epoll_wait时，不会再次响应应用程序并通知此事件。

### 1. LT模式

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。

### 2. ET模式

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)

ET模式在很大程度上减少了epoll事件被重复触发的次数，因此效率要比LT模式高。epoll工作在ET模式的时候，必须使用非阻塞套接口，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。

### 3. 总结

**假如有这样一个例子：**

1. 我们已经把一个用来从管道中读取数据的文件句柄(RFD)添加到epoll描述符
2. 这个时候从管道的另一端被写入了2KB的数据
3. 调用epoll_wait(2)，并且它会返回RFD，说明它已经准备好读取操作
4. 然后我们读取了1KB的数据
5. 调用epoll_wait(2).....

LT模式：

如果是LT模式，那么在第5步调用epoll_wait(2)之后，仍然能受到通知。

ET模式：

如果我们在第1步将RFD添加到epoll描述符的时候使用了EPOLLET标志，那么在第5步调用epoll_wait(2)之后将有可能会挂起，因为剩余的数据还存在于文件的输入缓冲区内，而且数据发出端还在等待一个针对已经发出数据的反馈信息。只有在监视的文件句柄上发生了某个事件的时候 ET 工作模式才会汇报事件。因此在第5步的时候，调用者可能会放弃等待仍在存在于文件输入缓冲区内的剩余数据。

当使用epoll的ET模型来工作时，当产生了一个EPOLLIN事件后，
读数据的时候需要考虑的是当recv()返回的大小如果等于请求的大小，那么很有可能是缓冲区还有数据未读完，也意味着该次事件还没有处理完，所以还需要再次读取：

```
while(rs){
  buflen = recv(activeevents[i].data.fd, buf, sizeof(buf), 0);
  if(buflen < 0){
    // 由于是非阻塞的模式,所以当errno为EAGAIN时,表示当前缓冲区已无数据可读
    // 在这里就当作是该次事件已处理处.
    if(errno == EAGAIN){
        break;
    }
    else{
        return;
    }
  }
  else if(buflen == 0){
     // 这里表示对端的socket已正常关闭.
  }

 if(buflen == sizeof(buf){
      rs = 1;   // 需要再次读取
 }
 else{
      rs = 0;
 }
}
```

> Linux中的EAGAIN含义

Linux环境下开发经常会碰到很多错误(设置errno)，其中EAGAIN是其中比较常见的一个错误(比如用在非阻塞操作中)。
从字面上来看，是提示再试一次。这个错误经常出现在当应用程序进行一些非阻塞(non-blocking)操作(对文

件或socket)的时候。

例如，以 O_NONBLOCK的标志打开文件/socket/FIFO，如果你连续做read操作而没有数据可读。此时程序不会阻塞起来等待数据准备就绪返回，read函数会返回一个错误EAGAIN，提示你的应用程序现在没有数据可读请稍后再试。
又例如，当一个系统调用(比如fork)因为没有足够的资源(比如虚拟内存)而执行失败，返回EAGAIN提示其再调用一次(也许下次就能成功)。

## 三 代码演示

下面是一段不完整的代码且格式不对，意在表述上面的过程，去掉了一些模板代码。

```
#define IPADDRESS   "127.0.0.1"
#define PORT        8787
#define MAXSIZE     1024
#define LISTENQ     5
#define FDSIZE      1000
#define EPOLLEVENTS 100

listenfd = socket_bind(IPADDRESS,PORT);

struct epoll_event events[EPOLLEVENTS];

//创建一个描述符
epollfd = epoll_create(FDSIZE);

//添加监听描述符事件
add_event(epollfd,listenfd,EPOLLIN);

//循环等待
for ( ; ; ){
    //该函数返回已经准备好的描述符事件数目
    ret = epoll_wait(epollfd,events,EPOLLEVENTS,-1);
    //处理接收到的连接
    handle_events(epollfd,events,ret,listenfd,buf);
```

## 四 epoll总结

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。(此处去掉了遍历文件描述符，而是通过监听回调的的机制 。这正是epoll的魅力所在。)

**epoll的优点主要是一下几个方面：**

1. 监视的描述符数量不受限制，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。select的最大缺点就是进程打开的fd是有数量限制的。这对 于连接数量比较大的服务器来说根本不能满足。虽然也可以选择多进程的解决方案( Apache就是这样实现的)，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。

1. IO的效率不会随着监视fd的数量的增长而下降。epoll不同于select和poll轮询的方式，而是通过每个fd定义的回调函数来实现的。只有就绪的fd才会执行回调函数。

> 如果没有大量的idle -connection或者dead-connection，epoll的效率并不会比select/poll高很多，但是当遇到大量的idle- connection，就会发现epoll的效率大大高于select/poll。

# 参考


用户空间与内核空间，进程上下文与中断上下文[总结]
进程切换
维基百科-文件描述符
Linux 中直接 I/O 机制的介绍
IO - 同步，异步，阻塞，非阻塞 （亡羊补牢篇）
Linux中select poll和epoll的区别
IO多路复用之select总结
IO多路复用之poll总结
IO多路复用之epoll总结




## 你可能感兴趣的文章


高性能服务器开发基础系列 （一）主线程与工作线程的分工
高性能网络编程(二)：上一个10年，著名的C10K并发连接问题
Web APP编程模型和IO策略
IO多路复用

基于tornado的web ssh项目

## 47 条评论

默认排序 | 时间排序

**testapp002** · 2016年01月20日

@人云思云 , 有个问题想请教一下。我看到一些文章说(http://www.cnblogs.com/huacw/p/3518461.html)，只有windows的iocp才是真正的异步IO。windows的IOCP会在接收到网络数据后，然后kernel copy data to appliction(也就是说从内核空间拷贝到用户空间)，等这个步骤完成后，才会通知应用程序。而linux的epoll不是真正的异步IO，还需要应用程序做一次系统调用，才会kernel copy data to appliction。

可是我又看到一种说法(http://www.smithfox.com/?e=191)，说epoll使用了mmap，大概意思是用户空间和内核空间共享一块内存。按照这种说法岂不是epoll和iocp差不多？ 这篇文章说，"而用 epoll 时, epoll 通知用户空间的Appliation时, 数据已经在用户空间, 所以 Appliation调用 read API 时, 只是读取用户空间的 buffer, 没有 kernal space和 user space的switch了"

而这篇文章(http://blog.csdn.net/thinkry/article/details/1296466)提到 "Iocp和Epoll的一个根本区别是Epoll是返回多个socket中有事件发生的socket，类似select，不过性能更高；Iocp则是在动作完成后返回通知（完成端口中的“完成”就是这个意思）。"

我有些晕了。请楼主帮忙解释一下， epoll和IOCP的区别在哪里？ 如果epoll动作也能做到数据放入到用户空间，那么感觉也是动作完成后才通知应用程序的。真的不太懂

另外我的邮箱testapp002@126.com，恳请楼主把QQ发给我，希望得到你的指教

赞 +1　回复

> 这些问题同样困扰着我。
> — **M莫M** · 4 天前

添加回复

**细胞核s** · 2016年07月06日

这里有一个疑问，select和poll在内核中，也应该是轮询所有的fd，来实现的吧？而并不仅仅是因为返回之后需要遍历?

赞 +1　回复

> 当然内核需要遍历所有的fd，不然它怎么知道那个fd就绪了从而导致select或者poll函数返回。
> — **sillyboy** · 1月19日

添加回复

**zy0695354** · 2015年08月11日

写得不错~!

赞 回复

**hadix** · 2015年11月27日

表达很清楚，图示也很容易理解，这是我看过解释这个问题最好的文章
http://segmentfault.com/a/1190000003063859?utm_source=Weibo&utm_medium=shareLink&utm_campaign=socialShare&from=timeline&isappinstalled=0

赞 回复

**lirau** · 2015年12月20日

您好，文章收获很大，但是有个问题，文中提到“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。”，不断的询问的周期是什么？如何选择合适的周期使得既不占太多的资源又可以及时获取结果?

赞 回复

**人云思云** 作者 · 2015年12月20日

正常情况下内核准备数据的过程比较长点，数据拷贝的过程只是内存操作，相对较快。但是非阻塞模式下进程依旧需要轮询内核数据准备结果。至于你问的周期优化问题，多路复用就是优化这样的问题。让一个进程同时监控多个文件描述符，只要一个准备好就可以返回。当然这里又牵扯到Select、Poll和epoll的内容，文中有做出一定的解释。

赞 回复

**manxisuo** · 2016年01月29日

赞赞赞赞赞!

赞 回复

**sadamu007** · 2016年02月25日

写得极好，强!

赞 回复

**Double_Kill** · 2016年04月13日

看完你的文章后，我对IO模型又多了一层认识。十分感谢

赞 回复

**qeesung** · 2016年05月16日

难的一见的好文，apue中讲的比较晦涩，图大赞!

赞 回复

**langwolf** · 2016年05月23日

非常好的一篇文章，狂赞

赞 回复

**rqy** · 2016年07月25日

文章很棒，思路很清晰。

赞 回复

**earlyme** · 2016年07月29日

当recv()返回的大小如果等于请求的大小，那么很有可能是缓冲区还有数据未读完---
这句话没看懂，recv接收的数据等于请求的大小，应该是读完了啊？为啥这样子呢

赞 回复

> 多半没读完，在读取一次直到小于请求大小
> — **tangwentao3014** · 2017年04月24日

添加回复

**lrenjundk** · 2016年08月08日

**相当赞**， 是我见过最好的讲述IO的文章了，特别是Non-blocking IO和异步IO的区别。

赞 回复

**renwotao** · 2016年08月10日

大赞！！！谢作者

赞 回复

**乌合之众** · 2016年08月10日

这篇文章可以参考一下http://www.kryptosx.info/archives/854.html

赞 回复

**renwotao** · 2016年08月11日

代码演示部分有些错误：
应该先delete_event(epollfd,fd,EPOLLOUT); //删除监听
然后在close(fd);
否则报Bad file decriptor 错误

赞 回复

**bestswifter** · 2016年08月12日

@人云思云 有一个问题请问博主，您说 select 或者 poll 方法调用后，还需要通过遍历来获取已就绪的 socket，请问为什么不能在方法中直接返回呢?

赞 回复

**selfrebuild** · 2016年09月06日

文章写的很好, 不过有参考<Unix 网络编程>,<Unix/Linux 系统编程手册>这2本书的内容, 比如关键插图来自<Unix 网络编程>, 应该予以注明呀?

赞 回复

**小强中国** · 2016年10月21日

作为一个 PHP 开发, 如何使用到这些? swoole 扩展中有这些, 但是相对我们来说偏底层了.

赞 回复

> php开发者开发了swoole
> — **chrisliu** · 2017年06月13日

添加回复

显示更多评论

文明社会，理性评论

发布评论