1.进程是资源的分配单位，线程是CPU的执行单位。进程内的线程除了应该拥有一

部分自己专属的资源以外，还共享这进程的剩下资源。

2.操作系统课程里面介绍的进程有5种状态：new、ready、running、pend

（waiting）、dead。少了一种suspend。

下面我谈一下suspend挂起和pend阻塞的区别：

1.挂起是主动的（自然对应的唤醒也是主动的），阻塞是被动的（是因为

资源的无法满足：等待IO（本地或者网络交互）、等待数据（比如说等其他进程的数据）等等）。

2.挂起是把内存上的写入磁盘，自然不占用CPU；阻塞是释放CPU，程序回

到装入内存的状态，就是只占有内存而不占有CPU（老师讲过存在特殊情况就是如果等待 的数据可以很快到来那么就会短时间内一直占有CPU）。

3.同步：整个执行流程像单线程一样

异步：多线程加信号（事件）机制（异步与单纯的多线程还是有点区别的，因为异步是分叉对数据进行计算，主支需要分支的数据，单纯的多线程缺少这种数据的依赖关系）之后异步可以靠信号、回调函数来得到结果。

你叫我吃饭，我听到了就立刻和你去吃饭，如果我没听到，你就会一直叫我，直到我听见和你一起去吃饭，这个过程叫同步；异步过程指你叫我吃饭，然后你就去吃饭了，而不管我是否和你一起去吃饭（我在完成事情以后才能用实际表明自己会不会去吃饭，在这段时间里面你做了其他的事情，所以也就是你不能立刻得到结果）。而我得到消息后可能立即就走，也可能过段时间再走。

如果我请你吃饭，就是同步，如果你请我吃饭就用异步，这样你比较省钱。哈哈哈。。。

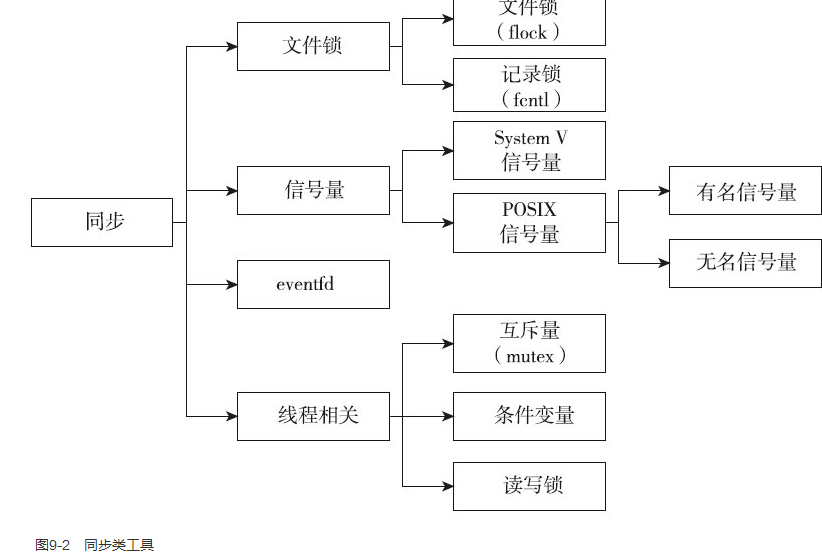
阻塞就是一个人顿在那里了，所以请你明白同步、异步与阻塞、非阻塞之间的区别。

阻塞：进程等待什么，放弃了CPU时间

非阻塞：持续占用CPU时间

同步与异步是对于事件发展的相关性来讲，阻塞与非阻塞是对线程来讲。

进程同步与线程同步：



还有CAS（java里面已经有对应的函数与很多使用），总结起来有一般的变量锁（信号量、互斥量、条件变量、读写锁）、文件锁、CAS、事件。

Linux下有5种阻塞模型：

1)阻塞I/O（blocking I/O）  
2)非阻塞I/O （nonblocking I/O）  
3) I/O复用(select 和poll) （I/O multiplexing）  
4)信号驱动I/O （signal driven I/O (SIGIO)）  
5)异步I/O （asynchronous I/O (the POSIX aio\_functions)）

阻塞I/O：

它的过程有点像终端与命令行一样，调用某个功能后阻塞自己知道功能返回。在网络编程里面就是：

 1．输入操作： recv()、recvfrom()、WSARecv()和WSARecvfrom()函数。以阻塞套接字为参数调用该函数接收数据。如果此时套接字缓冲区内没有数据可读，则调用线程在数据到来前一直睡眠。

    2．输出操作： send()、sendto()、WSASend()和WSASendto()函数。以阻塞套接字为参数调用该函数发送数据。如果套接字缓冲区没有可用空间，线程会一直睡眠，直到有空间。

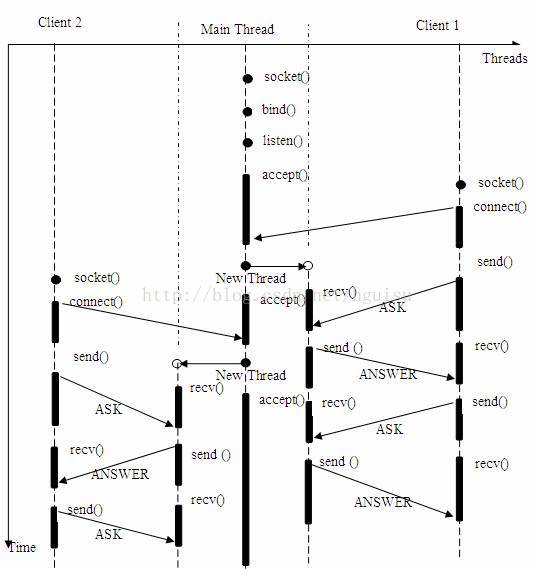
    3．接受连接：accept()和WSAAcept()函数。以阻塞套接字为参数调用该函数，等待接受对方的连接请求。如果此时没有连接请求，线程就会进入睡眠状态。

   4．外出连接：connect()和WSAConnect()函数。对于TCP连接，客户端以阻塞套接字为参数，调用该函数向服务器发起连接。该函数在收到服务器的应答前，不会返回。这意味着TCP连接总会等待至少到服务器的一次往返时间。

在调用recv()/recvfrom（）函数时，发生在内核中等待数据和复制数据的过程。从系统缓冲区到用户空间。



多线程阻塞：



就是再加上线程池，用完一个线程就回收利用。“线程池”和“连接池”技术也只是在一定程度上缓解了频繁调用 IO 接口（accept()和connect()）带来的资源占用。

非阻塞I/O：

把SOCKET设置为非阻塞模式，即通知系统内核：在调用Windows Sockets API时，不要让线程阻塞，而应该让函数立即返回。在返回时，该函数返回一个错误代码。图所示，一个非阻塞模式套接字多次调用recv()函数的过程。前三次调用recv()函数时，内核数据还没有准备好。因此，该函数立即返回WSAEWOULDBLOCK错误代码。第四次调用recv()函数时，数据已经准备好，被复制到应用程序的缓冲区中，recv()函数返回成功指示，应用程序开始处理数据。



使socket非阻塞的方法有：通过调用ioctlsocket()函数，将该套接字设置为非阻塞模式，还可以使用WSAAsyncselect()和WSAEventselect()函数。Linux下的函数是:fcntl().。

WSAAsyncSocket()

I/O复用模型：

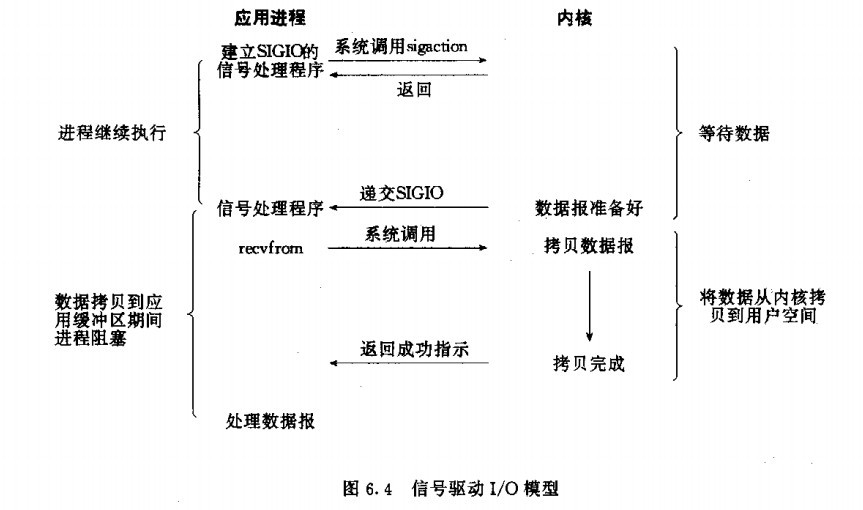
I/O复用模型会用到select、poll、epoll函数，这几个函数也会使进程阻塞(也可以不阻塞，看你时间怎么设置)，但是和阻塞I/O所不同的，虽然也是使用了while(true)循环，但是阻塞的过程中（将等待时间设置为0就不会阻塞），对IO情况是响应式反应。



信号驱动I/O：

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，进程继续运行并不阻塞。当数据准备好时，进程会收到一个SIGIO信号，可以在信号处理函数中调用I/O操作函数处理数据。

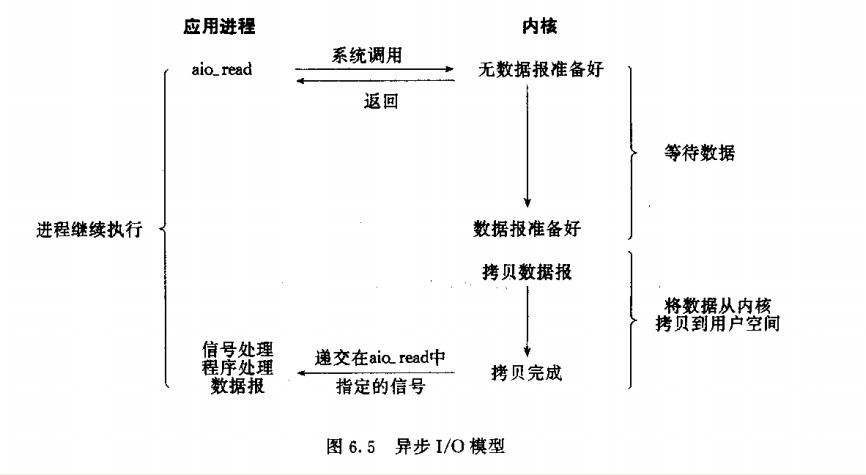
阻塞IO、非阻塞IO、IO复用模型、信号驱动IO都是同步IO，IO复用是对阻塞IO的扩展进行信道复用，信号驱动IO是对同步的简化（去掉了不必要的向内核询问检查，内核告诉他可以receive就是个同步信号）。



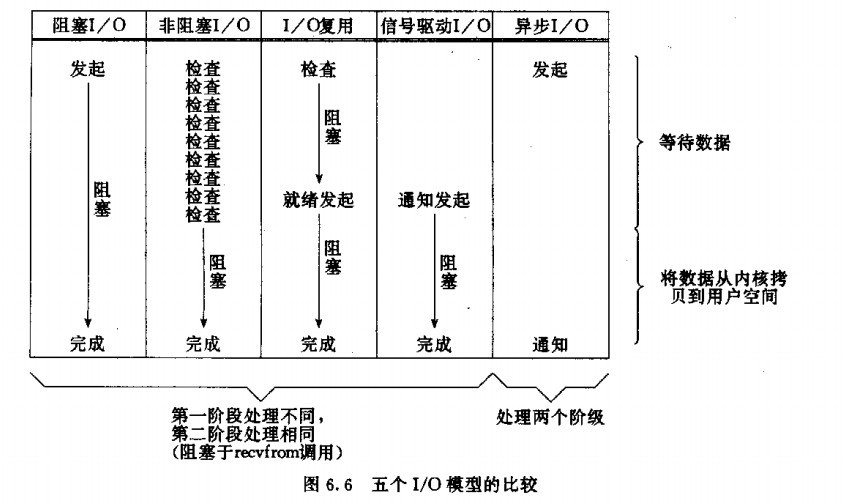
不同于I/O复用模型，这个信号不是自己管理的，所以就不会把自己阻塞掉。

异步I/O：

当一个异步过程调用发出后，调用者不能立刻得到结果（有数据来了但是进程并没有得到数据而是交给内核进行处理）。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者的输入输出操作



五种I/O模型对比：



4.守护进程，也就是通常说的Daemon进程，是Linux中的后台服务进程。它是一个

生存期较长的进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的

事件。守护进程常常在系统引导装入时启动，在系统关闭时终止。Linux系统有很多守护进

程，大多数服务都是通过守护进程实现的，同时，守护进程还能完成许多系统任务，例如，

作业规划进程crond、打印进程lqd等（这里的结尾字母d就是Daemon的意思）。其父进程是用户态的init。

大致步骤：

**创建子进程，父进程退出**：

这是编写守护进程的第一步。由于守护进程是脱离控制终端的，因此，完成第一步后就会在

Shell终端里造成一程序已经运行完毕的假象。之后的所有工作都在子进程中完成，而用户

在Shell终端里则可以执行其他命令，从而在形式上做到了与控制终端的脱离。

在Linux中父进程先于子进程退出会造成子进程成为孤儿进程，而每当系统发现一个孤儿进程时，就会自动由1号进程（init）收养它，这样，原先的子进程就会变成init进程的子进程。

在子进程中创建新会话：

这个步骤是创建守护进程中最重要的一步，虽然它的实现非常简单，但它的意义却非常重大

。在这里使用的是系统函数setsid，在具体介绍setsid之前，首先要了解两个概念：进程组和会话期

进程组：是一个或多个进程的集合。进程组有进程组ID来唯一标识。除了进程号（PID）之外，进程组ID也是一个进程的必备属性。每个进程组都有一个组长进程，其组长进程的进程号等于进程组ID。且该进程组ID不会因组长进程的退出而受到影响。

会话周期：会话期是一个或多个进程组的集合。通常，一个会话开始于用户登录，终止于用

户退出，在此期间该用户运行的所有进程都属于这个会话期。

接下来就可以具体介绍setsid的相关内容：

（1）**setsid函数作用**：

setsid函数用于创建一个新的会话，并担任该会话组的组长。调用setsid有下面的3个作用：

让进程摆脱原会话的控制

让进程摆脱原进程组的控制

让进程摆脱原控制终端的控制

那么，在创建守护进程时为什么要调用setsid函数呢？由于创建守护进程的第一步调用了

fork函数来创建子进程，再将父进程退出。由于在调用了fork函数时，子进程全盘拷贝了父进程的会话期、进程组、控制终端、全局变量（写时复制，我曾想着将指针的值复制过来，但是虽然两边地址是一样的，父子进程地址空间不是同一个）等，虽然父进程退出了，但会话期、进程组、控制终端等并没有改变，因此，这还不是真正意义上的独立开来，而setsid函数能够使进程完全独立出来，从而摆脱其他进程的控制。

**改变当前目录为根目录**：

这一步也是必要的步骤。使用fork创建的子进程继承了父进程的当前工作目录。由于在进程运行中，当前目录所在的文件系统（如“/mnt/usb”）是不能卸载的，这对以后的使用会造成诸多的麻烦（比如系统由于某种原因要进入单用户模式）。因此，通常的做法是让"/"作为守护进程的当前工作目录，这样就可以避免上述的问题，当然，如有特殊需要，也可以把当前工作目录换成其他的路径，如/tmp。改变工作目录的常见函数是chdir。这个函数不会改变调用函数生成文件的那个路径。

**重设文件权限掩码：**

文件权限掩码是指屏蔽掉文件权限中的对应位。比如，有个文件权限掩码是050，它就屏蔽了文件组拥有者的可读与可执行权限。由于使用fork函数新建的子进程继承了父进程的文件权限掩码，这就给该子进程使用文件带来了诸多的麻烦。因此，把文件权限掩码设置为0，可以大大增强该守护进程的灵活性。设置文件权限掩码的函数是umask。在这里，通常的使用方法为umask(0)。

**关闭文件描述符：**

同文件权限码一样，用fork函数新建的子进程会从父进程那里继承一些已经打开了的文件。

这些被打开的文件可能永远不会被守护进程读写，但它们一样消耗系统资源，而且可能导致

所在的文件系统无法卸下。

**重定向stdin、stdout、stderr：**

有人认为对于后台守护进程做此类重定向操作浪费资源，建议直接关闭0、1、2号句   
柄拉倒，这是非常不正确的。假设它们确实被关闭了，则一些普通数据文件句柄将等   
于0、1、2。以2号句柄为例，某些库函数失败后会向2号句柄输出错误信息，这将破   
坏原有数据。

int fd = open( "/dev/null", O\_RDWR ); //缺少验证是否成功

dup2( fd, 0 );

dup2( fd, 1 );

dup2( fd, 2 );

if ( fd > 2 ) {

close( fd );

}

在上面的第二步之后，守护进程已经与所属的控制终端失去了联系。因此从终端输入的字符

不可能达到守护进程，守护进程中用常规方法（如printf）输出的字符也不可能在终端上显

示出来。所以，文件描述符为0、1和2 的3个文件（常说的输入、输出和报错）已经失去了存在的价值，也应被关闭。通常按如下方式关闭文件描述符：

===============================

for(i=0;i<MAXFILE;i++) //这段代码是否需要使用自己看情况

close(i);

===============================

守护进程退出处理：

当用户需要外部停止守护进程运行时，往往会使用 kill命令停止该守护进程。所以，守护

进程中需要编码来实现kill发出的signal信号处理，达到进程的正常退出。

5.整个Linux系统的所有进程也是一个树形结构。树根是系统自动构造的，即在内核态下执行的0号进程，它是所有进程的祖先。由0号进程创建1号进程（内核态），1号负责执行内核的部分初始化工作及进行系统配置，并创建若干个用于高速缓存和虚拟主存管理的内核线程，还负责孤儿进程的收养。随后，1号进程调用execve（）运行可执行程序init，并演变成用户态1号进程，即init进程。它按照配置文件/etc/initab的要求，完成系统启动工作，创建编号为2号...的若干终端注册进程getty。

6.2、Linux的内核守护进程

#ps -aux

Init：用户态系统守护进程，他的进程ID是1，负责启动各运行层次的特定服务。

Keventd：为在内核中运行计划执行的函数提供上下文。

Kswapd：页面调出守护进程。将脏页面低速写到磁盘上，从而使这些页面在需要时仍可回收使用。

portmap：端口映射守护进程。

syslogd：可由操作人员将系统信息记录到日志的任何程序使用。

inetd：侦听系统网络接口，以便取得来自网络的各种网络请求。

nfsd、lockd、rpciod：提供网络文件系统的支持。

crond：指定的日期和时间执行指定的命令。

7.pid\_t fork( void);被称为分叉函数，在父线程里分叉出子线程。fork函数被调用一次但返回两次。两次返回的唯一区别是子进程中返回0值而父进程中返回子进程ID。借助这一点，我们常常会在fork后面加上关于pid和0比较的fi-else结构：

if(pc<0) {

printf("error fork\n");

exit(1);

}else if(pc>0) {

printf(“这是子进程?”);

exit(0);

}else {

Printf(“这才是子进程”);

}

pid\_t 是一个宏定义，其实质是int 被定义在#**include**<*sys/types.h*>中。不过它自己又说实际使用不需要包含这个头文件，我试验过#**include** <unistd.h>确实就够了。这个头文件里面已经包含了这个。

8.在VMware中挂起操作系统就会把他写到磁盘上，即使你重启了，也可以恢复VMware里面的操纵系统。

10.select的使用：

**int select(int maxfdp,fd\_set \*readfds,fd\_set \*writefds,fd\_set \*errorfds,struct timeval \*timeout);**

文件描述符的控制：

.**struct fd\_set**可以理解为一个集合，这个集合中存放的是文件描述符(filedescriptor)，即文件句柄，这可以是我们所说的普通意义的文件，当然Unix下任何设备、管道、FIFO等都是文件形式，全部包括在内，所以毫无疑问一个socket就是一个文件，socket句柄就是一个文件描述符。   
fd\_set集合可以通过一些宏由人为来操作，比如：  
**FD\_ZERO(fd\_set \*);** 清空集合  
**FD\_SET(int, fd\_set \*);** 将一个给定的文件描述符加入集合之中

**FD\_ISSET(int fd,fd\_set \*set)；**是否被触发（也就是可以读写了）   
**FD\_CLR(int, fd\_set\*);** 将一个给定的文件描述符从集合中删除

超时时间控制：

使用的是strutc timeval {

Int tv\_sec,//s

Int tc\_usec//ms

}

第一，若将NULL以形参传入，即不传入时间结构，就是将select置于阻塞状态，一定等到监视文件描述符集合中某个文件描述符发生变化为止；  
第二，若将时间值设为0秒0毫秒，就变成一个纯粹的非阻塞函数，不管文件描述符是否有变化，都立刻返回继续执行，文件无变化返回0，有变化返回一个正值；  
第三，timeout的值大于0，这就是等待的超时时间，即select在timeout时间内阻塞，超时时间之内有事件到来就返回了，否则在超时后不管怎样一定返回，返回值同上述。

返回值：

负值-1：select错误   
正值：某些文件可读写的标识符个数  
0：等待超时，没有可读写或错误的文件 。如果在一个描述符上碰到了文件尾端，则select会认为该描述符是可读的。然后调用read，它返回0，这是Unix系统指示到达文件尾端的方法。或者连接正常关闭。

记得当时写的时候非常奇怪，好像每一次在循环的末尾都要重置fd\_set与timeval，否则会被清空。

11．Poll

poll库是在linux2.1.23中引入的，windows平台不支持poll。 poll与select的基本方式相同，都是先创建一个关注事件的描述符的集合，然后再去等待这些事件发生，然后再轮询描述符集合，检查有没有事件发生，如果有，就进行处理。因此，poll有着与select相似的处理流程：

（1）创建描述符集合，设置关注的事件

（2）调用poll()，等待事件发生。下面是poll的原型：

        int poll(struct pollfd \**fds*, nfds\_t *nfds*, int *timeout*);

        类似select，poll也可以设置等待时间，效果与select一样。

（3）轮询描述符集合，检查事件，处理事件。

  在这里要说明的是，poll与select的主要区别在与，select需要为读、写、异常事件分别创建一个描述符集合，最后轮询的时候，需要分别轮询这三个集合。而poll只需要一个集合，在每个描述符对应的结构上分别设置读、写、异常事件，最后轮询的时候，可以同时检查三种事件。poll本质上和select没有区别，但是它没有最大连接数的限制（其实是有上限的，上限与内存大小有关，比如说1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接），原因是它是基于链表来存储的。因为每次调用时都会对连接进行线性遍历，所以两者都会随着FD（句柄）的增加会造成遍历速度慢的“线性下降性能问题”。说是说低效，但是这些并不是不可以拯救。在消息传递上面：内核需要将消息传递到用户空间，都需要内核拷贝动作。

12.epoll

因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。epoll还是poll的一种优化，返回后不需要对所有的fd进行遍历，在内核中维持了fd的列表。select和poll是将这个列表维持在用户态，然后传递到内核中。  
在消息传递上面：epoll通过内核和用户空间共享一块内存来实现的，只会出现一次拷贝。

如果这次没有把数据全部读写完(如读写缓冲区太小)，只通知一次就是边缘触发，反复通知就是水平触发。如果系统中有大量你不需要读写的就绪文件描述符，而它们每次都会返回，这样水平触发会大大降低处理程序检索自己关心的就绪文件描述符的效率！！！

唯一有点麻烦是epoll有2种工作方式：LT和ET。两种触发方式。

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](http://baike.baidu.com/view/1366.htm" \t "_blank)告诉你一个[文件描述符](http://baike.baidu.com/view/1303430.htm" \t "_blank)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non-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](http://baike.baidu.com/view/1366.htm" \t "_blank)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相关的系统调用有：epoll\_create, epoll\_ctl和epoll\_wait，Linux-2.6.19又引入了可以屏蔽指定信号的epoll\_wait: epoll\_pwait。其中epoll\_create用来创建一个epoll[文件描述符](http://baike.baidu.com/view/1303430.htm" \t "_blank)，epoll\_ctl用来添加/修改/删除需要侦听的文件描述符及其事件，epoll\_wait/epoll\_pwait接收发生在被侦听的描述符上的，用户感兴趣的IO事件。epoll文件描述符用完后，直接用close关闭即可，非常方便。事实上，任何被侦听的文件符只要其被关闭，那么它也会自动从被侦听的文件描述符集合中删除，很是智能。

每次添加/修改/删除被侦听文件描述符都需要调用epoll\_ctl，所以要尽量少地调用epoll\_ctl，防止其所引来的开销抵消其带来的好处。有的时候，应用中可能存在大量的[短连接](http://baike.baidu.com/view/3104205.htm" \t "_blank)（比如说Web服务器），epoll\_ctl将被频繁地调用，可能成为这个系统的瓶颈。创建代表连接的socket是到一定的，所以暂时我知道的就是维护一个文件描述符池。

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，我们要给出最大监听的fd+1的值，当创建好epoll句柄后，它就是会占用一个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。

 int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);  
epoll的事件注册函数，它不同与select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。第一个参数是epoll\_create()的返回值，第二个参数表示动作，用三个宏来表示：  
EPOLL\_CTL\_ADD：注册新的fd到epfd中；  
EPOLL\_CTL\_MOD：修改已经注册的fd的监听事件；  
EPOLL\_CTL\_DEL：从epfd中删除一个fd；  
第三个参数是需要监听的fd，第四个参数是告诉内核需要监听什么事，struct epoll\_event结构如下：  
  
typedef union epoll\_data {  
    void \*ptr;  
    int fd;  
    \_\_uint32\_t u32;  
    \_\_uint64\_t u64;  
} epoll\_data\_t;  
  
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，需要再次把这个socket加入到EPOLL队列里

int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);和innotify的响应机制一样，中间两个就是发生的时间集合缓冲区。如果你最后的timeout是0，但是就是检查一下是否已经有消息到来，之后并不会阻塞线程。

1. #include <unistd.h>
2. #include <iostream>
3. #include <sys/epoll.h>
4. using namespace std;
5. int main(void)
6. {
7. int epfd,nfds;
8. struct epoll\_event ev,events[5];//ev用于注册事件，数组用于返回要处理的事件
9. epfd=epoll\_create(1);//只需要监听一个描述符——标准输入
10. ev.data.fd=STDIN\_FILENO;
11. ev.events=EPOLLIN|EPOLLET;//监听读状态同时设置ET模式
12. epoll\_ctl(epfd,EPOLL\_CTL\_ADD,STDIN\_FILENO,&ev);//注册epoll事件
13. for(;;)
14. {
15. nfds=epoll\_wait(epfd,events,5,-1);
16. for(int i=0;i<nfds;i++)
17. {
18. if(events[i].data.fd==STDIN\_FILENO)
19. cout<<"hello world!"<<endl;
20. }
21. }
22. }

STDIN\_FILENO估计就是常量0

1. int main(void)
2. {
3. int epfd,nfds;
4. struct epoll\_event ev,events[5];//ev用于注册事件，数组用于返回要处理的事件
5. epfd=epoll\_create(1);//只需要监听一个描述符——标准输入
6. ev.data.fd=STDIN\_FILENO;
7. ev.events=EPOLLIN|EPOLLET;//使用默认LT模式
8. epoll\_ctl(epfd,EPOLL\_CTL\_ADD,STDIN\_FILENO,&ev);//注册epoll事件
9. for(;;)
10. {
11. nfds=epoll\_wait(epfd,events,5,-1);
12. for(int i=0;i<nfds;i++)
13. {
14. if(events[i].data.fd==STDIN\_FILENO)
15. {
16. cout<<"hello world!"<<endl;
17. ev.data.fd=STDIN\_FILENO;
18. ev.events=EPOLLIN|EPOLLET;//使用默认LT模式
19. epoll\_ctl(epfd,EPOLL\_CTL\_MOD,STDIN\_FILENO,&ev);//重新MOD事件（ADD无效）
20. }
21. }
22. }
23. }

不过由于重新mod的存在导致最后程序的输出就像是LT模式，会不停地重新触发。EPOLL\_CTL\_MOD修改指定fd的监听事件。

使用Linux epoll模型，**水平（LT）触发模式**，当socket可写时，会不停的触发socket可写的事件，如何处理？

第一种最普遍的方式：需要向socket写数据的时候才把socket加入epoll，等待可写事件。接受到可写事件后，调用write或者send发送数据。当所有数据都写完后，把socket移出epoll。这种方式的缺点是，即使发送很少的数据，也要把socket加入epoll，写完后在移出epoll，有一定操作代价。

一种改进的方式：开始不把socket加入epoll，需要向socket写数据的时候，直接调用write或者send发送数据。如果返回EAGAIN（表明你在非阻塞模式下调用了阻塞操作，如果是发送时返回就应该是发送缓存被占满也就是发送的数据太多了），把socket加入epoll，在epoll的驱动下写数据，全部数据发送完毕后，再移出epoll。这种方式的优点是：数据不多的时候可以避免epoll的事件处理，提高效率。

13.守护进程：

示例代码：

**#include<stdio.h>**

**#include<stdlib.h>**

**//#include<sys/types.h>**

**#include<unistd.h>**

**#include<sys/wait.h>**

**#include<sys/stat.h>**

**#define MAXFILE 65535**

**void sigterm\_handler(int arg);**

**volatile sig\_atomic\_t \_running = 1;**

**int main()**

**{**

**pid\_t pc,pid;**

**int i,fd,len;**

**char \*buf="this is a Dameon\n";**

**len = strlen(buf);**

**pc = fork(); //第一步，main父进程创建一个子线程**

**if(pc<0)//fork函数有点神奇，有两个返回值，父进程一个>0子进程一个=0。<0表示出错**

**{**

**printf("error fork\n");**

**exit(1);**

**}**

**else if(pc>0)**

**exit(0);**

**pid = setsid(); //第二步，成为会话组组长，脱离原有控制终端控制**

**if (pid < 0)**

**perror("setsid error");**

**chdir("/"); //第三步//chdir 是C语言中的一个系统调用函数（同cd）。切换目录为根目录等。**

**umask(0); //第四步，取消一切继承过来的文件权限的屏蔽**

**for(i=0;i<MAXFILE;i++) //第五步，关闭所有继承过来的套接字**

**close(i);**

**signal(SIGTERM, sigterm\_handler);**

**while( \_running )**

**{**

**if(fd=open("/tmp/daemon.log",O\_CREAT|O\_WRONLY|O\_APPEND,0600))<0)**

**{**

**perror("open");**

**exit(1);**

**}**

**write(fd,buf,len);**

**close(fd);**

**usleep(10\*1000); //10毫秒//usleep功能把进程挂起一段时间， 单位是微秒（千分之一毫秒）； 头文件： unistd.h**

**}**

**}**

**void sigterm\_handler(int arg)**

**{**

**\_running = 0;**

**}**

15.UNIT Winexec(LPCTSTR lpCmdLine, UINT nCmdShow) 和窗口程序差不多。这里第二个参数是SW\_SHOW，SW\_HIDE。SW\_HIDE估计容易被杀毒软件查杀？你可以去开一个隐藏的死循环等。

#include<windows.h>

int main()

{

WinExec("notepad.exe",SW\_SHOW);//运行计事本

return 0;

}

16.

BOOL CreateProcess(

LPCTSTR lpApplicationName,

LPTSTR lpCommandLine,//指定要传给新进程的参数

LPSECURITY\_ATTRIBUTES lpProcessAttributes, //NULL

LPsECURITY\_ATTRIBUTES lpThreadAttributes, //NULL

BOOL bInheritHandles, //是否继承可继承句柄

DWORD dwCreationFlags, //指定新进程的优先级以及其他创建标。一般情况下为0，如果要创建一个调试进程视为被父进程调试，那就使用DEBUG\_PROCESS，而且会被继承，所以DEBUG\_ONLY\_THIS\_PROCESS。如果创建了子线程而希望主线程挂起就CREATE\_SUSPEND，要唤醒的时候就ResumeThread()。

LPVOID lpEnvironment,

LPCTSTR lpCurrentDirectory, //指定新进程使用的当前目录

LPSTARTINFO lpStartupInfo,

LPPROCESS\_INFOMATION lpProcessInfomation

)

typedef struct \_STRARTUPINFO {

DOWRD cb, //这个成员要被赋值

还有对窗口的基本操作

输入输出的重定向

}STAREIPINFO, \*LPSTARTUPINFO;

typedef struct \_PROCESS\_INFOMATION {

HANDLE hProcess,

HANDLE hThread,

DWORD hProcessID,

DWORD hThreadID

}PROCESS\_INFORMATION，\* LPPROCESS\_INFOMATION;

PROCESS\_INFOMATION pi ={0};

STARTUPINFO si ={0}; si.cb =sizeof(STARTUPINFO);//这种很蠢的设计

BOOL bRet =CreateProcess(xxxx, NULL,

NULL, NULL, FALSE, 0,

NULL, NULL,

&si, &pi);

17.

UINT WinExec(

LPCSTR lpPath,

UINT uCmdShow //SW\_SHOW,SW\_HIDE

)

18.关闭一个进程：

由名字获取窗口句柄HWND-》得到进程id-》得到进程句柄HANDLE-》结束进程=》判断是否成功并关闭HANDLE

#include <windows.h>

int main() {

HWND hNoteWnd =FindWindow(NULL, "无标题-记事本");

if( NULL == hNoteWnd)

return -1;

DOWRD dwNotePid =0; //进程id应该是unsigned int类型

GetWindowThreadProcessId(hNoteWnd, &dwNotePid);

if( dwNotePid )

return -1;

HANDLE hNoteHandle =OpenProcess(PRCESS\_ALL\_ACCESS, FALSE, dwNotePid);

//第二个参数指定获取的句柄是否可以继承。这函数参数顺序sb！

if( NULL==hNoteHandle )

return -1;

BOOL bRet =TerminateProcess(hNoteHandle, 0);//第二个参数是退出码

//也可以调用系统命令kill来执行

if( 1==bRet )

MessageBox(NULL, "进程结束成功", NULL ,MB\_OK );

CloseHandle(hNoteHandle);

return 0;

//也可以用SendMessage发一个WM\_CLOSE给目标窗口

}

19.下面是对应用层进程的枚举，对于可以隐藏的进程常规方法是无效的。

Tlhelp32.h

HANDLE WINAPI CreateToolhelp32Snapshot( //该函数返回一个快照的句柄

DWORD deFlags,

DWORD th32ProcessID

)//第二个参数，如果是想要枚举进程或者线程那么NULL，如果是DLL，那么进程ID(ID为0则退出)。

BOOL WINAPI Process32First(

HANDLE hScapShot,

LPPROCESSENTRY32 lppe

)

BOOL WINAPI Process32Next(

HANDLE hScapShot,

LPPROCESSENTRY32 lppe

)

typedef struct tagPROCESSENTRY32{

DWORD dwSize;//为毛要让我们对结构体进行赋值，C语言没有自带初始化

...

DOWRD th32ProcessID;

.....

DOWRD th32ParentProcessID;

...

TCHAR szExeFile[MAX\_PATH]; //文件路径

} PROCESSENTRY32,\*PPROCESSENTRY32;

Thread32First()、Thread32Next()、Module32First()、Module32Next()都相似

结构体有些差别：

THREADENTRY32里面有

DWORD dwSize

DWORD th32ThreadID

DWORD th32OwnerProcessID

MODULEENTRY32里面有：

DWORD dwSize

TCHAR szModule[MAX\_MODULE\_NAME32+1]

TCHAR szExePath[MAX\_PATH]

VOID Cxxxx::ShowModule() {

//清空列表原有内容

m\_ListModule.DeleteAllItems();

//获取鼠标选中的进程号，这个函数自己写的吧

int nPid =GetSelectPid();

if( 0==nPid )

return ;

MODULEENTRY32 Me32 ={0};

Me32.dwSize =sizeof(MODULEENTRY32);

HANDLE hScap =CreateToolhelp32Snapshot(TH32CS\_SNAPMODULE, nPid);

if( INVALID\_HANDLE\_VALUE==hScap )

return ;

BOOL bRet=Module32First(hScap, &Me32);

int i=0;

CString str;

while( bRet ){

m\_ListModule.InsertItem(i, Me32.szModule);

m\_ListModule.InsertItem(i++, 1, Me32.szExePath);

bRet=Module32Next(hScao, &Me32);

}

CloseHandle(hScap);

}

不过获取进程与线程这些东西可能需要权限，如果你是在IDE当中进行运行程序，就是SeDebugPrivilege权限。我们自己运行a.exe可能就会因为进程权限不够而导致CreateToolhelp32Scapshot()调用失败。类似的，OpenProcess()打开smss.exe等系统进程调用失败也是因为这样。

OpenProcess 函数用来打开一个已存在的进程对象，并返回进程的句柄。

smss.exe(Session Manager Subsystem)任务/会话管理器

将进程改为调试权限有三个步骤：

HANDLE hToken =NULL;;

//以允许我们做任何事情的方式打开当前进程的访问令牌

BOOL bRet =OpenProcessToken(GetCurrnetProcess(),

TOKEN\_ALL\_ACCESS, &hTOken);

//如果打开成功就取得描述权限的Luid，并且更改权限

if( TRUE==bRet ) {

TOEKN\_PRIVILEGES tp; //类型名字里面有S，因为令牌管理很多权限么

tp.PrivilegeCount =1;//这里的privilege应该命名为复数形式

LookupPrivilegeValue(NULL, SE\_DEBUG\_NAME, &tp.Privileges[0].Luid)

/\*函数查看系统权限的特权值，返回信息到一个LUID结构体里。第一个参数表示所要查看的系统，本地系统直接用NULL。第二个参数指向一个以零结尾的字符串，指定特权的名称，如在WinNT h头文件定义。例如，此参数可指定常数，se\_security\_name，或其对应的字符串。第三个参数用来接收所返回的指定特权名称的信息（会不会是指定一个名称，放到一个组里面）。\*/

tp.Privileges[0].Attributes = SE\_PRIVILIGE\_ENABLED;//用来控制特权的开关

AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL ,NULl);// 这个函数启用或禁止 指定访问令牌的特权。所以第二个参数是false。

}

1. 文件、进程、线程、进程、socket的令牌的句柄类型都是普通的HANDLE。
2. 病毒有两个程序，他们不断相互检测，当一个病毒进程发现另外一个结束了就会把被结束的重新运行起来。所以要关掉的话，就要用足够快的速度将两个进程都结束掉，或者，关掉一个后就把他变成不可执行，如果允许挂起的话，那么这就是一个可以用来关闭病毒的漏洞。

23. 线程的挂起与唤醒：

HANDLE OpenThread(//HANDLE返回函数返回错误的话返INVALID\_HANDLE\_VALUE

DWORD dwDesiredAccess, //Access Right

BOOL bInheritHandle, //handle inheritance option

DWORD dwThreadId //又是id是最后一个参数

)

DWORD SuspendThread(HANDLE hThread)

DWORD ResumeThread(HANDLE hThread)

OpenThread这个函数在VC6提供的PSDK中是不存在的，要么使用更高版本的PSDK，要么就从库中动态导入。

通过THRADENTRY32结构体中th32ThreadID来表示当前线程的ID，th32OwnerProcessID表示所属进程的ID。再借助枚举已经可以对指定进程的线程进行操作？不过突然想到一个问题：你怎么判断这个线程是不是你想要的线程。

24.通过修改注册表可以让自己写的任务管理器代替当前任务管理器。C-Sh-Esc也会打开自己写的管理器。

25.线程创建：

HANDLE CreateThread(

LPCECURITY\_ATTRIBUTES lpThreadAttributes, //NULL 新线程的安全性描述符

DWORD dwStackSize, //0 原始堆栈大小

LPTHREAD\_START\_ROUTINE lpStartAddress, //指定线程运行的函数，属于回调哈数，线程建立时执行。

LPVOID lpParameter, //传参指针

DWORD dwCreationFlags,//0立刻执行，CREATE\_SUSPENDED立刻挂起

LPDWORD lpTreadId //返回新建线程的ID

)

DWORD WINAPI ThreadProc(LPVOID lpParameter)参数由CreateThread给定，函数名称可以任意。

26.主线程一旦退出了，其他的线程全部GG。所以：

DOWRD WaitForSingleObject(

HANDLE hHandle, //要等待的对象句柄

DOWRD dwMilliseconds //等待超时时间，0立即返回，INFINITE

)

#define INFINITE 0xFFFFFFFF

VOID ExitThread (DWORD dwExitCode);来结束线程

在主线程里面WaitForSingleObject(新线程Handle, INFINITE);那么就会等待新建线程结束，再继续执行主线程。如果是要等待多个线程的结束，那么就使用线程数组，然后WaitForMultipleObjects(数目, 头地址, TRUE, INFINITE);TRUE就是等待所有信号量有效才往下执行，否则有一个有效就执行。

waitForMultipleObjects

功能：WaiForMultipleObjects函数当下列条件之一满足时返回：

(1)任意一个或全部指定对象处于信号态；

(2)超时间隔已过。

格式：DWORD waitForMultipleObjects (DWORD nCount，

CONST HANDLE \* lpHandles,

BOOL fWaitAll，

IDWORD dwMilliSeconds) ;

参数说明：

nCount——句柄的数量 最大值为MAXIMUM\_WAIT\_OBJECTS（64）

HANDLE——句柄数组的指针。

HANDLE 类型可以为（Event，Mutex，Process，Thread，Semaphore ）

数组 。没想到信号量、互斥量、Event都是HANDLE类型。

bWaitAll ——为TRUE时，等待所有；为FALSE，一个有效即可

dwMilliseconds——超时时间 超时后向执行。注意，这些底层的都精准毫秒

如果为WSA\_INFINITE 永不超时。如果没有信号量就会在这死等。

不要等待已经结束了的线程，这会导致程序一直阻塞

27.Sleep函数会主动让出CPU时间片。usleep函数

在 <unistd.h>头文件或者<windows.h>中：

Sleep()单位为毫秒，sleep()单位为秒（如果需要更精确可以用usleep单位为微秒）

我在VS下就会#define sleep(time) Sleep(time\*1000)

在linux中包含头文件unistd.h就可以直接使用了

28.linux下的线程创建pthread.h（编译的时候需要链接-lpthread）： fork是子进程创建函数

int pthread\_create(pthread\_t \*tidp,

　　 pthread\_attr \*attr, 可以是NULL

　　 void \*(\*start\_trn)(void \*),

　　 void \*arg);

参数说明：

tidp——指向线程标识符的指针。

attr——设置线程属性。

start\_trn——线程运行函数的起始地址。

arg——运行函数的参数。

返回值：若线程创建成功，则返回0。若线程创建失败，

则返回出错编号。

int pthread\_join(pthread\_t thread, void \*\*retval);

参数说明：

thread——线程标识符，即线程ID。

retval——用户定义的指针，用来存储被等待线程的返回值。

返回值：0代表成功。 失败，返回的则是错误号。

说明：pthread\_join()函数，以阻塞的方式等待thread指定的

线程结束。当函数返回时，被等待线程的资源被收回。

如果进程已经结束，那么该函数会立即返回

pthread\_cancel(pthread\_t tid); 取消同一个进程中的线程

void pthread\_cleanup\_push(void (\*rtn)(void \*), void \*arg);

void pthread\_clean\_pop(int execute);

线程可以安排它退出时需要调用的函数，这样的函数称为线程清理处理程序，线程可以建立多个清理处理程序。处理程序记录在栈中，也就是说它们的执行顺序与它们注册时的顺序相反。类似，进程也会有退出回调函数。

include <sys/wait.h>

pid\_t wait(int \*status);

成功时，返回已退出子进程的进程ID；失败时，则返回-1并设置errno。

注意父子进程是两个进程，子进程退出和父进程调用wait（）函数来获取子进程的退出状态在时间上是独立的事件，因此会出现以下两种情况：

·子进程先退出，父进程后调用wait（）函数。

·父进程先调用wait（）函数，子进程后退出**。**调用时并无子进程退出，该函数调用就会陷入阻塞状态，直到某个子进程退出。**如果不存在子进程退出，wait（）只能阻塞，wait（）函数只能发现子进程的终止事件，其他信号都是不知道的。传统的wait函数只关注子进程的终止，而waitpid函数则可以通过前两个标志位来检测子进程的停止和从停止中恢复这两个事件。**

29.#define APIENTRY WINAPI

30.跨进程创建线程就是“远程线程”，由此可以对代码进行注入：

DLL远程注入：

DLL文件不会单独创建一个进程，而是被加载到进程的地址空间中，因此隐蔽性相对较好。利用远程线程创建函数可以强制注入dll。

HANDLE CreateRemoteThread(

HANDLE hProcess,

LPSECURITY\_ATTRIBUTES lpThreadAttributes,

DWORD dwStatckSize,

LPTHREAD\_STATE\_ROUTINE lpStartAddress,

LPVOID lpParameter,

DWORD dwCreationFlags,

LPDWORD lpThreadId

)

书上说CreateThread由CreateRemoteThread实现。第一个参数是NtGetCruuentProcess()来获取当前进程句柄。

32.从进程名字获取进程ID，目前知道的就是要走快照枚举然后看PROCESSENTRY32中的th32ProcessId。但是窗口名都可以使用windows API查到进程id，进程名估计也是有的。

33.OpenProcess这个函数的使用需要进程有足够的权限。

34.使用Volumn Shadow Copy创建无文件进程：

前提：需要取得Administrator权限

使用：可执行文件执行以后，文件被锁定而不能删除，但是这样增加被发现的可能。一般的

做法事做好隐藏，不过也可以删除。内存运行着的进程其实不需要这个文件了。

35.创建进程：

需要使用windows.h，已证其被bits/stdc++.h包括

bool CreateProcess(LPCTSTR lpApplicationName, LPTSTR lpCommandLine, LPSECURITY\_ATTRIBUTES lpProcessAttributes, LPSECURITY\_ATTRIBUTES lpThreadAttributes, bool blnheritHandles, DWOROD dwCreationFlags, LPVOID lpEnvironment, LPCTSTR lpCurrentDirectory, LPSTARTUPINFO lpStartupinfo, LPPROCESS\_INFOMATION lpProcessInfomation)

使用代码：

STARTUPINFO si;

memset(&si, 0, sizeof(STARTUPINFO) );

si.cb =sizeof(STARTUPINFO);

si.dwFlags =STARTF\_USESHOWWINDOWS;

si.wShowWindow =SW\_SHOW;

PROCESS\_INFOMATION pi;

if( !CreateProcess(Null, ”要运行的进程”, NULL, NULL, false, 0, NULL, NULL, &si, &pi) ) {

cout << “Create Fail!” << endl;

exit(1);

}else {

Xxxxxxx;

}

36. vfork()用法与fork()相似

#include<unistd.h>

1. fork():子进程拷贝父进程的数据段，代码段. vfork():子进程与父进程共享数据段.直到exec或exit。当进程调用一种exec函数时，该进程完全由新程序代换，

2. fork():父子进程的执行次序不确定. vfork()保证子进程先运行，在她调用exec或exit之后父进程才可能被调度运行。如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。

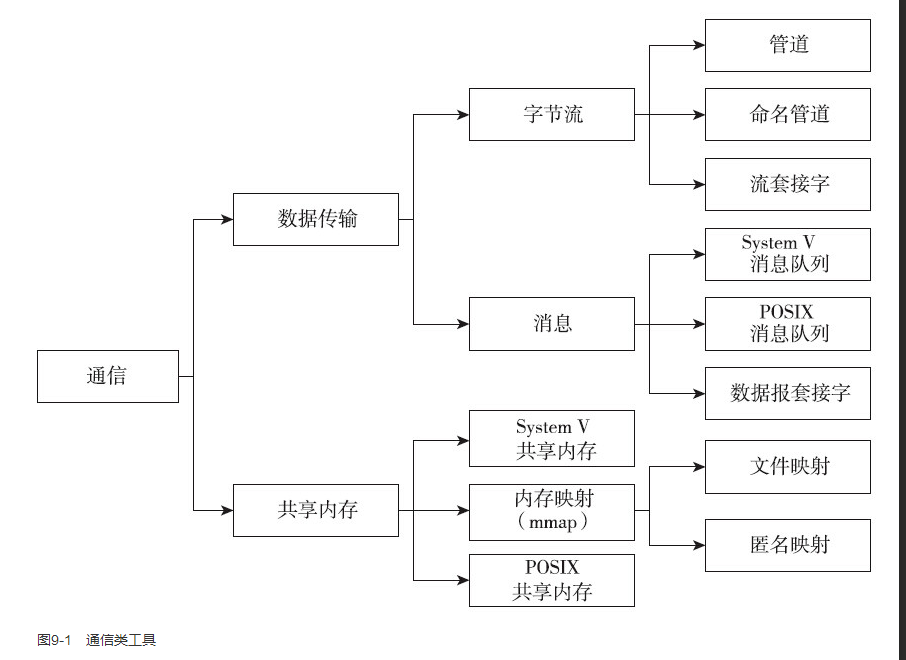
如果vfork失败则直接返回-1，失败原因存于[errno](http://baike.baidu.com/subview/3485007/3485007.htm" \t "_blank)中

#include <sys/prctl.h>

fork与vfork出来的子进程名字是不知道的，所以不好ps –aux查询，可以使用prctl(PR\_SET\_NAME, “process\_name”, NULL, NULL, NULL);  来更改进程名字，但是ps还是查不到。后来用一个死循环来占用一些CPU资源，并使用ps –aux | tail来查看可以发现ps里面名字都是./a.out。

代码空间是共享的。

37.fork当需要改变共享数据段中变量的值，则拷贝父进程写时复制（或COW）



38.linux 下的pipe

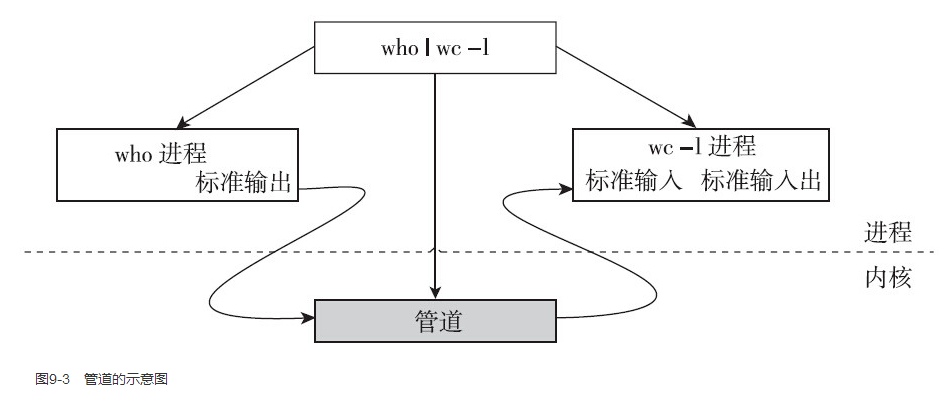
管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道。管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；

#include <unistd.h>

int pipe(int fd[2])这个函数会在进程内创建一个函数，然后通过fork()或者vfork()产生一个新的进程。由于进程的资源继承，之后这个文件描述符会被有共同祖先（有亲缘关系的进程共享）。如果成功，则返回值是0，如果失败，则返回值是-1，并且设置errno。 成功调用pipe函数之后，会返回两个打开的文件描述符，一个是管道的读取端描述符pipefd[0]，另一个是管道的写入端描述符pipefd[1]。管道没有文件名与之关联，因此程序没有选择，只能通过文件描述符来访问管道，只有那些能看到这两个文件描述符的进程才能够使用管道。

管道又不是一种普通的文件，它属于一种独特的文件系统：pipefs。管道的本质是内核维护了一块缓冲区与管道文件相关联，对管道文件的操作，被内核转换成对这块缓冲区内存的操作。

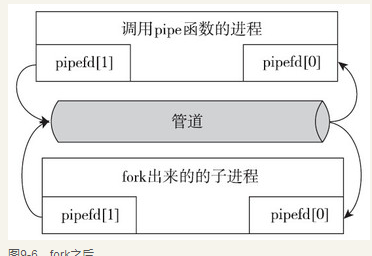
管道是第一个广泛应用的进程间通信手段。日常在终端执行shell命令时，会大量用到管道。但管道的缺陷在于只能在有亲缘关系（有共同的祖先）的进程之间使用。为了突破这个限制，后来引入了命名管道。



一旦向管道的写入端写入数据后，就可以对读取端描述符pipefd[0]调用read，读出管道里面的内容。如下所示，管道上的read调用返回的字节数等于请求字节数和管道中当前存在的字节数的最小值。如果当前管道为空，那么read调用会阻塞（如果没有设置O\_NONBLOCK标志位的话）。

调用pipe函数返回的两个文件描述符中，读取端pipefd[0]支持的文件操作定义在read\_pipefifo\_fops，写入端pipefd[1]支持的文件操作定义在write\_pipefifo\_fops对读取端描述符执行write操作，内核就会执行bad\_pipe\_w函数；对写入端描述符执行read操作，内核就会执行bad\_pipe\_r函数。这两个函数比较简单，都是直接返回-EBADF。因此对应的read和write调用都会失败，返回-1，并置errno为EBADF。

Fork()之后由于继承就会变成这样子的情况，记住管道本质上是内核管理的缓冲区。为了避免数据的混杂，这两条路都是可选的但是不能同时选择。所以需要子进程关掉一个文件描述符，父进程对应也关掉一个文件描述符。父进程再次创建一个子进程B，子进程B就持有管道写入端，这时候两个子进程之间就可以通过管道通信了。父进程为了不干扰两个子进程通信，很自觉地关闭了自己的写入端。从此管道成为了两个子进程之间的单向的通信通道。在shell中执行管道命令就是这种情景，只是略有特殊之处，其特殊的地方是管道描述符占用了标准输入和标准输出两个文件描述符 （别问我父进程是谁）（可以使用dup2(pipefd[0], STDOUT)做到绑定）



1、只有当所有的写入端描述符都已经关闭了，而且管道中的数据都被读出，对读取描述符调用read函数才返回0（读到EOF标志）。也就是成功读取0字节数据。

2、如果所有的读取端描述符都已经关闭了，此时进程再次往管道里面写入数据，写操作将会失败，并且内核会像进程发送一个SIGPIPE信号(默认杀死进程)。

3、当所有的读端与写端都已经关闭时，管道才会关闭.。好像C+智能指针

就因为有这些特性，我们要即使关闭没用的管道文件描述符

在调用pipe时，进程很可能已经关闭了标准输入和标准输出。

调用pipe函数时，内核会分配最小的文件描述符，所以pipe的文件描述符可能等于0或1。分别是标准输出和标准输入。2是标准错误

每一个进程都会有一个自己三个标准文件：输入、输出、错误。

Exec族函数：

int execl(const char \*path,const char \*arg,...);最后一个参数要是(char \*)0，则就会warning: missing sentinel in function call [-Wformat=]。

    int execlp(const char \*file,const char \*arg,...);

    int execle(const char \*path,const chr \*arg,...,char \* const envp[]);

    int execv(const char \*path,char \* const argv[]);

    int execvp(const char \*file,char \* const argv[]);

path的话就要写/bin/ls这样，file倒是用ls其实会根据PATH里面的设置自动寻找路径。所以不管那个第二个参数都要再来次命令。比如”ls”, ”ls”, “-l”, “a.txt”,(char\*)0

execl("/bin/sh", "sh", "-c", cmdstring, (**char** \*) 0);   调用一个新的sh来执行

battery@ubuntu:~/Desktop$ /bin/sh

$ ^C

$ exit

battery@ubuntu:~/Desktop$ sudo /bin/sh

[sudo] password for battery:

# exit

 应该说, /bin/sh 与 /bin/bash 虽然大体上没什么区别, 但仍存在不同的标准. 标记为 “#!/bin/sh” 的脚本不应使用任何 POSIX 没有规定的特性。

battery@ubuntu:~/Desktop$ /bin/bash

battery@ubuntu:~/Desktop$ exit

exit

/bin/bash –c ls 这条命令执行完以后会自动退出开启的bash

如果要执行一个自己写的脚本:

execl("/bin/sh", "sh", "-c" ,"/home/battery/Desktop/raspivid.sh", (char\*)0); 第四个参数前面没有点。

execle("t.sh", (char\*)0, env);使用execle函数，也可以向shell脚本传递参数，在脚本中使用$0,$1$2等等来进行引用，$1为第一个参数，这个与main函数的argv类似。

这里execle函数发现pathname参数并不是由连接编辑器产生的机器可执行文件，则认为该文件是一个shell脚本，于是试图调用/bin/sh，并以filename作shell的输入。

我们通过fork、exec及wait家族函数一样也可以实现system的功能。但终归是不方便，system函数为我们提供了一些便利（可是没有返回输出，不过会等待程序执行结束因为还使用了waitpid这个函数）。

battery@ubuntu:~/Desktop$ echo 123; echo 123

123

123

battery@ubuntu:~/Desktop$ echo 123;echo 123

123

123

考虑到system会开启一个shell进程，所以如果同时用到多次命令，那么用;隔开在一次执行完。也可以将命令写到一个shell脚本里面执行。

system返回值：指调用system函数后的返回值，比如上例中status为system返回值

调用命令返回值：指system所调用的shell命令的返回值，比如上例中，test.sh中返回的值为shell返回值。echo $?记录的只是用一个终端。

阶段1：创建子进程等准备工作。如果失败，返回-1。

阶段2：调用/bin/sh拉起shell脚本，如果拉起失败或者shell未正常执行结束（参见备注1），原因值被写入到status的低8~15比特位中。system的man中只说明了会写了127这个值，但实测发现还会写126等值。

阶段3：如果shell脚本正常执行结束，将shell返回值填到status的低8~15比特位中。

system(“cat test”)虽然会重开一个shell，但是cat执行的显示会显示在运行程序的shell里面(因为继承导致资源还是一样的)，popen(“cat -”, ”w”)因为是”w”所以也是这样，执行cat但是显示在父进程的父进程bash中。

使用system的话比exec好在可以直接在命令中带着管道，不过记住标准流被重定向了，而且会等待子线程的执行完毕。不想要阻塞只能自己实现或者使用popen。

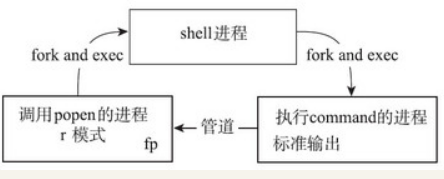
与system同样的道理，只用pipe函数及dup2等函数，也能完成popen要完成的工作，但popen接口给我们提供了便利。   
popen接口定义如下：

#include <stdio.h>

FILE \*popen(const char \*command, const char \*type);

int pclose(FILE \*stream); 会杀死创建的进程

注意，gccc –std=c99以后会说popen和pclose为声明，因为这不是c标准里面的函数，虽然它们的定义是在stdio.h，我们需要自己给它加上声明。

popen函数会创建一个管道，并且创建一个子进程来执行shell，shell会创建一个子进程来执行command。根据type值的不同，分成以下两种情况。   
如果type是r：command执行的标准输出，就会写入管道，从而被调用popen的进程读到。R是对我们创建的对象而言。通过对popen返回的FILE类型指针执行read或fgets等操作，就可以读取到command的标准输出。

如果type是w：调用popen的进程，可以通过对FILE类型的指针fp执行write、fputs等操作，负责往管道里面写入，写入的内容经过管道传给执行command的进程，作为命令的输入

popen函数成功时，会返回stdio库封装的FILE类型的指针，失败时会返回NULL，并且设置errno。常见的失败有fork失败，pipe失败，或者分配内存失败。

popen函数和system有很多相似的地方，但是也有显著的不同。调用system函数时，shell命令的执行被封装在了函数内部，所以若system函数不返回，调用system的进程就不再继续执行。但是popen函数不同，一旦调用popen函数，调用进程和执行command的进程便处于并行状态。然后pclose函数才会关闭管道，等待执行command的进程退出。

在并行期间，调用popen的进程可能会创建其他子进程，所以标准规定popen不能阻塞SIGCHLD信号.这也意味着，popen创建的子进程可能被提前执行的等待操作所捕获。若发生这种情况，调用pclose函数时，已经无法等待command子进程的退出，这种情况下，将返回-1，并且errno为ECHILD。

Popen函数是由fork+pipe+exec族+dup2()实现的，dup2()用于将表标准输入输出重定向到目标文件描述符。

SIGCHLD，在一个进程终止或者停止时，将SIGCHLD信号发送给其父进程，按系统默认将忽略此信号，如果父进程希望被告知其子系统的这种状态，则应捕捉此信号。

 ctrl-c（SIGINT）

可以使用一个信号处理程序来监控 ctrl-c（SIGINT）并且重置一个标志（keep\_running）使应用程序了解终止操作。使用signal()函数，第一个参数是信号，第二个参数是处理函数。

#include <unistd.h> //在C++里面也是这个名字，因为说这并不是C语言的标准库

//#include <sys/types.h>

#include <errno.h>

Int main(void)

{

int pipe\_fd[2];

pid\_t pid;

char r\_buf[100];

char w\_buf[4];

char\* p\_wbuf;

int r\_num;

int cmd;

memset(r\_buf,0,sizeof(r\_buf));

memset(w\_buf,0,sizeof(r\_buf));

p\_wbuf=w\_buf;

if(pipe(pipe\_fd)<0)

{

printf("pipe create error ");

return -1;

}

if((pid=fork())==0)

{

printf(" ");

close(pipe\_fd[1]);

sleep(3);//确保父进程关闭写端

r\_num=read(pipe\_fd[0],r\_buf,100);

printf( "read num is %d the data read from the pipe is %d ",r\_num,atoi(r\_buf));

close(pipe\_fd[0]);

exit();

}else if(pid>0) {

close(pipe\_fd[0]);//read

strcpy(w\_buf,"111");

if(write(pipe\_fd[1],w\_buf,4)!=-1)

printf("parent write over ");

close(pipe\_fd[1]);//write

printf("parent close fd[1] over ");

sleep(10);

}

}

pipe 写入的原子性，是说多个进程写入管道，如果每次写入的字节数目都小于PIPE\_BUF，那么各个进程写入的数据彼此不会交叠

"原子操作(atomic operation)是不需要synchronized"，这是Java多线程编程的老生常谈了。所谓原子操作是指不会被[线程调度](http://baike.baidu.com/view/1336364.htm" \t "_blank)机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch

向管道中写入数据时，linux将不保证写入的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读走管道缓冲区中的数据，那么写操作将一直阻塞。读写问题应该是自带锁吧。内核必须利用一定的机制同步对管道的访问，为此，内核使用了锁、等待队列和信号。

在 Linux 中，管道的实现并没有使用专门的[数据结构](http://lib.csdn.net/base/datastructure" \t "_blank" \o "算法与数据结构知识库)，而是借助了文件系统的file结构和VFS的索引节点inode。通过将两个 file 结构指向同一个临时的 VFS 索引节点，而这个 VFS 索引节点又指向一个物理页面而实现的。

While( event\_check() && …) {….}

int event\_check (int fd)

{

fd\_set rfds;

FD\_ZERO (&rfds);

FD\_SET (fd, &rfds);

/\* Wait until an event happens or we get interrupted

by a signal that we catch \*/

return select (FD\_SETSIZE, &rfds, NULL, NULL, NULL); //放到循环里面回显的循环可读性差

}

FIFO管道（命名管道）是一种文件类型，在文件系统中可以看到。程序中可以查看文件stat结构中st\_mode成员的值来判断文件是否是FIFO文件。创建一个FIFO文件类似于创建文件，FIFO文件就像普通文件一样。FIFO中可以很好地解决在无关进程间数据交换的要求，并且由于它们是存在于文件系统中的类似于在进程中使用文件来传输数据，只不过FIFO类型文件同时具有管道的特性，这也提供了一种比匿名管道更持久稳定的通信办法。

mkfifo filename就会在当前目录下创建一个具有管道特性的文件，Ubuntu中ls会用橙色标出。

cat /tmp/fifoFfmpeg，如果命名管道里面没有数据就会等待一次传输数据的结束。如果文件中一直在被写入，那么cat会一直读取，是因为写入太快了，cat读取到显示是很慢的。

一个终端执行脚本：

#!/bin/bash

if [ ! -e /tmp/fifoGst ];then

mkfifo /tmp/fifoGst

fi

if [ ! -e /tmp/fifoFfmpeg ];then

mkfifo /tmp/fifoFfmpeg

fi

while [ true ]

do

echo -n 1

sleep 3

done

./this.sh | tee /tmp/fifoGst > /tmp/fifoFfmpeg

还有两个bash分别cat /tmp/fifoGst 和/tmp/fifoFfmpeg。没想到cat会一直执行，直到三个部分中有一个出错断了就整个崩了。如果其中一个cat断掉就会让数据源阻塞自己，然后就会导致另外一个cat断了，再然后大概就是发现读取的人都没了然后数据源进程也断了。

#include <iostream>

#include <string.h>    //strlen()

#include <fcntl.h>    //O\_WRONLY O\_RDONLY

#include <pthread.h>

#include <sys/types.h>

#include <sys/stat.h>    //mkfifo()

#include <errno.h>    //EEXIST

#define FIFOPATH1 "/home/xbq/dev/thread/fifo1"

#define FIFOPATH2 "/home/xbq/dev/thread/fifo2"

#define FILE\_MODE (S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH)

using namespace std;

void\* thread1 (void\*)

{

    int readfd, writefd;

    char\* buff = (char\*)"nihao thread!";

    writefd = open(FIFOPATH1, O\_WRONLY, 0);

    readfd = open(FIFOPATH2, O\_RDONLY, 0);

    int len = strlen(buff);

    write(writefd, buff, len);

    read(readfd, buff, 512);

    cout << buff << "in thread1" << endl;

    close(readfd);

    close(writefd);

}

void\* thread2 (void\*)

{

    int readfd, writefd, n;

    char\* buff;

    buff = new char[15];

    readfd = open(FIFOPATH1, O\_RDONLY, 0);

    writefd = open(FIFOPATH2, O\_WRONLY, 0);

    cout << read(readfd, buff, 13) << endl;

    buff[strlen(buff)] = '\0';

    cout << buff << "in thread2" << endl;

    sprintf(buff, "%s%s", buff, "!!");

    cout << buff << "in thread2" << endl;

    write(writefd, buff, strlen(buff));

    delete buff;

    close(readfd);

    close(writefd);

}

int main (void)

{

    pthread\_t tid1, tid2;

    cout << pthread\_self() << endl;

    cout << mkfifo(FIFOPATH1, FILE\_MODE);

    cout << mkfifo(FIFOPATH2, FILE\_MODE);

    pthread\_create(&tid1, NULL, thread1, NULL);

    pthread\_create(&tid2, NULL, thread2, NULL);

    unlink(FIFOPATH1);

    unlink(FIFOPATH2);

    return 0;

}

但是并不是所有的程序都可以走命名管道来传递数据，test-launch就不能走，如果通过命名管道来传送h264流给它，这样在rtsp客户端就会出现卡顿然后一下子快速显示那段时间里面缺失的视频（估计是因为FIFO的实现，我猜测可能实现了缓冲区）。命名管道带来的是数据传送上的方便，但是效率上还是又损失的，如果要解决这种读效率依赖高的问题又不想让因为while中的popen、fread、fwrite造成CPU使用高（发热与耗能高），就需要自己在编程实现那种让程序自动传送数据的管道。？这问题还有待研究。

39.

#include  
int dup(int oldfd);  
int dup2(int oldfd,int newfd);  
  
       dup用来复制oldfd所指的文件描述符。但复制成功时返回最小的尚未被使用的文件描述符。若有错误则返回－1，错误代码存入errno中。返回的新文件描述符和参数oldfd指向同一个文件，共享所有的锁定，读写指针，和各项权限或标志位。  
       dup2可以用参数newfd指定新文件描述符的数值。若newfd已经被程序使用，系统就会将其关闭以释放该文件描述符；若newfd与oldfd相等，dup2将返回newfd，而不关闭他。dup2调用成功返回新的文件描述符，出错则返回－1。  
  
      标准输入（stdin），标准输出（stdout），标准出错信息（stderr）的文件号分别为0，1，2

40.下面这个程序片段不能正常自己结束：

pid\_t pid =vfork();

if(pid == 0) {

//system("mkfifo fifoGst"); 这里没有检验文件已经存在，mk系列是不会覆盖的。

//system("mkfifo fifoFfmpeg");

system("echo 123 | tee fifoGst > fifoFfmpeg");

return 0;

}

else if(pid > 0) {

execl("/bin/cat","cat", "fifoFfmpeg", (char\*)0);

}else

selfSyslog(LOG\_ERR, "process fork() error");

因为子进程输出到通道但是没人来读，死锁了。由于是vfork()那么父进程只有当子进程结束以后才会执行，需要改为fork。而且父进程里面execl只能执行一次命令，所以还需要你开一个终端cat fifoGst，才能使得进程结束。

mkfifo $tmp\_fifo //创建命名管道

exec 6<>$tmp\_fifo #这句话能把管道变成非阻塞写！

也可以自己想打开文件一样，非阻塞打开

40.5. 信号量可以实现互斥量，大部分情况下也可以实现条件变量。甚至使用信号量的实现远比其他实现更容易理解。然而很多时候使用信号量替换条件变量的可能会降低并发程序的性能，因为多个条件变量也只需要一个锁或者临界区封装，但是每一个信号量都是一个锁。 不过信号量代码写起来更加简单，最为一般的选择。

41.互斥量Mutex： 只能0/1变换的信号量，但是信号量还可以是负数

在windows下使用mutex可能会有问题，需要注意：

hBuf2Mutex = CreateMutex(NULL, FALSE, NULL);

printf("%d", WaitForSingleObject(hBuf2Mutex, INFINITE));看似会发生上面，但是问题在于不会阻塞，如果不阻塞，那函数成功运行，返回0。CreateMutex中第二个参数是FALSE（我是在VS中编写C语言代码，所以可使用FALSE），那么这个Mutex创建以后就没有被拥有，直到有一个线程使用对其使用了Wait..()系列函数才会拥有，一旦没有任何线程拥有mutex，这个mutex便处于激发状态，因此，如果没有任何线程拥有那个mutex，Wait...()便会成功。然后就会变成未激发的状态，其他线程使用Wait...会被阻塞。我在测试中发现在Windows下，拥有Mutex对象的线程与创建Mutex的线程（也就是调用CreateMutex的线程）都不会被这个Mutex所阻塞，网上面的人似乎称这个现象较“重入”。所以windows下不要用互斥量，还是用Semaphore来代替好了。

HANDLE CreateMutex(LPSECURITY\_ATTRIBUTES lpMutexAttributes,

BOOL bInitialOwner,

LPCTSTR lpName)；

参数说明：

lpMutexAttributes—取值NULL。

bInitialOwner—指示当前线程是否马上拥有该互斥量(即马上加锁)。

lpName—互斥量名称。lpName: The name is limited to MAX\_PATH characters.If lpName matches the name of an existing named mutex object, this function requests the MUTEX\_ALL\_ACCESS access right.

如果函数调用之前，有名互斥对象已存在，那么函数给已存在的对象返回一个句柄。

BOOL WIANPI ReleaseMutex(HANDLE hMutex);

参数说明：

hMutex—互斥对象句柄。

返回值：TRUE表示成功，FALSE表示失败。

Linux Mutex：

pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

参数说明：

mutex——互斥信号量指针。

返回值：在成功完成之后会返回零。其他任何返回值都表示

出现了错误。如果该互斥锁已被另一个线程锁定和拥有，则调用该线程将阻塞，直到该互斥锁变为可用为止

pthread\_mutex\_unlock

功能：给互斥量解锁。

格式：

int pthread\_mutex\_unock(pthread\_mutex\_t \*mutex);

参数说明：

mutex——需要解锁的互斥信号量指针。

返回值：在成功完成之后会返回零。其他任何返回值都表示

出现了错误。

42.信号量Semaphore：

[PV操作](https://baike.baidu.com/item/PV%E6%93%8D%E4%BD%9C" \t "/home/vega/Documents\\x/_blank)与信号量的处理相关，P表示通过的意思，V表示释放的意思。

HANDLE CreateSemaphore(

LPSECURITY\_ATTRIBUTES lpSemaphoreAttributes,

LONG lInitialCount,

LONG lMaximumCount,

LPCTSTR lpName);

If the lpSemaphoreAttributes parameter is not **NULL**, it determines whether the handle can be inherited, but its security-descriptor member is ignored

参数说明：

lpSemaphoreAttributes——用于定义信号量的安全特性。

lInitialCount——设置信号量的初始计数。（没有开始增加之前能有多少个减少，一开始能进来多少个）

lMaximumCount——设置信号量的最大计数。

lpName——指定信号量对象的名称。

BOOL ReleaseSemaphore(

HANDLE hSemaphore,

　　 LONG lReleaseCount,

　　 LPLONG lpPreviousCount);

参数说明：

hSemaphore——所要释放的信号量句柄。

lReleaseCount——所要释放信号量的数目。 信号量就是这样对数字的操控你可以“随意”

lpPreviousCount——指向返回信号量上次值的变量的指针，如果不需要信号量上次的值，那么这个参数可以设置为NULL。就是用来看值么。。。

Linux semaphore: <semaphore.h>

sem\_t binSem; 创建一个信号量变量

struct semaphore {

spinlock\_t lock; /\* 自旋锁结构体变量 \*/

unsigned int count; /\* 用于计录资料数量 \*/

struct list\_head wait\_list; /\* 内部链表结构体变量 \*/

};

从此结构体中可以看从，semaphore机制也就是自旋锁的包装

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

sem\_init函数是Posix信号量操作中的函数。用来初始化信号量。sem\_init() 成功时返回 0；错误时，返回 -1，并把 errno 设置为合适的值。如果 pshared 是非零值，那么信号量将在进程之间共享

int sem\_post(sem\_t \*sem); 使信号量加1，是个原子操作

int sem\_wait(sem\_t \*sem); 使信号量减1，是个原子操作

参数说明：

sem——信号量资源指针。

返回值：在成功完成之后会返回零。其他任何返回值都表示

出现了错误。

int sem\_timedwait(sem\_t \*sem, const struct timespec \*abs\_timeout);阻塞等待若干时间直到信号量>0或者时间结束。第二个参数是时间戳。

int sem\_destroy(sem\_t \* sem);

参数说明：

sem——信号量资源指针。

返回值：在成功完成之后会返回零。其他任何返回值都表示

出现了错误。

43. windows操作系统里面的临界区（准确来说是管程的实现）：

临界区结构CRITICAL\_SECTION

VOID InitializeCriticalSection(LPCRITICAL\_SECTION lpCriticalSection)//初始化临界区

VOID EnterCriticalSection( LPCRITICAL\_SECTION lpCriticalSection);//进入临界区

VOID LeaveCriticalSection( LPCRITICAL\_SECTION lpCriticalSection)//离开临界区

VOID DeleteCriticalSection( LPCRITICAL\_SECTION lpCriticalSection) //临界区肯定也是要占用一些变量的，既然要自己调用函数来创建or初始化那么就要自己来删除。

使用临界区主要是因为高效，进不去的时候自旋，这里就有个dwSpinCount，是自旋次数，当达到这个值得时候，还进不去的话就要进入内核态挂起等待。切换到内核态较之线程context切换成本很高。

44.管程monitor：

P(S)和V(S)操作大量分散在各个进程中，不易管理，易发生死锁，于是有了管程封装了同步操作，对进程隐蔽了同步细节，简化了同步功能的调用界面，用户编写并发程序如同编写顺序(串行)程序。

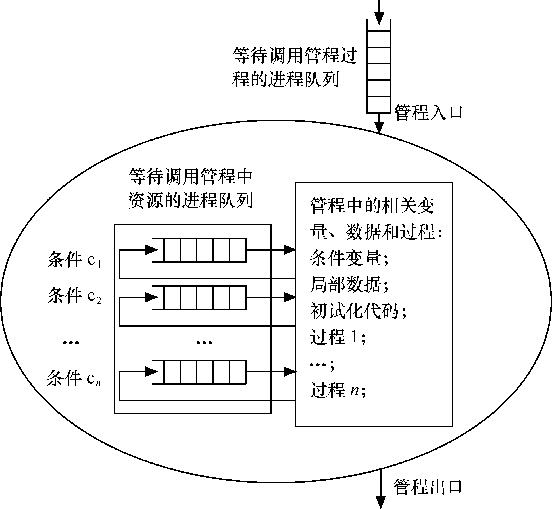
为此Dijkstra于1971年提出：把所有进程对某一种[临界资源](https://baike.baidu.com/item/%E4%B8%B4%E7%95%8C%E8%B5%84%E6%BA%90" \t "/home/vega/Documents\\x/_blank)的同步操作都集中起来，构成一个所谓的秘书进程。凡要访问该临界资源的进程，都需先报告秘书，由秘书来实现诸进程对同一临界资源的互斥使用。管程可以单独作为一个模块编译。

管程内部使用条件变量管理资源（资源等待队列），外部使用信号量管理入口队列。

**管程由4部分组成：就像是一个class封装**

1. **管程的名称**
2. **局部于管程内部的共享数据结构说明**
3. **对该数据结构进行操作的一组过程**
4. **对局部于管程内部的共享数据设置初始的语句**

管程只有一个入口和一个出口，并发进程在管程外等待调用管程中的过程，如果进程在管程中不能访问临界资源，则在管程中阻塞等待访问完临界资源的进程将其唤醒 。



一旦资源不足就会进行等待直到被唤醒，但是因为存在“虚假唤醒”问题（惊群效应），于是需要wait函数外面嵌套一个while，而这个检查很明显不是原子操作，于是管程才会加上临界区或者需要一个mutex来保证操作的原子性。

java的管程只要保证静态类内部操作函数是互斥的就好

Windows Monitor:

condition variables是微软从vista和2008以后引入的技术，xp和2003的系统不支持。

CONDITION\_VARIABLE ConditionVar;

WakeConditionVariable           唤醒一个等待条件变量的线程

WakeAllConditionVariable      唤醒所有等待条件变量的线程；

SleepConditionVariableCS       释放临界区锁和等待条件变量作为原子性操作

SleepConditionVariableSRW   释放SRW锁和等待条件变量作为原子性操作.

使用方法如下：windows管程需要配合临界区使用

Thread1：

   EnterCriticalSection(&CritSection);

   while( TestPredicate() == FALSE ){

      SleepConditionVariableCS(&ConditionVar, &CritSection, INFINITE);

   }

   DoSth();

   LeaveCriticalSection(&CritSection);

Thread2：

   EnterCriticalSection(&CritSection);

      WakeConditionVariable (ConditionVar);

   LeaveCriticalSection(&CritSection);

C++11：编译器很可能不支持这个C++11特性

std::condition\_variable cv;

std::mutex mtx;

std::unique\_lock lck(mtx);

cv.wait(lck) //需要独立锁

cv.notify\_one();//随便唤醒一个

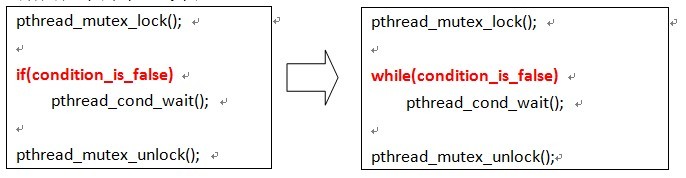
cv.notify\_all();

Linux Monitor： Linux管程配合条件变量使用，那就还要保证原子性

使用了Ubuntu 16.04系统，才能成功支持pthread库。之前试过了kali以及Ubuntu 12，但是都只是能编译成功,在kali里面我gdb调试是到了create\_pthread就会说找不到什么什么然后抛出异常产生段错误。

Linux中帮助中提到的：在多核处理器下，pthread\_cond\_signal可能会激活多于一个线程（阻塞在条件变量上的线程）。结果是，当一个线程调用pthread\_cond\_signal()后，多个调用pthread\_cond\_wait()或pthread\_cond\_timedwait()的线程返回。这种效应成为”虚假唤醒”(spurious wakeup)

“虚假唤醒”解决办法：



pthread\_cond\_t notempty = PTHREAD\_COND\_INITIALIZER;

pthread\_cond\_t notfull = PTHREAD\_COND\_INITIALIZER;

int pthread\_cond\_wait(pthread\_cond\_t \*cond,

pthread\_mutex\_t \*mutex);

参数说明：

cond——条件变量指针。

mutex——互斥信号量指针。

返回值：成功则返回0，失败则返回出错编号。

这个函数不同于：

pthread\_mutex\_unlock(mtx);

pthread\_cond\_just\_wait(cv);

pthread\_mutex\_lock(mtx);

拆分掉以后就会失去原子性，然后你的unlock唤醒的可能就是其他线程

先将线程附加到等待队列（这一步是加锁的），释放mutex，进入等待，确保unlock之后唤醒的是本线程。管程内部也有资源等待队列，入口有一个入口等待队列。

pthread\_cond\_signal

功能：释放条件变量，唤醒一个等待在指定条件变量上的线程。

格式：int pthread\_cond\_signal(pthread\_cond\_t \*cond);

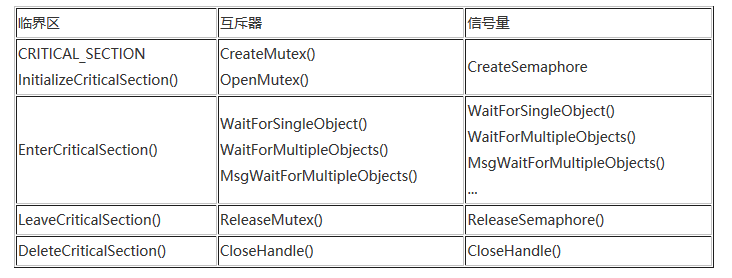
参数说明：

cond——将要释放的条件变量指针。

返回值：成功则返回0，失败则返回出错编号。

45. 在Posix.1基本原理一文声称，有了互斥锁和条件变量还提供信号量的原因是：“本标准提供信号量的而主要目的是提供一种进程间同步的方式；这些进程可能共享也可能不共享内存区。互斥锁和条件变量是作为线程间的同步机制说明的

46.windows下： 这里是对上面没有提到的部分进行补充



顺便希望你能注意，没想道这些函数首字母就开始大写了。

46.5僵尸进程

在fork()/execve（）过程中，假设子进程结束时父进程仍存在，而父进程fork（）之前既没安装SIGCHLD信号处理函数调用waitpid（）等待子进程结束，又没有显式忽略该信号，则子进程成为僵尸进程（父进程没有为子进程收尸），无法正常结束，此时即使是root身份kill -9也不能杀死僵尸进程。如果这时父进程结束了，那么[init进程](https://baike.baidu.com/item/init%E8%BF%9B%E7%A8%8B" \t "/home/vega/Documents\\x/_blank)自动会接手这个子进程，为它收尸，它还是能被清除的。但是如果父进程是一个循环，不会结束，那么子进程就会一直保持僵尸状态，这就是为什么系统中有时会有很多的僵尸进程。

查看僵尸进程，利用命令ps，可以看到有标记为Z的进程就是僵尸进程。

守护进程会通过孤儿进程的方式，所以只要编写者别搞事情，一定不会成为僵尸进程。

47. 当线程组的最后一个线程退出时，如果发现：

·该线程不是线程组的主线程。

·线程组的主线程已经退出，且处于僵尸状态。

·自己是最后一个线程。

同时满足这三个条件的时候，该子进程就需要冒充线程组的组长，即以子进程的主线程的身份来通知父进程。

48. 协程coroutine（又称微线程，纤程）：

子程序调用是通过栈实现的，一个线程就是执行一个子程序。协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行，协程的特点在于是一个线程执行，也就减少了上下文切换。

它是非抢占式的（不存在调度执行顺序无法确定），导致多任务时间片不能公平分享，所以后来全部废弃了协程改成抢占式的线程。抢占式的线程更加能利用多核优势。不管是进程还是线程，每次阻塞、切换都需要陷入系统调用(system call)，先让CPU跑操作系统的调度程序，然后再由调度程序决定该跑哪一个进程(线程)。协程是用户态线程写法也不会存在这部分调用，所以说是非抢占。

从上面可以看到，实现一个用户态线程有两个必须要处理的问题：一是碰着阻塞式I\O会导致整个进程被挂起；二是由于缺乏时钟阻塞，进程需要自己拥有调度线程的能力。如果一种实现使得每个线程需要自己通过调用某个方法，主动交出控制权。那么我们就称这种用户态线程是协作式的，即是**协程**。

我们在x360、xbox1和ps4上做游戏的时候，开线程用来做数据加载、解压这种不需要或者很少需要数据同步的任务的时候效率杠杠的，官方文档也建议，协程只是为了老代码移植和兼容性，不推荐新代码使用。

协程一个好处是按照同步思维写异步代码

Lua的协程其实是单线程对多线程的模拟。本质是单线程的，所以并不能发挥多核并发的优势。Python对协程的支持还非常有限，用在generator中的yield可以一定程度上实现协程

import time

def consumer():

r = ''

while True:

n = yield r

if not n:

return

print('[CONSUMER] Consuming %s...' % n)

time.sleep(1)

r = '200 OK'

def produce(c):

c.next()

n = 0

while n < 5:

n = n + 1

print('[PRODUCER] Producing %s...' % n)

r = c.send(n)

print('[PRODUCER] Consumer return: %s' % r)

c.close()

if \_\_name\_\_=='\_\_main\_\_':

c = consumer()

produce(c)

注意到consumer函数是一个generator（生成器），把一个consumer传入produce后：

1. 首先调用c.next()启动生成器；
2. 然后，一旦生产了东西，通过c.send(n)切换到consumer执行；
3. consumer通过yield拿到消息，处理，又通过yield把结果传回；
4. produce拿到consumer处理的结果，继续生产下一条消息；
5. produce决定不生产了，通过c.close()关闭consumer，整个过程结束。

整个流程无锁，由一个线程执行，produce和consumer协作完成任务，所以称为“协程”，而非线程的抢占式多任务。

最后套用Donald Knuth的一句话总结协程的特点：

“子程序就是协程的一种特例。”

49.消息服务擅长于解决多系统、异构系统间的数据交换（消息通知/通讯）问题，你也可以把它用于系统间服务的相互调用（RPC）。RabbitMQ就是当前最主流的消息中间件之一。

50. 我们可以通过锁定单个节点而不是整个链表来提高并发。

**乐观同步**：不需要获取锁就可以查找，对找到的节点进行加锁，然后确认锁住的节点是正确的；如果一个同步冲突导致节点被错误的锁定，则释放这些锁重新开始。

**惰性同步**: 设置一个flag这样的逻辑操作与类似于删除结点的链接这种对结构的物理改变分。每一次涉及到对结构的改变就需要记录改变的位置，而这会缩小能够进行逻辑操作的范围。

51. WaitForInputIdle()

函数作用：等待新进程完成它的初始化并等待用户输入。

53.冻结进程

Carbonite 这个内核模块能够冻结Linux中的task\_struct 中的进程状态，从而用于发现恶意的LKM。

1. Thrift是Google开发的用于跨语言RPC通信，它拥有功能强大的软件堆栈和代码生成引擎，允许定义一个简单的IDL文件来生成不同语言的代码，服务器端和客户端通过共享这个IDL文件来构建来完成通信。Flume的Thrift Source就是Thrift RPC的服务端，需要自己实现一个客户端用于发送数据。

<http://www.cnblogs.com/vincent-vg/p/5813505.html>

55.自旋锁是专为防止多处理器并发而引入的一种锁，它在[内核](https://baike.baidu.com/item/%E5%86%85%E6%A0%B8" \t "/home/vega/Documents\\x/_blank)中大量应用于[中断处理](https://baike.baidu.com/item/%E4%B8%AD%E6%96%AD%E5%A4%84%E7%90%86" \t "/home/vega/Documents\\x/_blank)等部分（对于单处理器来说，防止中断处理中的并发可简单采用关闭中断的方式，即在[标志寄存器](https://baike.baidu.com/item/%E6%A0%87%E5%BF%97%E5%AF%84%E5%AD%98%E5%99%A8" \t "/home/vega/Documents\\x/_blank)中关闭/打开中断标志位，不需要自旋锁）。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。

56.惊群现象：指一个fd的事件被触发后，等候这个fd的所有线程/进程都被唤醒。虽然都被唤醒，但是只有一个会去响应。

57. 多进程 相比较 多线程 的好处在于如果因为一些问题导致一个线程崩溃，多进程还可以补救。

因为一直固守单进程技术，而在内存使用效率上被 Chrome、Safari、Edge 等浏览器甩在后面的 Firefox，在最新的版本中，总算将迎来改观。早些时候发布的 Firefox 54 终于用上了多进程的技术，它可以将不同分页分为不同进程，最多四进程一起跑，能够效率更高地对电脑硬件加以利用。但是我的win10上面，firefox还是多进程，好像说是多线程需要开启设置。

1. CPU亲缘性可以分为两大类——软亲缘性和硬亲缘性:

Linux 内核进程调度器天生就具有被称为 CPU 软亲缘性（soft affinity） 的特性，这意味着进程通常不会在处理器之间频繁迁移。这种状态正是我们希望的，因为进程迁移的频率小就意味着产生的负载小。但不代表不会进行小范围的迁移。

      CPU 硬亲缘性是指通过Linux提供的相关CPU亲缘性设置接口，显示的指定某个进程固定的某个处理器上运行。本文所提到的CPU亲缘性主要是指硬亲缘性。Nginx支持。

Windows最大线程数：   
在32位系统中，默认情况下一个线程的栈要预留1M的内存空间，而一个进程中可用的内存空间只有2G，所以理论上一个进程中最多可以开2048个线程

Linux最大进程数与最大线程数，测试环境：2核4G内存 64位系统   
最大进程数（pid\_max）：32768（理论值）   
最大用户进程数（max user processes）：10240（理论值）   
最大线程数（threads-max）：77132（理论值）   
最大用户线程数：1024（理论值）

1. 阻塞加个超时返回，或者设置为非阻塞
2. 如何在不加锁的情况下实现多线程安全访问一个数据结构？当时没有答上来，后来想了想，应该使用观察者模式，一个线程负责修改，剩下的线程读取就行了。
3. 共享内存允许两个或多个进程共享一给定的存储区，因为数据不需要来回复制，所以是最快的一种进程间通信机制。共享内存的使用有两种方式：
   1. mmap：通过映射一个普通文件实现

相关读写代码： <https://blog.csdn.net/hj605635529/article/details/73163513>

mmap操作更加方便，因为使用磁盘存储，所以数据不会因为意外丢失。感觉和管道有点像，但是管道信息读了就会没掉，而且未读取的信息可以一直长下去。

缺点：每一个进程中都会开辟出一块空间，而且信息的转储涉及磁盘操作。



* 1. shm：每个进程的共享内存都直接映射到实际物理存储器里面。

通过int shmget(key\_t key, size\_t size, int shmflg);在物理内存创建一个共享内存，返回共享内存的编号。  
通过void \*shmat(int shmid, constvoid shmaddr,int shmflg);连接成功后把共享内存区对象映射到调用进程的地址空间  
通过void \*shmdt(constvoid\* shmaddr);断开用户级页表到共享内存的那根箭头。  
通过int shmctl(int shmid, int cmd, struct shmid\_ds\* buf);释放物理内存中的那块共享内存。

映射的时候不会真正改变进程的页表，还是需要一次缺页异常。