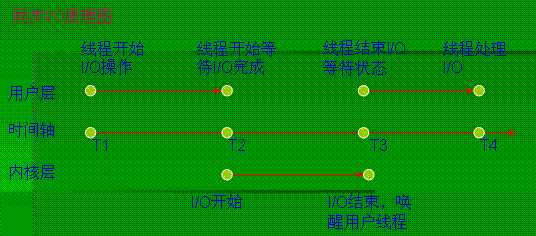
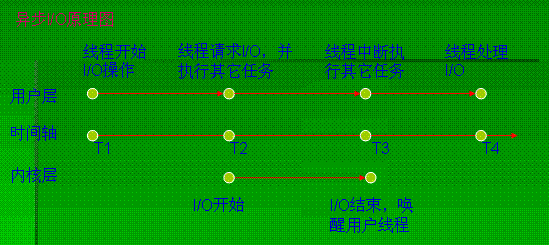
同步IO和异步IO

有两种类型的文件IO同步：同步文件IO和异步文件IO。异步文件IO也就是重叠IO。在同步文件IO中，线程启动一个IO操作然后就立即进入等待状态，直到IO操作完成后才醒来继续执行。而异步文件IO方式中，线程发送一个IO请求到内核，然后继续处理其他的事情，内核完成IO请求后，将会通知线程IO操作完成了。  
  
如果IO请求需要大量时间执行的话，异步文件IO方式可以显著提高效率，因为在线程等待的这段时间内，CPU将会调度其他线程进行执行，如果没有其他线程需要执行的话，这段时间将会浪费掉（可能会调度操作系统的零页线程）。如果IO请求操作很快，用异步IO方式反而还低效，还不如用同步IO方式。  
同步IO在同一时刻只允许一个IO操作，也就是说对于同一个文件句柄的IO操作是序列化的，即使使用两个线程也不能同时对同一个文件句柄同时发出读写操作。重叠IO允许一个或多个线程同时发出IO请求。  
异步IO在请求完成时，通过将文件句柄设为有信号状态来通知应用程序，或者应用程序通过GetOverlappedResult察看IO请求是否完成，也可以通过一个事件对象来通知应用程序。

简单的说“同步在编程里，一般是指某个IO操作执行完后，才可以执行后面的操作。异步则是，将某个操作给系统，主线程去忙别的事情，等内核完成操作后通知主线程异步操作已经完成。”

**I、Windows同步I/O与异步I/O**  
一、  同步I/O和异步I/O  
  
在介绍这部分内容之前先来认识下“异步I/O”。  
  说起异步IO，很容易联想到同步I/O，对于同一个I/O对象句柄在同一时刻只允许一个I/O操作，其原理如下图所示：  
    
  
  显然，当内核真正处理I/O的时间段（T2~T4），用户线程是处于等待状态的，如果这个时间段比较段的话，没有什么影响；倘若这个时间段很长的话，线程就会长时间处于挂起状态。事实上，该线程完全可以利用这段时间用处理其他事务。  
  
  异步I/O恰好可以解决同步I/O中的问题，而且支持对同一个I/O对象的并行处理，其原理如下图所示：  
    
  
  异步I/O在I/O请求完成时，可以使用让I/O对象或者事件对象受信来通知用户线程，而用户线程中可以使用GetOverlappedResult来查看I/O的执行情况。  
    
由于异步I/O在进行I/O请求后会立即返回，这样就会产生一个问题：“程序是如何取得I/O处理的结果的？”。  
  
  有多种方法可以实现异步I/O，其不同资料上的分类一般都不尽相同，但原理上都类似，这里我把实现异步I/O的方法分为3类，本文就针对这3类方法进行详细的讨论。  
（1）重叠I/O  
（2）异步过程调用（APC），扩展I/O  
（3）使用完成端口（IOCP）  
  
二、使用重叠I/O实现异步I/O  
    
  同一个线程可以对多个I/O对象进行I/O操作，不同的线程也可以对同一个I/O对象进行操作，在我的理解中，重叠的命名就是这么来的。  
  
  在使用重叠I/O时，线程需要创建OVERLAPPED结构以供I/O处理。该结构中最重要的成员是hEvent，它是作为一个同步对象而存在，如果hEvent为NULL，那么此时的同步对象即为文件句柄、管道句柄等I/O操作对象。当I/O完成后，会使这里的同步对象受信，从而通知用户线程。  
  
  由于在进行I/O请求后会立即返回，但有时用户线程需要知道I/O当前的执行情况，此时就可以使用GetOverlappedResult。如果该函数的bWait参数为true,那么改函数就会阻塞线程直到目标I/O处理完成为止；如果bWait为false，那么就会立即返回，如果此时的I/O尚未完，调用GetLastError就会返回ERROR\_IO\_INCOMPLETE。  
  
代码示例一：

代码:

DWORD   nReadByte ;

BYTE   bBuf[BUF\_SIZE] ;

OVERLAPPED ov = { 0, 0, 0, 0, NULL } ;  // hEvent = NULL ;

HANDLE hFile = CreateFile ( ……, FILE\_FLAG\_OVERLAPPED, …… ) ;

ReadFile ( hFile, bBuf, sizeof(bBuf), &nReadByte, &ov ) ;

// 由于此时hEvent=NULL，所以同步对象为hFile,下面两句的效果一样

WaitForSingleObject ( hFile, INFINITE ) ;

//GetOverlappedResult ( hFile, &ov, &nRead, TRUE ) ;

这段代码在调用ReadFile后会立即返回，但在随后的WaitForSingleObject或者GetOverlappedResult中阻塞，利用同步对象hFile进行同步。  
  
  这段代码在这里可以实现正常的异步I/O，但存在一个问题，倘若现在需要对hFile句柄进行多个I/O操作，就会出现问题。见下面这段代码。  
  
代码示例二：

代码:

DWORD   nReadByte ;

BYTE   bBuf1[BUF\_SIZE],bBuf2[BUF\_SIZE],bBuf3[BUF\_SIZE] ;

OVERLAPPED ov1 = { 0, 0, 0, 0, NULL } ;

OVERLAPPED ov2 = { 0, 0, 0, 0, NULL } ;

OVERLAPPED ov3 = { 0, 0, 0, 0, NULL } ;

HANDLE hFile = CreateFile ( ……, FILE\_FLAG\_OVERLAPPED, …… ) ;

ReadFile ( hFile, bBuf1, sizeof(bBuf1), &nReadByte, &ov1 ) ;

ReadFile ( hFile, bBuf2, sizeof(bBuf2), &nReadByte, &ov2 ) ;

ReadFile ( hFile, bBuf3, sizeof(bBuf3), &nReadByte, &ov3 ) ;

//假设三个I/O处理的时间比较长，到这里还没有结束

GetOverlappedResult ( hFile, &ov1, &nRead, TRUE ) ;

  这里对于hFile有三个重叠的I/O操作，但他们的同步对象却都为hFile。使用GetOverlappedResult进行等待操作，这里看似在等待第一个I/O处理的完成，其实只要有任何一个I/O处理完成，该函数就会返回，相当于忽略了其他两个I/O操作的结果。  
  
  其实，这里有一个很重要的原则：对于一个重叠句柄上有多于一个I/O操作的时候，应该使用事件对象而不是文件句柄来实现同步。正确的实现见示例三。  
    
代码示例三：

代码:

DWORD   nReadByte ;

BYTE   bBuf1[BUF\_SIZE],bBuf2[BUF\_SIZE],bBuf3[BUF\_SIZE] ;

HANDLE  hEvent1 = CreateEvent ( NULL, FALSE, FALSE, NULL ) ;

HANDLE  hEvent2 = CreateEvent ( NULL, FALSE, FALSE, NULL ) ;

HANDLE  hEvent3 = CreateEvent ( NULL, FALSE, FALSE, NULL ) ;

OVERLAPPED ov1 = { 0, 0, 0, 0, hEvent1 } ;

OVERLAPPED ov2 = { 0, 0, 0, 0, hEvent2 } ;

OVERLAPPED ov3 = { 0, 0, 0, 0, hEvent3 } ;

HANDLE hFile = CreateFile ( ……, FILE\_FLAG\_OVERLAPPED, …… ) ;

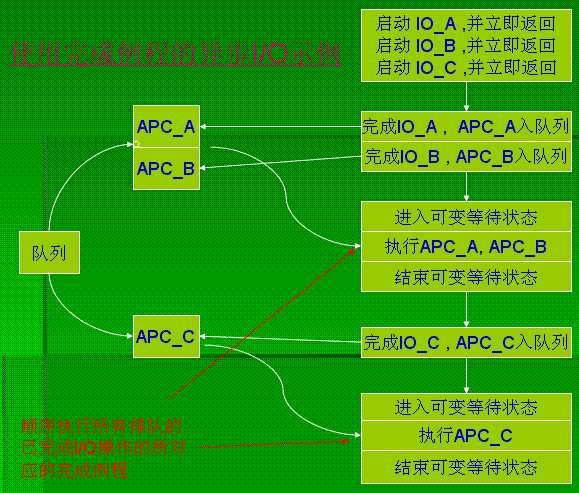
ReadFile ( hFile, bBuf1, sizeof(bBuf1), &nReadByte, &ov1 ) ;

ReadFile ( hFile, bBuf2, sizeof(bBuf2), &nReadByte, &ov2 ) ;

ReadFile ( hFile, bBuf3, sizeof(bBuf3), &nReadByte, &ov3 ) ;

//此时3个I/O操作的同步对象分别为hEvent1,hEvent2,hEvent3

GetOverlappedResult ( hFile, &ov1, &nRead, TRUE ) ;

  这样，这个GetOverlappedResult就可以实现对第一个I/O处理的等待  
关于重叠I/O的就讨论到这里，关于重叠I/O的实际应用，可以参考《Windows系统编程之进程通信》其中的命名管道实例。  
http://bbs.pediy.com/showthread.php?s=&threadid=26252  
   
三、  使用异步过程调用实现异步I/O  
  
异步过程调用（APC），即在特定的上下文中异步的执行一个调用。在异步I/O中可以使用APC，即让操作系统的IO系统在完成异步I/O后立即调用你的程序。（在有些资料中，把异步I/O中的APC称为“完成例程”，感觉这个名称比较贴切，下文就以“完成例程”来表述。另外通常APC是作为线程同步这一块的内容，这里尽量淡化这个概念以免混淆。关于APC的详细内容到线程同步时再介绍 ）  
  
这里需要注意三点：  
（1）  APC总是在调用线程中被调用；  
（2）  当执行APC时，调用线程会进入可变等待状态；  
（3）  线程需要使用扩展I/O系列函数，例如ReadFileEx,WriteFileEx, 另外可变等待函数也是必须的（至少下面其中之一）：  
WaitForSingleObjectEx  
WaitForMultipleObjectEx  
SleepEx  
SignalObjectAndWait  
MsgWaitForMultipleObjectsEx  
    
  在使用ReadFileEx,WriteFileEx时，重叠结构OVERLAPPED中的hEvent成员并非一定要指定，因为系统会忽略它。当多个IO操作共用同一个完成例程时，可以使用hEvent来携带序号等信息，用于区别不同的I/O操作，因为该重叠结构会传递给完成例程。如果多个IO操作使用的完成例程都不相同时，则直接把hEvent设置为NULL就可以了。  
  
在系统调用完成例程有两个条件：  
（1）  I/O操作必须完成  
（2）  调用线程处于可变等待状态  
  
对于第一个条件比较容易，显然完成例程只有在I/O操作完成时才调用；至于第二个条件就需要进行认为的控制，通过使用可变等待函数，让调用线程处于可变等待状态，这样就可以执行完成例程了。这里可以通过调节调用可变等待函数的时机来控制完成例程的执行，即可以确保完成例程不会被过早的执行。  
  
当线程具有多个完成例程时，就会形成一个队列。使用可变等待函数使线程进入可变等待状态时有一个表示超时值的参数，如果使用INFINITE，那么只有所有排队的完成例程被执行或者句柄获得信号时该等待函数才返回。  
  
上面已经对利用完成例程实现异步I/O的一些比较重要的细节进行的简洁的阐述，接下来就以一个实例来说明完成例程的具体实现过程。  
  
  
  
实例一：使用完成例程的异步I/O示例  
  
1、  设计目标  
体会完成例程的异步I/O实现原理及过程。  
  
2、  问题的分析与设计  
设计流程图如下：  
    
示图说明：  
  三个IO操作分别是IO\_A, IO\_B, IO\_C, 他们的完成例程分别是APC\_A, APC\_B, APC\_C。IO\_A, IO\_B是两个很短的IO操作，IO\_C是一个比较费时的IO操作。  
3、  详细设计（关键代码如下,具体参见附件中的源代码CompletionRoutine）

代码:

VOID WINAPI APC\_A ( DWORD dwError, DWORD cbTransferred, LPOVERLAPPED lpo )

{

  pTempInfo.push\_back ( "执行IO\_A的完成例程" ) ;

}

VOID WINAPI APC\_B ( DWORD dwError, DWORD cbTransferred, LPOVERLAPPED lpo )

{

  pTempInfo.push\_back ( "执行IO\_B的完成例程" ) ;

}

VOID WINAPI APC\_C ( DWORD dwError, DWORD cbTransferred, LPOVERLAPPED lpo )

{

  pTempInfo.push\_back ( "执行IO\_C的完成例程" ) ;

}

void CCompletionRoutineDlg::OnTest()

{

  // TODO: Add your control notification handler code here

  HANDLE    hFile\_A, hFile\_B, hFile\_C ;

  OVERLAPPED  ov\_A = {0}, ov\_B = {0}, ov\_C = {0} ;

#define C\_SIZE 1024 \* 1024 \* 32

  string  szText\_A = "Sample A !" ;

  string  szText\_B = "Sampel B !" ;

  string  szText\_C ;

  szText\_C.resize ( C\_SIZE ) ;

  memset ( &(szText\_C[0]), 0x40, C\_SIZE ) ;

  pTempInfo.clear () ;

  hFile\_A = CreateFile ( "A.txt", GENERIC\_WRITE, 0, NULL, /

              CREATE\_ALWAYS, FILE\_FLAG\_OVERLAPPED, NULL ) ;

  hFile\_B = CreateFile ( "B.txt", GENERIC\_WRITE, 0, NULL, /

              CREATE\_ALWAYS, FILE\_FLAG\_OVERLAPPED, NULL ) ;

  hFile\_C = CreateFile ( "C.txt", GENERIC\_WRITE, 0, NULL, /

              CREATE\_ALWAYS, FILE\_FLAG\_OVERLAPPED, NULL ) ;

  WriteFileEx ( hFile\_A, &(szText\_A[0]), szText\_A.length(), &ov\_A, APC\_A ) ;

  pTempInfo.push\_back ( "启动IO\_A, 并立即返回" ) ;

  WriteFileEx ( hFile\_B, &(szText\_B[0]), szText\_B.length(), &ov\_B, APC\_B ) ;

  pTempInfo.push\_back ( "启动IO\_B, 并立即返回" ) ;

  WriteFileEx ( hFile\_C, &(szText\_C[0]), szText\_C.size(), &ov\_C, APC\_C ) ;

  pTempInfo.push\_back ( "启动IO\_C, 并立即返回" ) ;

  pTempInfo.push\_back ( "进入可变等待状态" ) ;

  SleepEx ( 1, true ) ;

  pTempInfo.push\_back ( "结束可变等待状态" ) ;

  pTempInfo.push\_back ( "进入可变等待状态" ) ;

  SleepEx ( 10000, true ) ;

  pTempInfo.push\_back ( "结束可变等待状态" ) ;

  CloseHandle ( hFile\_A ) ;

  CloseHandle ( hFile\_B ) ;

  CloseHandle ( hFile\_C ) ;

  m\_ListBox.ResetContent () ;

  list<string>::iterator p ;

  for ( p = pTempInfo.begin(); p != pTempInfo.end(); p++ )

  {

    m\_ListBox.AddString ( p->data() ) ;

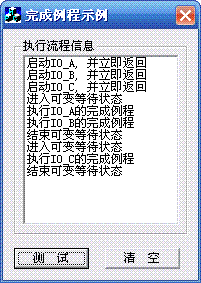
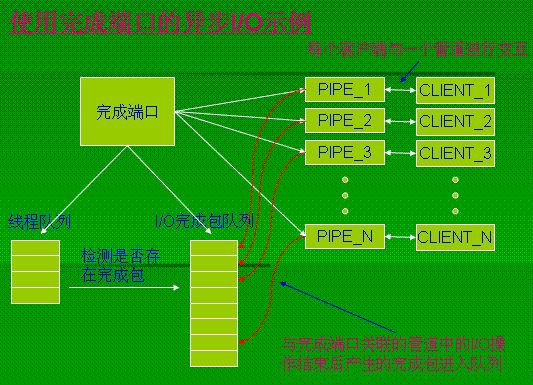
  }

  DeleteFile ( "A.txt" ) ;

  DeleteFile ( "B.txt" ) ;

  DeleteFile ( "C.txt" ) ;

}

执行后的效果如下（WinXP+SP2+VC6.0）：  
    
  
4、  心得体会  
每当一个IO操作结束时会产生一个完成信息，如果该IO操作有完成例程的话就添加到完成例程队列。一旦调用线程进入可变等待状态，就会依次执行队列中的完成例程。  
在这个示例中还有一个问题，如果把这个软件放在系统分区的文件目录下可以正常执行，而放在其他盘符下就会出现问题，执行结果就不同，真是奇怪了。  
  
  
四、使用完成端口（IOCP）  
  
实例二、使用IOCP的异步I/O示例  
1、设计目标  
体会完成端口的异步I/O实现原理及过程。  
  
2、  问题的分析与设计  


说明：  
  每个客户端与一个管道进行交互，而在交互过程中I/O操作结束后产生的完成包就会进入“I/O完成包队列”。完成端口的线程队列中的线程使用GetQueuedCompletionStatus来检测“I/O完成包队列”中是否有完成包信息。   
3、详细设计（关键代码如下）

代码:

UINT ServerThread ( LPVOID lpParameter )

{

  ……

  while ( true )

  {

    GetQueuedCompletionStatus ( pMyDlg->hCompletionPort, &cbTrans, &dwCompletionKey, &lpov, INFINITE ) ;

    if ( dwCompletionKey == -1 )

      break ;

    // 读取管道信息

    // 响应管道信息（写入）

  }

  return 0 ;

}

void CMyDlg::OnStart()

{

  // 创建完成端口

  hCompletionPort = CreateIoCompletionPort ( INVALID\_HANDLE\_VALUE, NULL, 0, nMaxThread ) ;

  CString lpPipeName = "////.//Pipe//NamedPipe" ;

  for ( UINT i = 0; i < nMaxPipe; i++ )

  {

    // 创建命名管道

    PipeInst[i].hPipe =  CreateNamedPipe ( lpPipeName, PIPE\_ACCESS\_DUPLEX|FILE\_FLAG\_OVERLAPPED, /

          PIPE\_TYPE\_BYTE|PIPE\_READMODE\_BYTE|PIPE\_WAIT, nMaxPipe, 0, 0, INFINITE, NULL ) ;

    ……

    // 把命名管道与完成端口关联起来

    HANDLE hRet = CreateIoCompletionPort ( PipeInst[i].hPipe, hCompletionPort, i, nMaxThread ) ;

    ……

    // 等待连接

    ConnectNamedPipe ( PipeInst[i].hPipe, &(PipeInst[i].ov) ) ;

  }

  // 创建线程

  for ( i = 0; i < nMaxThread; i++ )

  {

    hThread[i] = AfxBeginThread ( ServerThread, NULL, THREAD\_PRIORITY\_NORMAL ) ;

  }

  ……

}

void CMyDlg::OnStop()

{

  for ( UINT i = 0; i < nMaxThread; i++ )

  {

    // 用来唤醒线程的虚假I/O完成包

    PostQueuedCompletionStatus ( hCompletionPort, 0, -1, NULL ) ;

    CloseHandle ( hThread[i] ) ;

  }

  for ( i = 0; i < nMaxPipe; i++ )

  {

    DisconnectNamedPipe ( PipeInst[i].hPipe ) ;

    CloseHandle ( PipeInst[i].hPipe ) ;

  }

  ……

}

4、心得体会  
  上面这个例子是关于完成端口的简单应用。可以这样来理解完成端口，它与三种资源相关分别是管道、I/O完成包队列、线程队列，它的作用是协调这三种资源。

参考：<http://www.pediy.com/bbshtml/bbs8/pediy8-709.htm>

**II、Linux同步I/O与异步I/O**

linux中SIO（同步IO）和AIO（异步IO）机制  
1.read/write:  
对于read操作来说，它是同步的。也就是说只有当申请读的内容真正存放到buffer中后（user mode的buffer），read函数才返回。在此期间，它会因为等待IO完成而被阻塞。研究过源码的朋友应该知道，这个阻塞发生在两个地方：一是read操作刚刚发起，kernel会检查其它进程的need\_sched标志，如果有其它进程需要调度，主动阻塞read操作，这个时候其实I/O操作还没有发起。二是I/O操作发起后，调用lock\_page对已经加锁的页面申请锁，这时由于页面已经加锁，所以加锁操作被阻塞，从而read操作阻塞，直到I/O操作完成后页面被解锁，read操作继续执行。所以说read是同步的，其阻塞的原因如上。  
对于write操作通常是异步的。因为linux中有page cache机制，所有的写操作实际上是把文件对应的page cache中的相应页设置为dirty，然后write操作返回。这个时候对文件的修改并没有真正写到磁盘上去。所以说write是异步的，这种方式下write不会被阻塞。如果设置了O\_SYNC标志的文件，write操作再返回前会把修改的页flush到磁盘上去，发起真正的I/O请求，这种模式下会阻塞。  
2.Direct I/O  
linux支持Direct I/O, 以O\_DIRCET标志打开的文件，在read和write的时候会绕开page cache，直接使用user mode的buffer做为I/O操作的buffer。这种情况下的read和write直接发起I/O操作，都是同步的，并会被阻塞。  
3.AIO  
目前大多数的linux用的AIO是基于2.4内核中的patch，使用librt库中的接口。这种方式实现很简单，就是一个父进程clone出子进程帮其做I/O，完成后通过signal或者callback通知父进程。用户看来是AIO，实质还是SIO。linux kernel中AIO的实现概念类似，只不过是以一组kernel thread去做的。这些kernel thread做I/O的时候使用的是和Direct I/O相同的方式。  
4.mmap()  
抛开它中讲vm\_area和page cache映射在一起的机制不说。真正发起I/O时和read、write使用的是相同的机制，同步阻塞。