# 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);

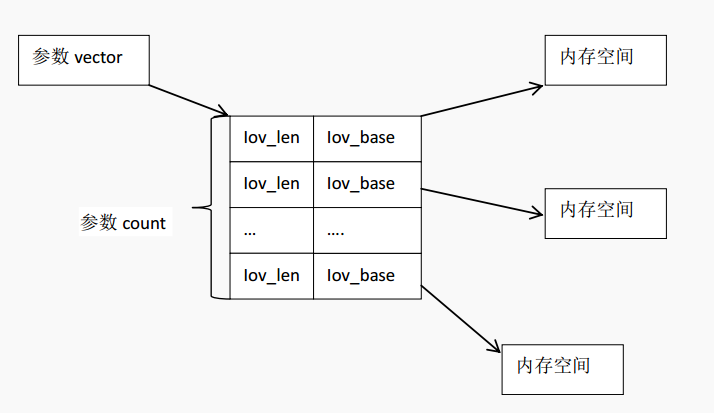
这两个函数的第二个参数都是一个指向某个iovec结构数组的一个指针：

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：

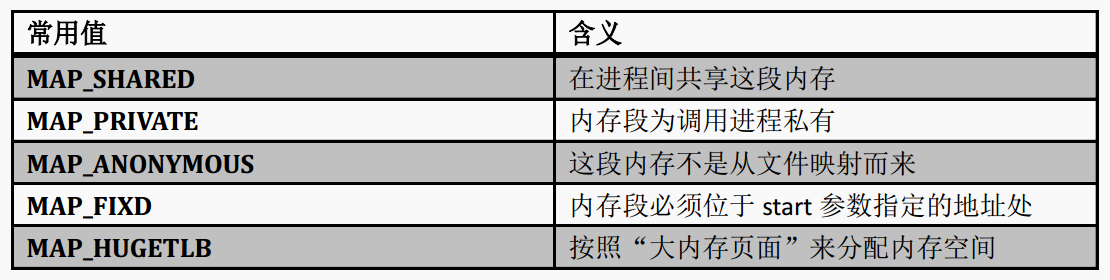
PROT\_READ：内存段可读

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));

}

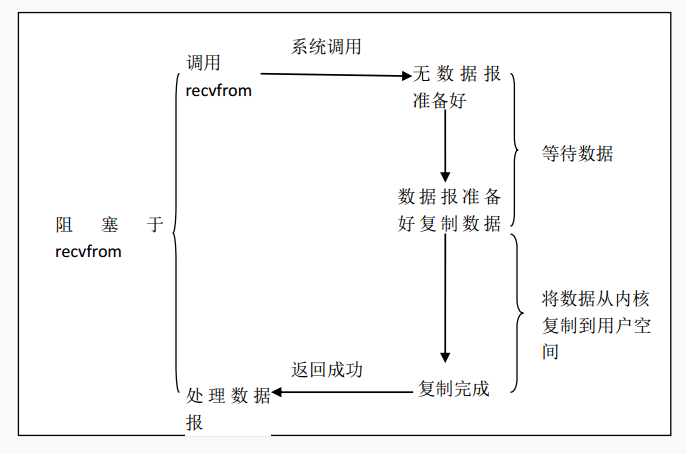
}

}

# IO模型

## 阻塞式I/O

### 原理



### 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

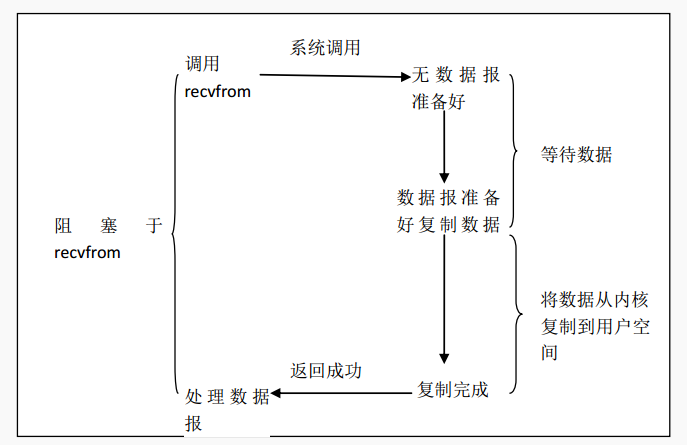
阻塞式I/O的缺点：

效率低

不适合高并发的开发

## 非阻塞式I/O

### 原理



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

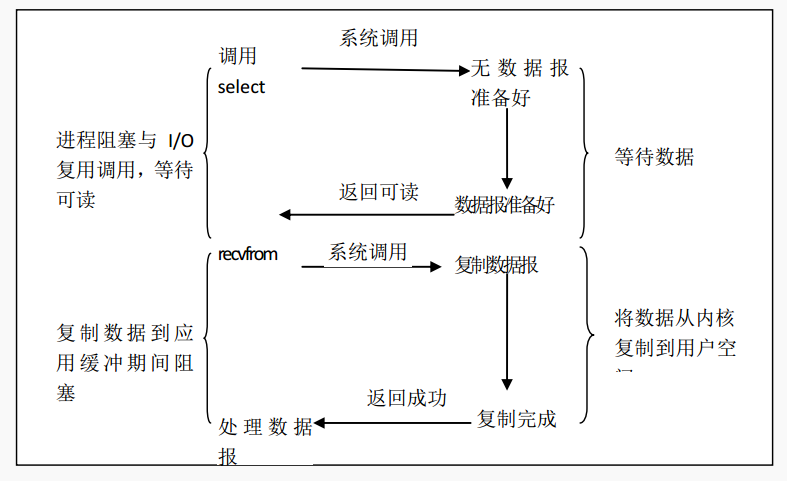
调用fcntl把套接字描述符设置为非阻塞

调用connect开始建立连接

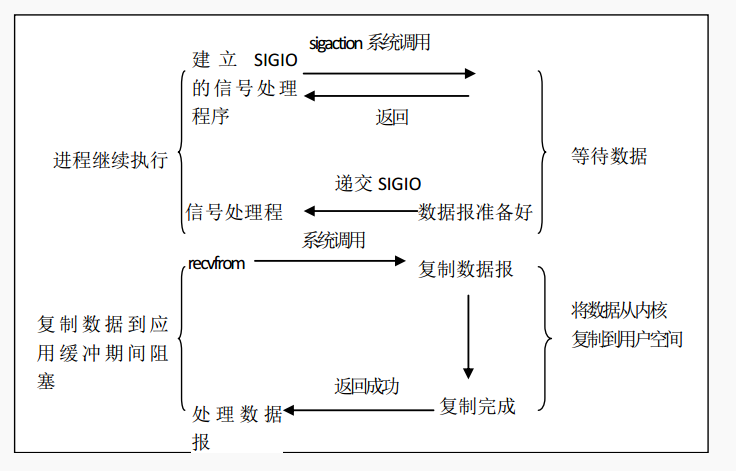
判断连接是否成功建立（需要处理前面提到的问题）

## I/O复用

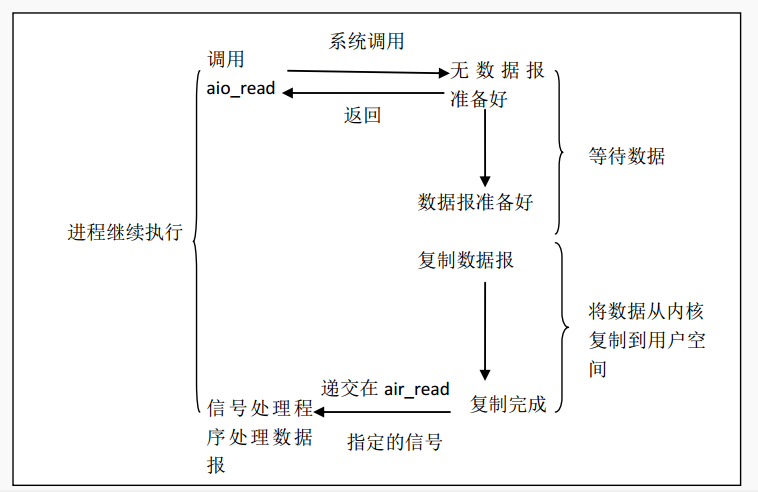
多线程需要进行上下文切换，故而引入IO多路复用。



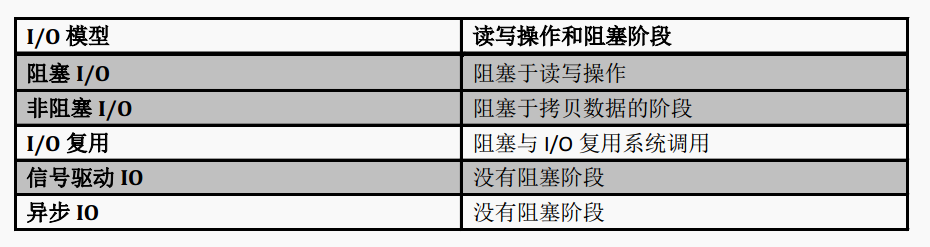
## 信号驱动式I/O



## 异步I/O



## 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

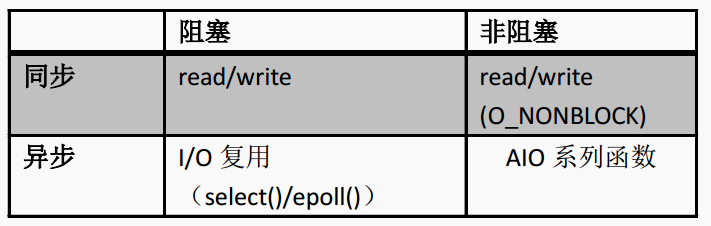
异步调用，要想获得结果，一般由两种方式：

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。