# IO

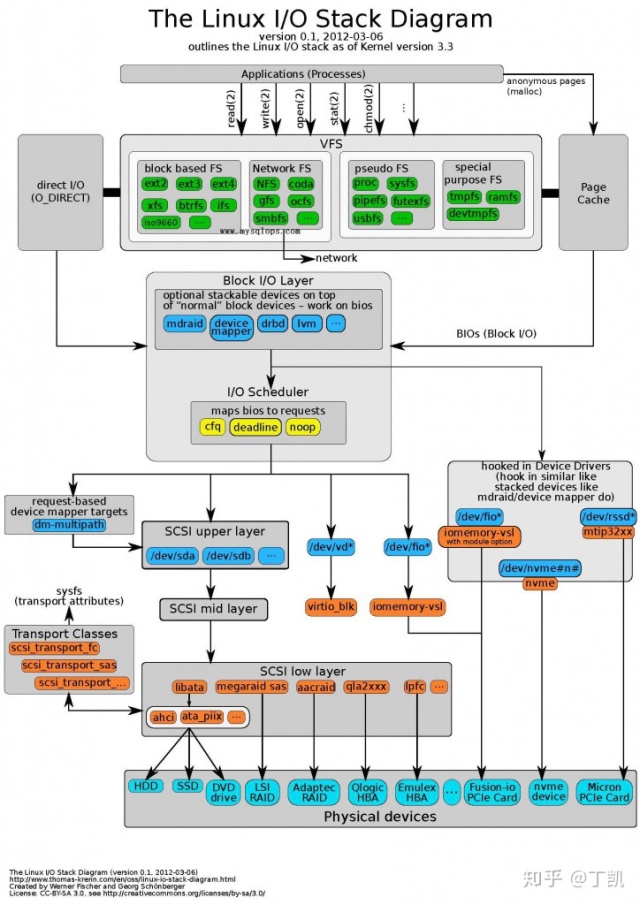
## 概述

I/O通常有内存IO、网络I/O、磁盘I/O等，但我们通常说的是网络I/O以及磁盘I/O。网络I/O：本质是socket读取。

每次I/O请求，都会有两个阶段组成： 第一步：等待数据，即数据从磁盘到内核内存；将数据从磁盘文件先加载到内核内存空间（缓冲区），等待数据准备完成，时间较长。第二步：复制数据，即数据内核内存到进程内存；将数据从内核缓冲复制到用户空间的进程内存中，时间较短。

## 原理

### 整体架构



#### 应用程序

这没什么好说的，通过相关系统调用(如open/read/write)发起IO请求，属于IO请求的源头。

#### 文件系统

应用程序的请求直接到达文件系统层。文件系统又分为VFS和具体文件系统（ext3、ext4等），VFS对应用层提供统一的访问接口，而ext3等文件系统则具体实现了这些接口。另外，为了提供IO性能，在该层还实现了诸如page cache等功能。同时，**用户也可以选择绕过page cache，而是直接使用direct模式进行IO（如数据库）**。

注：数据库绕过系统的页缓存，自己实现了自己的缓存。

#### 块设备层

文件系统将IO请求打包提交给块设备层，该层会对这些IO请求作合并、排序、调度等，然后以新的格式发往更底层。在该层次上实现了多种电梯调度算法，如cfq、deadline等。

#### SCSI层

块设备层将请求发往SCSI层，SCSI就开始真实处理这些IO请求，但是SCSI层又对其内部按照功能划分了不同层次：

SCSI高层：高层驱动负责管理disk，接收块设备层发出的IO请求，打包成SCSI层可识别的命令格式，继续往下发；

SCSI中层：中层负责通用功能，如错误处理，超时重试等；

SCSI低层：底层负责识别物理设备，将其抽象提供给高层，同时接收高层派发的scsi命令，交给物理设备处理。

#### 各层接口

清晰的接口能让复杂的系统变得容易理解和维护。

##### 应用程序-->文件系统

做开发的人可能都应该了解，通过诸如open/read/pread/write/writev等POSIX接口来调用文件系统各种功能。

##### 文件系统-->块设备层

这里我们将文件系统当成一个整体，并不区分VFS和具体文件系统。块设备层对文件系统提供的接口为submit\_bio()，接口形式如下：void submit\_bio(int rw, struct bio \*bio)文件系统向块设备提交的每个bio请求都设置了完成回调函数，记录在bio->bi\_end\_io。bio请求完成后，通过该字段通知文件系统。

##### 块设备层-->SCSI上层

scsi\_reuqest\_fn()和struct request\_queue。老实来说，块设备层和SCSI上层之间分的没有那么清楚，耦合的稍微紧密，块设备层看到的IO请求结构是request。而SCSI层看到的IO命令则是scsi\_cmnd。每个scsi设备（如scsi disk）均维护了一个请求队列request\_queue，而每个scsi设备对上层呈现的其实是一个块设备。因此，块设备和scsi设备有着天然的联系，request\_queue则是连接块设备层和SCSI层的纽带。块设备层对request请求最终会派发至request\_queue中。而在特定条件下通过泄流机制将request\_queue中积攒的request派发至SCSI层处理。而泄流的实际处理过程就是scsi\_request\_fn()函数，因此说它是块设备层和SCSI上层的接口也不为过，虽然不是特别准确。在scsi\_reuqest\_fn内会进行request至scsi\_cmnd的转换。

##### SCSI上层-->SCSI中间层

SCSI上层在收到块设备层发起的scsi命令后马不停蹄又将其转发至SCSI中间层。SCSI上层至SCSI中间层的接口是scsi\_dispatch\_cmd

static void scsi\_request\_fn(struct request\_queue \*q) {

......

// 设置scsi命令完成回调函数

cmd->scsi\_done = scsi\_done;

rtn = scsi\_dispatch\_cmd(cmd);

......

}

##### SCSI中间层--> SCSI低层

SCSI中间层收到块设备层发下来的scsi\_cmnd命令后，中间层作自己处理后，然后再将该命令继续往下传递，接下来该命令到了scsi底层，而传递的接口是 queuecommand()

static int scsi\_dispatch\_cmd(struct scsi\_cmnd \*cmd) {

......

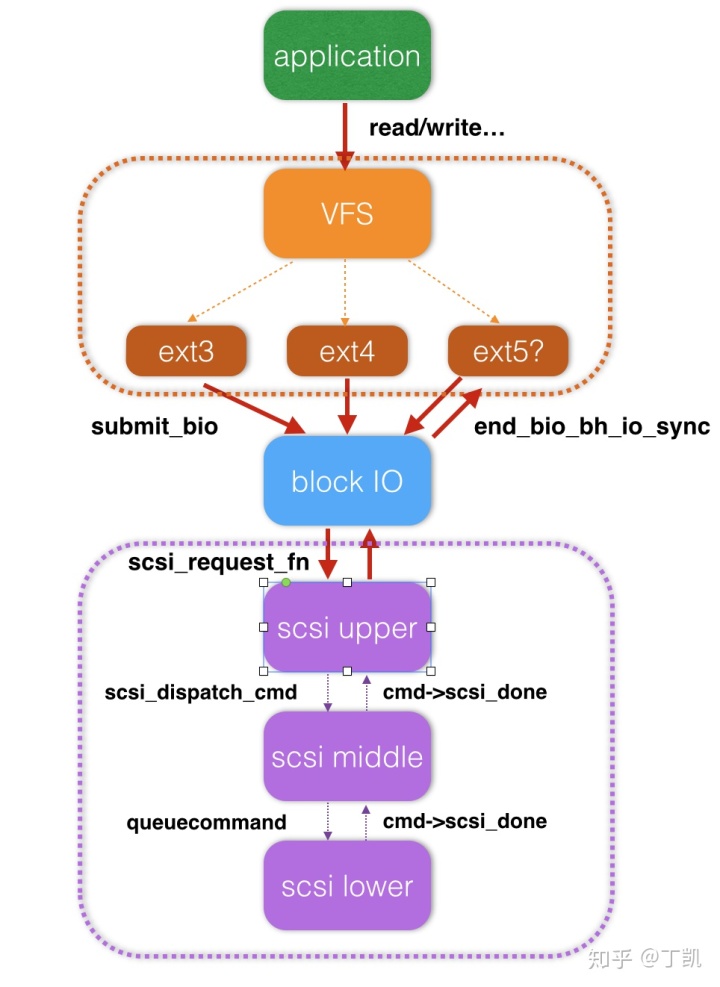
rtn = host->hostt->queuecommand(host, cmd);

......

}

host为该设备所属的主机适配器结构。任何一个SCSI主机适配器都需要实现queuecommand接口。注意这个提交过程是异步的，无需等待该命令完成便直接返回。scsi 命令完成后，会通过记录在命令内的完成函数回调上层处理，具体是cmd->scsi\_done。

#### 总结



### I/O软件目标

#### 设备独立性

现在让我们转向对I/O软件的研究，I/O软件设计一个很重要的目标就是设备独立性(device independence)。啥意思呢？这意味着我们能够编写访问任何设备的应用程序，而不用事先指定特定的设备。比如你编写了一个能够从设备读入文件的应用程序，那么这个应用程序可以从硬盘、DVD或者USB进行读入，不必再为每个设备定制应用程序。这其实就体现了设备独立性的概念。

与设备独立性密切相关的一个指标就是统一命名(uniform naming)。设备的代号应该是一个整数或者是字符串，它们不应该依赖于具体的设备。在UNIX中，所有的磁盘都能够被集成到文件系统中，所以用户不用记住每个设备的具体名称，直接记住对应的路径即可，如果路径记不住，也可以通过ls等指令找到具体的集成位置。举个例子来说，比如一个USB磁盘被挂载到了/usr/cxuan/backup下，那么你把文件复制到/usr/cxuan/backup/device下，就相当于是把文件复制到了磁盘中，通过这种方式，实现了向任何磁盘写入文件都相当于是向指定的路径输出文件。

#### 错误处理

除了设备独立性外，I/O 软件实现的第二个重要的目标就是错误处理(error handling)。通常情况下来说，错误应该交给硬件层面去处理。如果设备控制器发现了读错误的话，它会尽可能的去修复这个错误。如果设备控制器处理不了这个问题，那么设备驱动程序应该进行处理，设备驱动程序会再次尝试读取操作，很多错误都是偶然性的，如果设备驱动程序无法处理这个错误，才会把错误向上抛到硬件层面（上层）进行处理，很多时候，上层并不需要知道下层是如何解决错误的。这就很像项目经理不用把每个决定都告诉老板；程序员不用把每行代码如何写告诉项目经理。这种处理方式不够透明。

#### 同步和异步传输

I/O 软件实现的第三个目标就是 同步(synchronous) 和 异步(asynchronous，即中断驱动)传输。这里先说一下同步和异步是怎么回事吧。

同步传输中数据通常以块或帧的形式发送。发送方和接收方在数据传输之前应该具有同步时钟。而在异步传输中，数据通常以字节或者字符的形式发送，异步传输则不需要同步时钟，但是会在传输之前向数据添加奇偶校验位。下面是同步和异步的主要区别

回到正题。大部分物理IO(physical I/O) 是异步的。物理 I/O 中的 CPU 是很聪明的，CPU 传输完成后会转而做其他事情，它和中断心灵相通，等到中断发生后，CPU 才会回到传输这件事情上来。

I/O 分为两种：物理I/O 和 逻辑I/O(Logical I/O)。

物理 I/O 通常是从磁盘等存储设备实际获取数据。逻辑 I/O 是对存储器（块，缓冲区）获取数据。

缓冲

I/O 软件的最后一个问题是缓冲(buffering)。通常情况下，从一个设备发出的数据不会直接到达最后的设备。其间会经过一系列的校验、检查、缓冲等操作才能到达。举个例子来说，从网络上发送一个数据包，会经过一系列检查之后首先到达缓冲区，从而消除缓冲区填满速率和缓冲区过载。

共享和独占

I/O 软件引起的最后一个问题就是共享设备和独占设备的问题。有些 I/O 设备能够被许多用户共同使用。一些设备比如磁盘，让多个用户使用一般不会产生什么问题，但是某些设备必须具有独占性，即只允许单个用户使用完成后才能让其他用户使用。

下面，我们来探讨一下如何使用程序来控制 I/O 设备。一共有三种控制 I/O 设备的方法

使用程序控制 I/O

使用中断驱动 I/O

使用 DMA 驱动 I/O

使用程序控制 I/O

使用程序控制 I/O 又被称为 可编程I/O，它是指由 CPU 在驱动程序软件控制下启动的数据传输，来访问设备上的寄存器或者其他存储器。CPU 会发出命令，然后等待 I/O 操作的完成。由于 CPU 的速度比 I/O 模块的速度快很多，因此可编程 I/O 的问题在于，CPU 必须等待很长时间才能等到处理结果。CPU 在等待时会采用轮询(polling)或者 忙等(busy waiting) 的方式，结果，整个系统的性能被严重拉低。可编程 I/O 十分简单，如果需要等待的时间非常短的话，可编程 I/O 倒是一个很好的方式。一个可编程的 I/O 会经历如下操作

CPU 请求 I/O 操作

I/O 模块执行响应

I/O 模块设置状态位

CPU 会定期检查状态位

I/O 不会直接通知 CPU 操作完成

I/O 也不会中断 CPU

CPU 可能会等待或在随后的过程中返回

使用中断驱动 I/O

鉴于上面可编程 I/O 的缺陷，我们提出一种改良方案，我们想要在 CPU 等待 I/O 设备的同时，能够做其他事情，等到 I/O 设备完成后，它就会产生一个中断，这个中断会停止当前进程并保存当前的状态。一个可能的示意图如下

尽管中断减轻了 CPU 和 I/O 设备的等待时间的负担，但是由于还需要在 CPU 和 I/O 模块之前进行大量的逐字传输，因此在大量数据传输中效率仍然很低。下面是中断的基本操作

CPU 进行读取操作

I/O 设备从外围设备获取数据，同时 CPU 执行其他操作

I/O 设备中断通知 CPU

CPU 请求数据

I/O 模块传输数据

所以我们现在着手需要解决的就是 CPU 和 I/O 模块间数据传输的效率问题。

使用 DMA 的 I/O

DMA 的中文名称是直接内存访问，它意味着 CPU 授予 I/O 模块权限在不涉及 CPU 的情况下读取或写入内存。也就是 DMA 可以不需要 CPU 的参与。这个过程由称为 DMA 控制器（DMAC）的芯片管理。由于 DMA 设备可以直接在内存之间传输数据，而不是使用 CPU 作为中介，因此可以缓解总线上的拥塞。DMA 通过允许 CPU 执行任务，同时 DMA 系统通过系统和内存总线传输数据来提高系统并发性。

## 分类

### 内存I/O

### 网络I/O

### 磁盘I/O

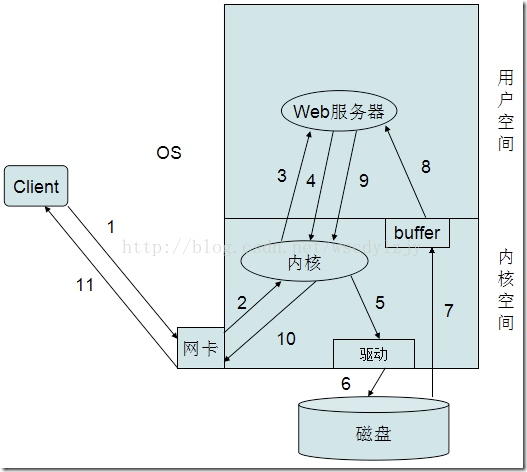
### ----

### 直接I/O

### 标准I/O/缓存I/O

### mmap

## Web请求处理过程



1.客户端发起情况到服务器网卡

2.服务器网卡接受到请求后转交给内核处理

3.内核根据请求对应的套接字，将请求交给工作在用户空间的Web服务器进程

4.Web服务器进程根据用户请求，向内核进行系统调用，申请获取相应资源（如：客户端获取图片）

5.内核发现Web服务器进程请求的是一个存放在本地硬盘上的资源，因此通过驱动程序连接磁盘

6.内核调用磁盘，获取需要的资源

7.内核将资源存放在自己的缓存区中，并通知Web服务器进程

8.Web服务器进程通过系统调用取得资源，并将其复制到进程自己的缓冲区中

9.Web服务器进程形成响应，通过系统调用再次发给内核以响应请求

10.内核将响应发送至网卡

11.网卡发送响应给用户

通过这样的一个复杂过程，一次请求就完成了

简单来说就是：

用户请求—>送达用户空间—>系统调用—>内核空间—>内核到磁盘上读取图片资源—>返回到用户空间—>响应给用户

上述简单的说明了一下，客户端向Web服务器请求过程，在这个过程中，有两个I/O过程：一是客户端请求的网络I/O，二个是Web服务器请求图片磁盘I/O。

# IO函数

## I/O读写函数

### recv

使用recv函数接收数据：

#include <sys/socket.h>

ssize\_t recv(int fd,void\* buf,size\_t len,int flags);

其中，buf是用户接收缓冲区的地址，len表示接收数据长度，recv函数从内核的接收缓冲区中复制数据到用户指定的缓冲区中：当内核中的数据比指定的缓冲区小时，一般会复制内核缓冲区的所有数据到用户缓冲区，并返回数据的长度；当内核接收缓冲区的数据比用户指定的长度多时，会将用户指定len长度的数据复制到用户缓冲区，其余数据等下次调用的时候复制。内核在复制完内核缓冲区的数据后，会销毁该数据。

其中flag的取值：

 recv函数返回错误时，错误码的含义：

recv函数的特点：

1. recv()常用于TCP连接，recvfrom()用于UDP连接（在数据报套接字DUP绑定了地址和端口后，也可以使用recv接收数据）；
2. 可以使用flag参数使其适用于多种场景；
3. 需要根据返回值来判断数据是否接收完毕。

### send

使用send函数发送数据：

#include <sys/socket.h>

ssize\_t send(int fd,const void\* buf,size\_t len,int flags);

send()函数的特点：

需要根据返回值来判断数据是否发送完毕

send()用于处理连接状态的描述符

flag标志位使得其行为多样

send函数errno含义：



### readv

#include <sys/uio.h>

ssize\_t readv(int fd,struct iovec\* vector,int count);

ssize\_t writev(int fd,const struct iovec\* vector,int count);

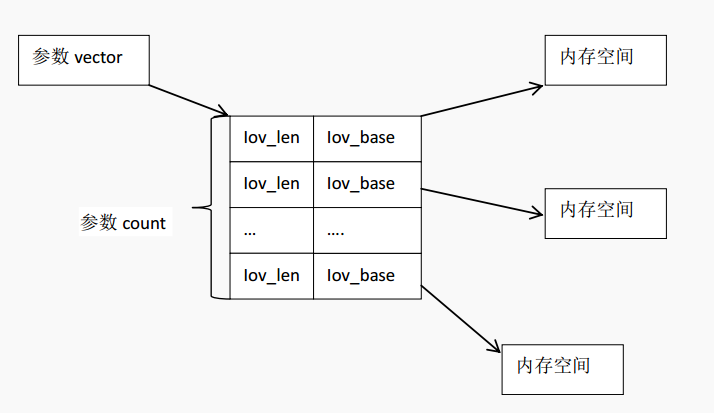
这两个函数的第二个参数都是一个指向某个iovec结构数组的一个指针：

struct iovec{

void\* iov\_base; //缓冲开始地址

size\_t iov\_len; //缓冲长度

};



### writev

### recvmsg

#include <sys/socket.h>

ssize\_t recvmsg(int fd,struct msghdr\* msg,int flags);

ssize\_t sendmsg(int fd,struct msghdr\* msg,int flags);

### sendmsg

### I/O函数对比



## I/O操作函数

### pipe

创建管道实现进程间通信函数pipe()：

int pipe(int fd[2]);

使用pipe()函数需要注意的事项：

fd[0]用于读取数据，fd[1]用于写数据

fd[0]和fd[1]默认情况下为阻塞

关闭管道的依据是引用计数

写失败触发SIGPIPE信号

pipe()函数创建管道的特性：

管道内部传输的数据流是字节流

单向管道

管道容量有限制

基于上述缺陷，内核提供了sockerpair创建双向的管道

### socketpair

创建双向管道函数socketpair()：

int socketpair(int domain,int type,int protocol,int fd[2]);

### mmap

创建共享内存函数mmap()：

void\* mmap(void\* start,size\_t length,int prot,int flags,int fd,off\_t

offset);

int munmap(void\* start,size\_t length);

mmap()函数port参数用来设置内存段的访问权限：

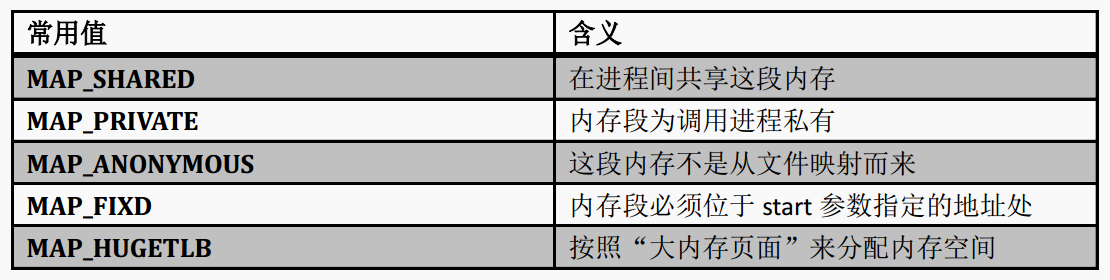
PROT\_READ：内存段可读

PORT\_WRITE：内存段可写

PORT\_EXEC：内存段可执行

PORT\_NONE：内存段不能被访问

参数flag的取值：



### fcntl

用于修改描述符属性的函数fcntl()：

int fcntl(int fd,int cmd,…);

使用fcntl函数获取和设置文件描述符的状态标志：

F\_GETFL：获取fd的状态标志

F\_SETFL：设置fd的状态标志

注：ioctl比fcntl有更多控制行为，但是对于文件描述符的控制，使用fncntl。

### splice

用于两个文件描述符之间移动数据的函数splice()：

ssize\_t splice(int fd\_in,loff\_t\* off\_in,int fd\_out,loff\_t\*

off\_out,size\_t len,unsigned int flags);

flag参数的含义：



使用splice()函数的注意事项：

fd\_in和fd\_out必须至少有一个是管道文件描述符

返回值为0时，表示没有数据需要移动

### 实例

#include <sys/socket.h>

#include <sys/types.h>

#include <stdlib.h>

#include <stdio.h>

int main()

{

int fd[2];

int ret = socketpair(AF\_UNIX,SOCK\_STREAM,0,fd);

if(ret < 0)

{

perror("socketpair() error\r\n");

exit(1);

}

if(fork())

{

printf("parent process \r\n");

int val = 0;

close(fd[1]);

while(1)

{

sleep(1);

++val;

printf("send val to child : %d\r\n",val);

write(fd[0],&val,sizeof(val));

read(fd[0],&val,sizeof(val));

printf("read val from child : %d \r\n",val);

}

}

else

{

printf("child process \r\n");

int val;

close(fd[0]);

while(1)

{

read(fd[1],&val,sizeof(val));

++val;

write(fd[1],&val,sizeof(val));

}

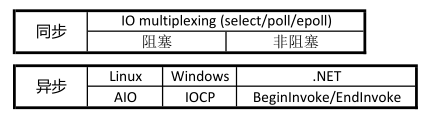
}

}

# IO模型

参考：

<https://blog.csdn.net/z_ryan/article/details/80873449>



## 阻塞式I/O

阻塞和非阻塞指的是执行一个操作时等操作结束再返回结果，还是马上返回结果。

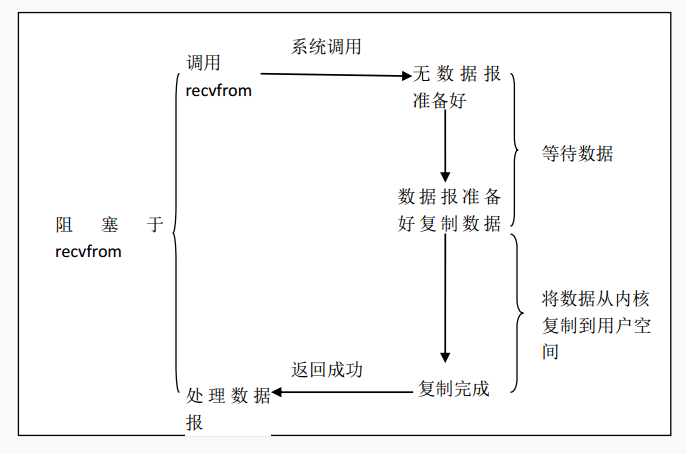
阻塞（blocking）：指IO操作需要彻底完成后才返回到用户空间，调用结果返回之前，调用者被挂起（当前线程进入非可执行状态，在这个状态，CPU不会分配时间片，线程暂停运行）只有到到结果才进入活动状态；

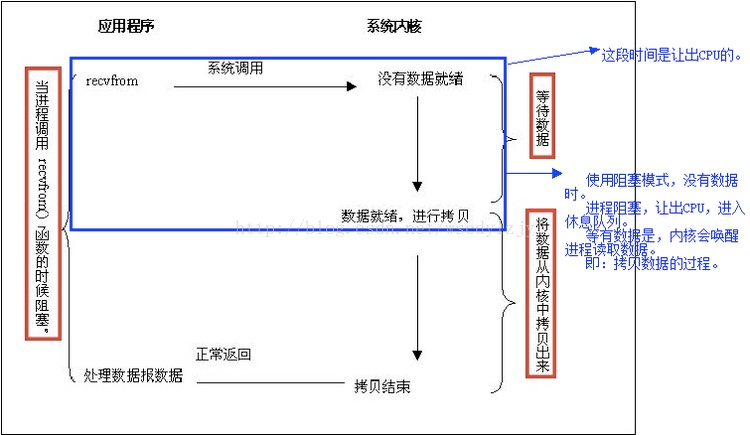
阻塞例子：海底捞的服务器为你点菜，当你点完菜后，服务员把消息传到后厨，这时你就在餐桌上等待，直到厨师把汤锅和配菜都准备好以后送到你桌上，你才能开吃。在上菜的过程中你还不能离开，因为你离开了之后服务员上菜了却找不到你人，所以你就是能等待，这个时候你处于阻塞等待状态，就是前面说的，你是调用者，你被挂起了，进入了非可执行状态。

非阻塞（nonblocking）：指I/O操作被调用后**立即返回给用户一个状态值**，无需等到I/O操作彻底完成，最终的调用结果返回之前，调用者不会被挂起；

非阻塞例子：海底捞的服务器为你点菜，当你点完菜后，服务员把消息传到后厨，过了三分钟，你跑到后厨问，我的锅底或者肥牛卷好了没有？后厨说没好，然后你去处理其它事情，然后又过了五分钟，你又跑到后厨问，我的某个菜好了没有，如果没有，你还是继续做其他事情，然后等会再问一次，这个时候就是在I/O操作的同时，你没有被挂起，可以操作其他事情，但是如果I/O操作完成，你需要立马接受。

### 原理





说明：应用程序调用一个IO的recvfrom函数，会导致应用程序阻塞，进入阻塞状态后直到I/O操作结束才会返回；如果系统内核数据没有准备好，那就一直等待数据准备，因为是调用了recvfrom函数导致了应用程序阻塞，所以一直在等，做不了任何事情，内核数据准备好之后把数据从内核拷贝到用户空间，拷贝结束后，I/O函数返回成功指示。

注：其阻塞时在I/O操作阶段。

### 分类

可能阻塞套接字的系统调用分为以下几类：

输入操作中的read()、readv()、recv()、recvfrom()

输出操作中的write()、writev()、send()、sendto()

接收连接中的accept()

发起连接中的connect()

### 特点

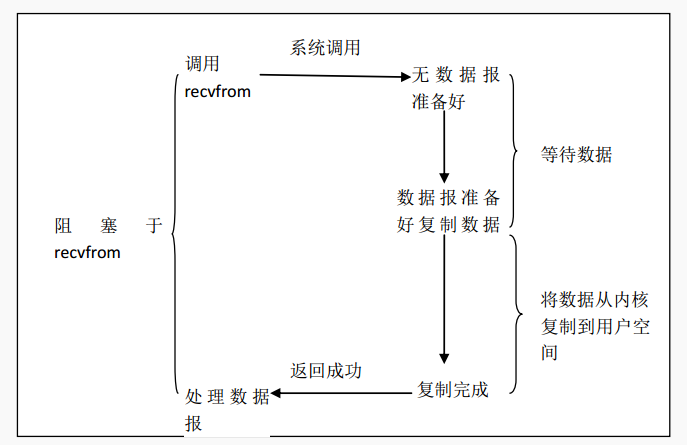
阻塞式I/O的缺点：

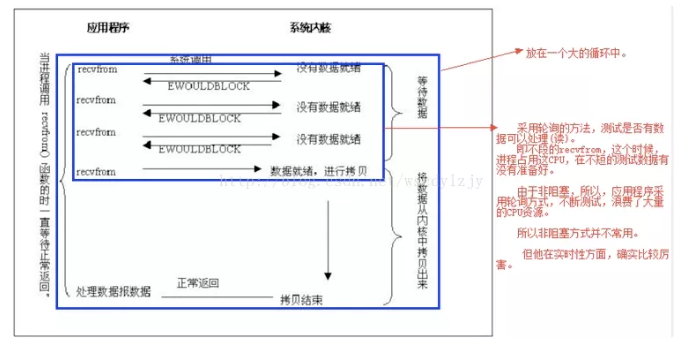
效率低

不适合高并发的开发

## 非阻塞式I/O

### 原理





说明：用户线程发起IO请求时立即返回。但并未读取到任何数据，则返回字段为“EWOULDBLOCK”，用户线程需要不断地发起IO请求，直到数据到达后，才真正读取到数据，继续执行。即“轮询”机制。整个IO请求过程中，虽然用户线程每次发起IO请求后可以立即返回，但是为了等到数据。仍需要不断地轮询、重复请求、消耗了大量的CPU资源；是比较浪费CPU的方式，一般很少用这种模型，而是在其他模型中使用非阻塞IO这一特性。

在读某些文件时，如果文件没有数据，往往会导致读操作阻塞（休眠），比如：

1. 读鼠标、键盘灯字符设备文件；
2. 读管道文件（有名/无名）

读普通文件时，如果读到了数据就成功返回，如果没有读到数据返回0，总之不会阻塞。

在写某些文件时，当文件不能立即接收写入的数据时，也可能会导致写操作阻塞，一直阻塞到写成功为止。一般来说，写文件不会阻塞，因此我们不考虑写文件阻塞的情况。

由于一般情况下，“读鼠标”和“读键盘”都是阻塞的，为了不要让“读鼠标”和“读键盘”因为阻塞而相互干扰，可以采取如下办法读：

1. fork子进程，然后父子进程两线任务

父进程：读键盘

子进程：读鼠标

这种方式肯定是没问题的，这仅是一种方案，实际并不提倡这么做，多线任务时使用多进程实现，开销太大了。

1. 创建次线程，主线程和次线程两线任务

主线程：读键盘

次线程：读鼠标

这种方式是我们经常实现的方式。

1. 将鼠标和键盘设置为“非阻塞”，while轮询读
2. IO多路复用

**阻塞是好还是坏？**

实际上读文件因为没有数据而阻塞，其实是好事，因为这样就进入休眠状态，休眠时就不会占用CPU，节省了CPU的资源。

我们可以将阻塞的读修改为非阻塞的读。非阻塞读意思就是，如果有数据就成功读到，如果没有读到数据就出错返回，而不是阻塞。

尽管我们很少非阻塞的读，但是有些时候还是需要非阻塞读，因此OS还是提供了非阻塞操作方式。

**如何实现非阻塞读？**

1. 打开文件时指定O\_NONBLOCK状态标志；、

在IPC有名管道，如果不希望阻塞的话，就可以在open打开“有名管道”时，指定O\_NONBLOCK，然后读有名管道无数据时就不会阻塞。

1. 通过fcntl函数指定O\_NONBLOCK来实现；

什么情况下使用fcntl来实现：

1. 当文件已经被open打开了，但是open是并没有指定你要的文件状态标志，而你又不想去修改open的参数，此时可以使用fcntl来重设或者补设；
2. 没办法在open指定，你手里只有一个文件描述符fd，此时就使用fcntl来重设或补设：比如无名管道，无名管道连名字都没有，没有办法使用open函数，无名管道是使用pipe函数来返回文件描述符的，如果你想非阻塞的读无名管道的话，是没有办法通过open来指定O\_NONBLOCK的，此时就需要使用fcntl来重设或补设。

当然我们使用fcntl不仅仅只能重设或补设O\_NONBLOCK，也可以重设或补设O\_TEUNC/O\_APPEND等任何你需要的“文件状态”标志。

设置两种方式：

1. 重设

fcntl(0, F\_SETFL, O\_READONLY|O\_NONBLOCK);

1. 补设

flag = fcntl(0, F\_GETFL); //获取原有文件状态标志

flag = O\_NONBLOCK; //通过|操作，在已有标志上增加O\_NONBLOCK

fcntl(0, F\_SETFL, flag);

### 非阻塞accept

struct linger的l\_onoff标志设为1，l\_linger设为0，关闭TCP连接会先在socket上发送一个RST包：

select返回监听socket可读

服务器接收到客户发送过来的RST

已经完成的连接被从队列中删除

服务器调用accept，但是连接已经消息

上述问题的解决方式：

监听socket设置为非阻塞模式

忽略EWOULDBLOCK错误

忽略FCONNABORTED错误

忽略EPROTO错误

### 非阻塞connect

非阻塞connect的用途：

把三次握手叠加到其他处理上

可以同时建立多个连接

借助I/O复用，实现connect的超时

**使用非阻塞connect的时候需要注意：**

需要处理非阻塞连接本机的情况

在Berkeley的实现中，连接建立成功时，套接字描述符变为可写

在Berkeley的实现中，连接出错时，套接字描述符变为既可读又可写

三次握手过程中发生中断

**移植非阻塞的connect时需要注意的地方：**

对出错的套接字的不同处理

有可能在调用select之前，连接就已经建立成功，而且对方的数据已经到来

**移植非阻塞connect时问题的解决方法：**

调用getpeername代替getsockopt

调用read，读取长度为0字节的数据

再调用一次connect

**非阻塞connect的步骤：**

创建socket，返回套接字描述符

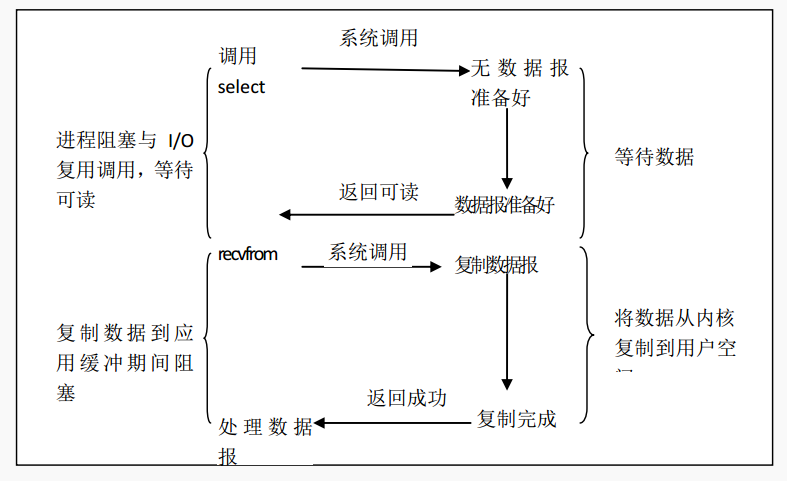
调用fcntl把套接字描述符设置为非阻塞

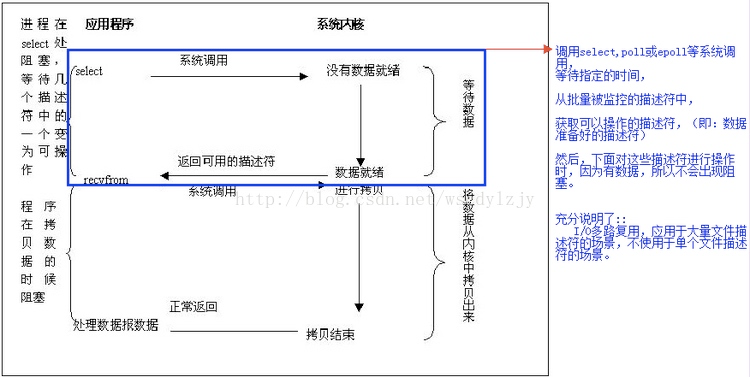
调用connect开始建立连接

判断连接是否成功建立（需要处理前面提到的问题）

## I/O复用

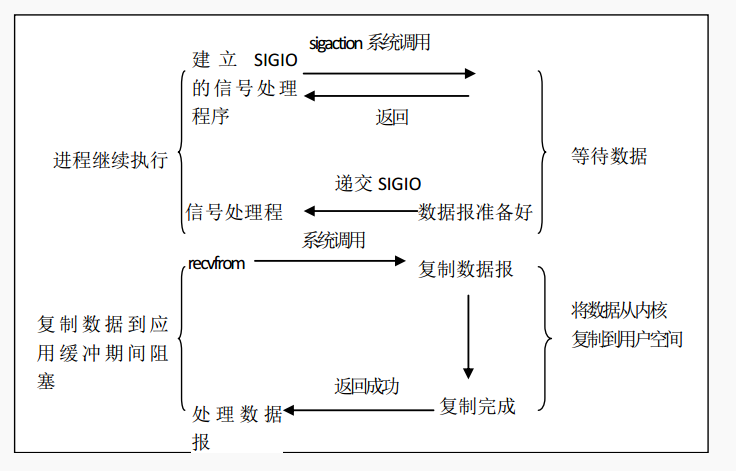
多线程需要进行上下文切换，故而引入IO多路复用。

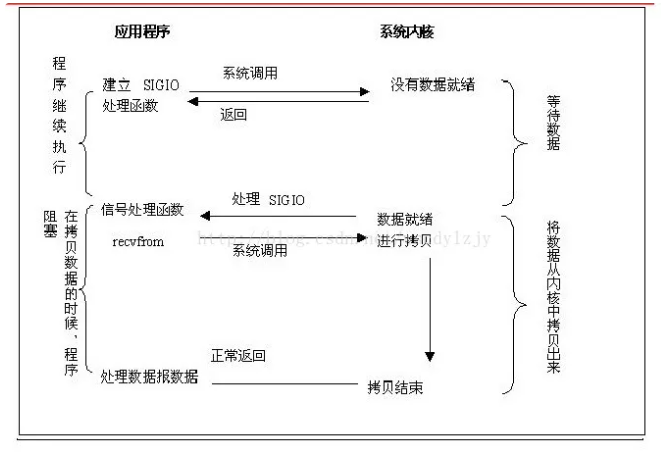




说明：I/O复用模型会用到select或poll函数，在I/O复用模型中，并不是阻塞到I/O操作过程中，而是阻塞到select或者poll函数中；以select为例：进程在select处阻塞，等待几个描述符中的一个变为可操作，如果没等待到就继续阻塞在第一阶段，如果等到了一个描述符变为了可操作，则调用recvfrom函数将数据拷贝到应用缓冲区。

## 信号驱动式I/O





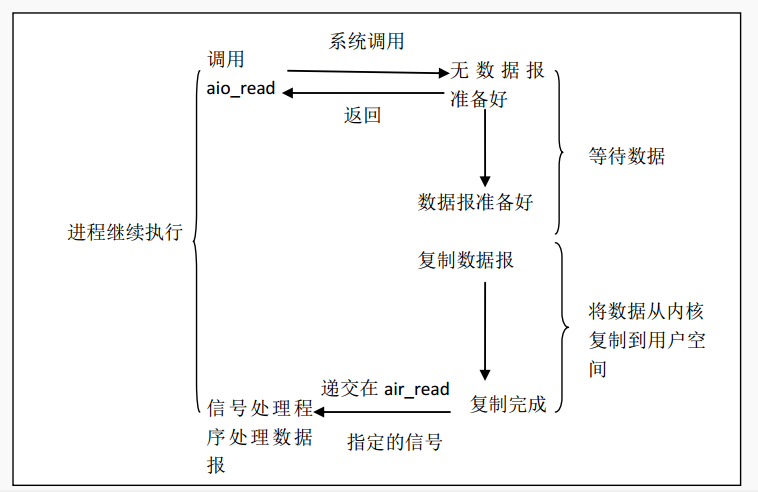
说明：首先，我们允许套接口进行信号驱动I/O，并安装一个信号处理函数SIGIO，如果数据没有准备好，则立即返回结果，进程继续工作并不阻塞。当数据准备好时，系统内核会主动发送一个SIGIO信号给应用程序，应用程序收到信号后，可以在信号处理函数中调用I/O操作函数recvfrom进行数据处理。信号驱动I/O模型的优点是当数据报到达时，可以不阻塞，主循环可以继续执行，只是等待处理程序的通知，或者数据已经准备好被处理，或者数据报已经准备好被读了。

## 异步I/O

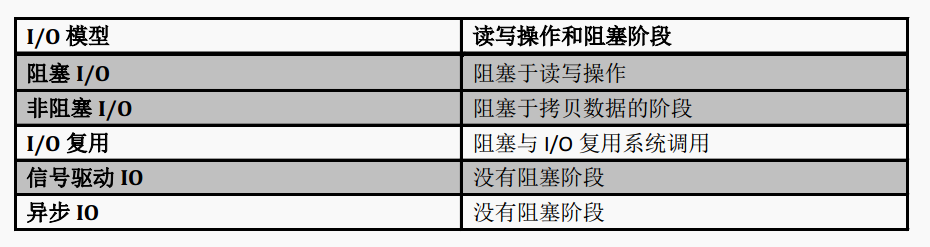
同步（synchronous）：调用者等待被调用者返回消息，才能继续执行。同步阻塞例子：去餐馆吃饭，点了一个盖浇饭，然后在餐桌上一直等到盖浇饭做好，自己端到餐桌就餐。这就是典型的同步阻塞。当厨师给你做饭的时候，你需要一直在那里等着。

同步非阻塞例子：去餐馆吃饭，点了一个盖浇饭，你点完饭之后，过了几分钟感觉时间差不多了，就去问老板饭做好了没有，如果好了就去端，如果没好等一会再去问，实时同步做饭进度，依次循环去问直到饭做好，这就是同步非阻塞。

异步（asynchronous）：被调用者通过状态、通知或回调机制主动通知调用者被调用者的运行状态。



## I/O模型比较



### 同步/异步

同步和异步与**消息的通知机制**有关：

同步：在发出一个功能调用时，在没有得到结果之前，该调用就不返回

异步：异步过程调用发出后，调用者不能立刻得到结果

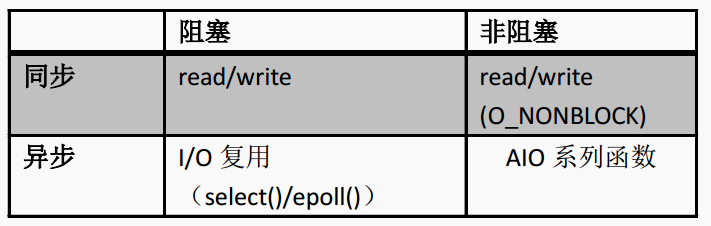
异步调用，要想获得结果，一般由两种方式：

1. 主动轮询异步调用的结果；
2. 被调用方通过callback来通知调用方调用结果。

同步和异步的使用特点：

关注的是消息如何通知的机制

异步机制中，消息通知和消息处理之间需要一个桥梁



### 阻塞/非阻塞

阻塞和非阻塞和等待消息时的状态有关：

阻塞：函数只有在得到结果之后才会返回

非阻塞：不能立刻得到结果的情况下，调用会立刻返回

另外一种解释：

阻塞和非阻塞的重点在于进程/线程等待消息时候的行为，也就是在等待消息的时候，当前进程/线程是挂起状态还是非挂起状态。

阻塞调用在发出去后，在消息返回之前，当前进程/线程会被挂起，直到有消息返回，当前进程/线程才会被激活。

非阻塞调用在发出去后，不会阻塞当前进程/线程，而是立即返回。

### 总结

1. 同步与异步，重点在于消息通知的方式；
2. 阻塞与非阻塞，重点在于等待消息时候的行为。

四种组合方式：

1. 同步阻塞：
2. 同步非阻塞：
3. 异步阻塞：
4. 异步非阻塞：

# 文件锁

文件锁也被称为记录锁，是用来保护文件数据的。

当多个进程共享读写同一个文件时，为了不让进程各自读写数据时相互干扰，我们可以使用进程信号量来互斥实现，除了可以使用进程信号量以外，还可以使用“文件锁”来实现，而且功能更加丰富，使用相对容易。

多进程共享读写同一文件时，如果数据很重要的话，为了防止数据相互修改，应该满足如下读写条件：

1. 写与写应该互斥

当某个进程正在写文件，而且在数据没有写完时，其他进程不能写，否则会相互打乱对方写的数据。

1. 读与写应该互斥

分两种情况：

1. 某个进程正在写操作，而且在数据没有写完时，其他进程不能读数据。因为别人在没有写完之前，读到的数据是不完整的，所以读和写是互斥的。
2. 某个进程正在读数据，在数据没有读完之前，其他进程不能写数据。因为可能会扰乱别人读到的数据。
3. 读与读共享

某个进程在读数据时，就算数据没有读完，其他进程也可以共享读数据，并不需要互斥等待别人读完后才能读取。因为读文件时不会修改文件的内容，所以不用担心数据相互干扰的问题。

总结起来就是，多进程读写文件时，如果你想进行资源保护的话，完美的资源保护应该满足以下条件：

1. 写与写之间互斥；
2. 读与写之间互斥；
3. 读与读之间共享。

如何实现以上读写要求？

如果使用信号量实现保护的话，只能是一律互斥，包括读与读都是互斥的，不能够同时实现互斥和共享，但是文件锁可以做到。

# 存储映射

## 概述

存储映射是一个磁盘文件与存储空间的一个缓冲相映射，对缓冲数据的读写就相应的完成了文件的读写。

文件存储映射操作函数：mmap，munmap

## mmap

void \*mmap(coid\* addr,size\_t len, int port, int flag, int fields, off\_t off);

参数：

addr：存储映射区的起始地址，通常设为0

fileds：文件描述符

len：需要映射的字节数

off：映射字节在文件中的偏移量

port：

PORT\_READ映射区可读

PORT\_WAITE映射区可写

PORT\_EXEC映射区可执行

PORT\_NONE映射区不可访问

flag：

MAP\_FIXED返回地址必须等于addr，不鼓励使用

MAP\_SHARED存储操作立即修改映射文件内容

MAP\_PRIVATE存储操作导致创建映射文件的副本，并对副本进行读写

返回：成功返回0，出错返回MAP\_FAILED

总结：IO模型，存储映射，文件锁都属于文件高级操作。