

Multithreading Applications in Win32

(Chapter 6. Overlapped I/O or Juggling Behind Your Back)

Seong Jong Choi
chois@uos.ac.kr

[Multimedia Lab.](#)

Dept. of Electrical and Computer Eng.
University of Seoul
Seoul, Korea

Contents

- 6.0 Main Objectives
- 6.1 Overview
- 6.2 [The Win32 File Calls](#)
- 6.3 [-Signaled File Handles](#)
- 6.4 [-Signaled Event Objects](#)
- 6.5 [-A Synchronous Procedure Calls\(APCs\)](#)
- 6.6 Drawbacks to Overlapped I/O with Files
- 6.7 [-I/O Completion Ports](#)
- 6.8 Using Overlapped I/O with Sockets
- 6.9 Summary

6.0 Main Objectives

- How to use overlapped, or asynchronous, I/O.

6.1 Overview

- Waiting for I/O to complete is not a particularly good use of time for your program.
- An obvious solution to this problem is would be to use another thread to do the I/O
 - (problem : synchronization , handling errors , put up dialogs ...)
- Overlapped I/O in Win32
 - You can ask the operating system to transfer data for you and to notify you when it is finished

6.1 Overview

- Non-overlapped vs. overlapped
 - (I/O Devices are slow !)
 - **Non-overlapped** : it is not a particularly good use of time
 - An awfully long time is needed to finish for an application
 - **Overlapped** : you can ask the operating system to transfer data for you and to notify when it is finished
 - This arrangement leaves your application free to continue processing while the I/O is being performed

Overlapped I/O (Windows NT only)

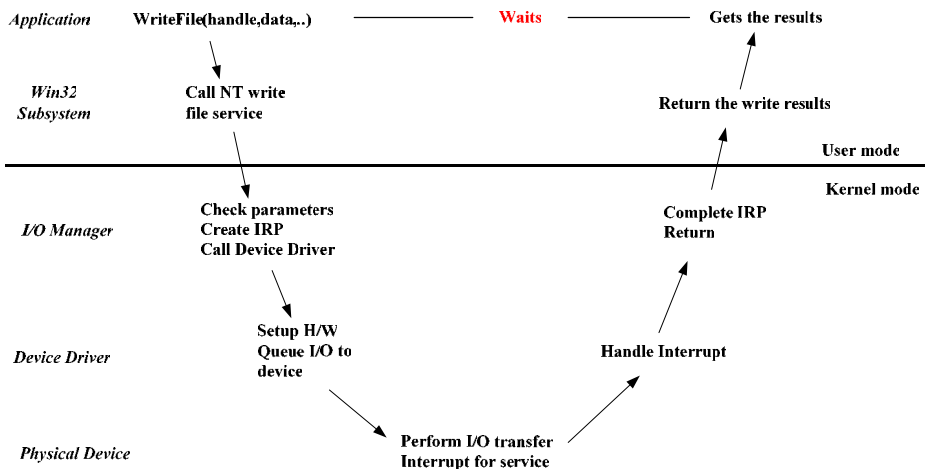
- WIN95 does not support overlapped I/O completely
- = asynchronous, non-blocking I/O (other OS)

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-5

Synchronous I/O Processing

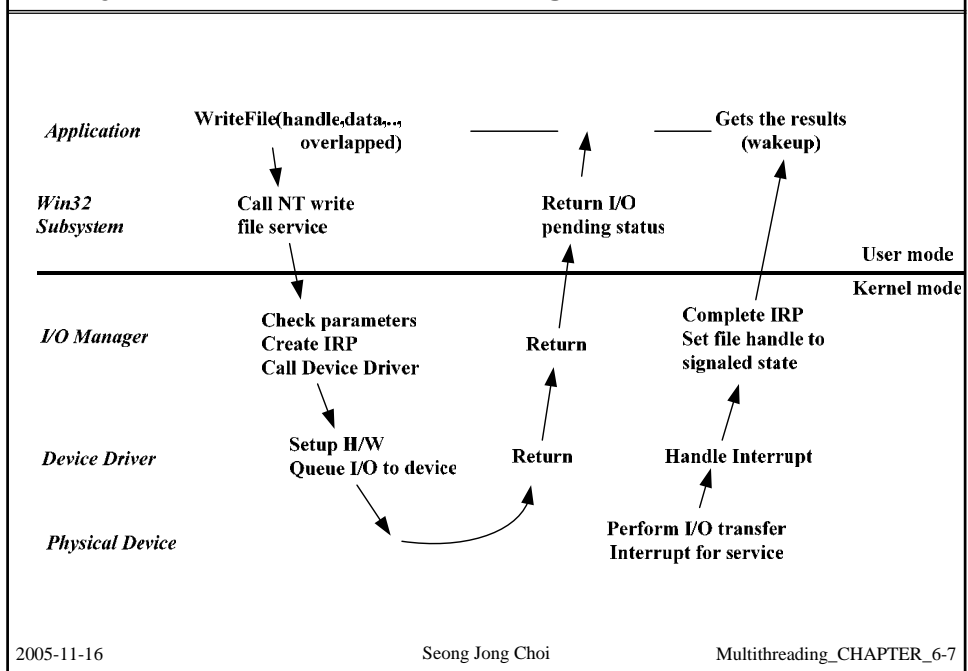


2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-6

Asynchronous I/O Processing



6.1 Overview

- 4 MS ways for overlapped I/O
 - Signaled file handles
 - Signaled event objects
 - Asynchronous Procedure Calls (APCs)
 - I/O completion ports
 - Important !! Because they are the only mechanism that is suited for high volume-servers that must maintain many simultaneous connections

6.2 The Win32 File Calls

- Three basic calls for performing I/O in Win32
 - Create File()
 - ReadFile()
 - WriteFile()
 - There is no way for closing a file ??
 - You can use CloseHandle() to do this
- CreateFile() can be used to open a wide variety of resources
 - Files(hard disk , floppy disk , CD-ROM , and others
 - Serial and parallel ports
 - Named pipes
 - Console (CH.8)

6.2 The Win32 File Calls

1. CreateFile()

```
HANDLE CreateFile(  
LPCTSTR lpFileName,  
DWORD dwDesiredAccess,  
DWORD dwShareMode,  
LPSECURITY_ATTRIBUTES //lpSecurityAttributes,  
DWORD dwCreationDistribution, //how to create  
DWORD dwFlagsAndAttributes, //file Attributes  
HANDLE hTemplateFile  
);
```

6.2 The Win32 File Calls

- dwFlagAndAttributes (6th parameter)
 - the Key to using overlapped I/O
 - use `FILE_FLAG_OVERLAPPED`

Characteristics of overlapped I/O

- If a handle is created for overlapped I/O, every operation on the file handle must be overlapped.
- Can read and/or write from multiple parts of the file at the same time.
- No concept of "current file offset."
- Every I/O call must include file offset.
- No support for overlapped text-based I/O. (e.g. reading a line)

6.2 The Win32 File Calls

```
BOOL ReadFile(
    HANDLE hFile,
    LPVOID lpBuffer,
    DWORD nNumberOfBytesToRead,
    LPDWORD lpNumberOfBytesRead,
    LPOVERLAPPED lpOverlapped //ptr to overlapped info
);
```

```
BOOL WriteFile(
    HANDLE hFile,
    LPCVOID lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPDWORD lpNumberOfBytesWritten,
    LPOVERLAPPED lpOverlapped //ptr to overlapped info
);
```

6.2 The Win32 File Calls

- We will focus on the parameters for overlapped I/O operations
- We'll cover the rest of the input parameters later.
- Refer to MSDN or "Windows System Programming," by J.M. Hart

6.2 The Win32 File Calls

- Similar to `fread()` and `fwrite()` except for the last parameter
- You must supply a pointer to a valid `OVERLAPPED` structure when `CreateFile()` is called with `FILE_FLAG_OVERLAPPED`
- Return Value of the overlapped operations
 - `TRUE`, if the operations are completed immediately.
 - `FALSE`, if the operations are pending or if there are other errors.

The OVERLAPPED Structure

- Information for `file offset (or file position)`, and `event object`.
- Acts as a `key that uniquely identifies each overlapped operation` currently in process.
- Provides a `shared area between you and the system` where parameters can be passed in each direction.
- It is important to put the structure in a safe place
 - Typically a local variable is not a safe place because it will go out of scope too soon.
 - It is usually safest to `allocate the OVERLAPPED structure on the heap`, i.e. using `malloc()`.

6.2 The Win32 File Calls

```
typedef struct OVERLAPPED{  
    DWORD Internal;  
    DWORD InternalHigh;  
    DWORD Offset;  
    DWORD OffsetHigh;  
    HANDLE hEvent;  
} OVERLAPPED;
```

Member	Description
Internal	Reserved for operating system use. This member, which specifies a system-dependent status, is valid when the GetOverlappedResult function returns without setting the extended error information to ERROR_IO_PENDING.

6.2 The Win32 File Call

Member	Description
InterHigh	Reserved for operating system use. This member, which specifies the length of the data transferred, is valid when the GetOverlappedResult function returns TRUE.
Offset	Specifies a file position at which to start the transfer. The file position is a byte offset from the start of the file. The calling process sets this member before calling the ReadFile or WriteFile function. This member is ignored when reading from or writing to named pipes and communications devices and should be zero.
OffsetHigh	Specifies the high word of the byte offset at which to start the transfer. This member is ignored when reading from or writing to named pipes and communications devices and should be zero.

6.2 The Win32 File Call

Member	Description
hEvent	Handle to a manual reset event set to the signaled state when the operation has been completed. The calling process must set this member either to zero or a valid event handle before calling any overlapped functions. To create an event object, use the CreateEvent function. Functions such as WriteFile set the event to the nonsignaled state before they begin an I/O operation.

6.3 Signaled File Handles

- The simplest type of overlapped I/O operation
- The file handle, as a kernel object, will be **signaled when the operation is complete**.

Steps:

1. Opening the file with `FILE_FLAG_OVERLAPPED`
2. Set up an `OVERLAPPED` structure
3. Call `ReadFile()` and pass the address of the `OVERLAPPED` structure
4. `GetOverlappedResult()` to find out what happened, `GetLastError()`.

IOBYFILE.C

```
hFile = CreateFile( szPath,
    GENERIC_READ,
    FILE_SHARE_READ|FILE_SHARE_WRITE,
    NULL,
    OPEN_EXISTING,
    FILE_FLAG_OVERLAPPED,
    NULL
);
if (hFile == INVALID_HANDLE_VALUE)
{
    printf("Could not open %s\n", szPath);
    return -1;
}
```

```
// Initialize the OVERLAPPED structure
memset(&overlap, 0, sizeof(overlap));
overlap.Offset = 1500;
```

```
// Request the data
rc = ReadFile(
    hFile,
    buf,
    READ_SIZE,
    &numread,
    &overlap
);
printf("Issued read request\n");
```

```
// Was the operation queued?
if (rc)
{
    //If return TRUE, The data was read successfully
    printf("Request was returned immediately\n");
}
```

```
else
{
    if (GetLastError() == ERROR_IO_PENDING)
    {
        // We could do something else for awhile here...
        printf("Request queued, waiting...\n");
        WaitForSingleObject(hFile, INFINITE);
        printf("Request completed.\n");
        rc = GetOverlappedResult(
            hFile,
            &overlap,
            &numread,
            FALSE
        );
        printf("Result was %d\n", rc);
    }
```

```
else
{
    // We should check for memory and quota errors here and retry.
    // See the samples IoByEvt and IoByAPC.

    // Something went wrong
    printf("Error reading file\n");
}
```

```
CloseHandle(hFile);
}
```

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-21

IOBYFILE.C – Part 1

```
hFile = CreateFile( szPath,
    GENERIC_READ,
    FILE_SHARE_READ|FILE_SHARE_WRITE,
    NULL,
    OPEN_EXISTING,
    FILE_FLAG_OVERLAPPED,
    NULL
);
if (hFile == INVALID_HANDLE_VALUE)
{
    printf("Could not open %s\n", szPath);
    return -1;
}
```

```
// Initialize the OVERLAPPED structure
memset(&overlap, 0, sizeof(overlap));
overlap.Offset = 1500;
```

```
// Request the data
rc = ReadFile(
    hFile,
    buf,
    READ_SIZE,
    &numread,
    &overlap
);
printf("Issued read request\n");
```

Step #1: Opening the file with
FILE_FLAG_OVERLAPPED

Step #2: Set up an OVERLAPPED
structure

Step #3: Call ReadFile() and pass the
address of the OVERLAPPED
structure

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-22

IOBYFILE.C – Part 2

If ReadFile() returns **TRUE**, the data was read immediately. In this case, the operation is not overlapped.

If ReadFile() returns **FALSE** and GetLastError() returns **ERROR_IO_PENDING**, still reading the data. Do something else and wait for the file object.

If ReadFile() returns **FALSE** and GetLastError() does not return **ERROR_IO_PENDING**, something else went wrong.

```
// Was the operation queued?  
if (rc)  
{  
    //If return TRUE, The data was read successfully  
    printf("Request was returned immediately\n");  
}
```

```
else  
{  
    if (GetLastError() == ERROR_IO_PENDING)  
    {  
        // We could do something else for awhile here...  
        printf("Request queued, waiting...\n");  
        WaitForSingleObject(hFile, INFINITE);  
        printf("Request completed.\n");  
        rc = GetOverlappedResult(  
            hFile,  
            &overlap,  
            &numread,  
            FALSE  
        );  
        printf("Result was %d\n", rc);  
    }  
}
```

```
else  
{  
    // We should check for memory and quota errors here and retry.  
    // See the samples IoByEvt and IoByAPC.  
  
    // Something went wrong  
    printf("Error reading file\n");  
}
```

```
CloseHandle(hFile);  
}
```

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-23

6.3 Signaled File Handles

```
BOOL GetOverlappedResult(  
    HANDLE hFile,  
    LPOVERLAPPED lpOverlapped,  
    LPDWORD lpNumberOfBytesTransferred,  
    BOOL bWait //True -> wait until finished  
)
```

Return Value

True : overlapped operation succeed

False : failed(maybe pending)-> GetLastError()

Let's see the Listing 6-1. (p.123)

2005-11-16

Seong Jong Choi

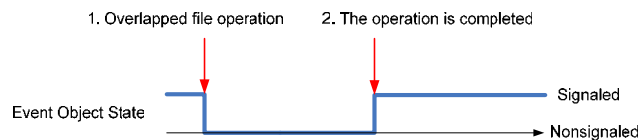
Multithreading_CHAPTER_6-24

6.4 Using Event Objects

- A limitation to using file handles as the signaling mechanism is that you can't tell which operation has completed when several operations on the same file handle
- Solution :
 - GetOverlappedResult() or
 - OVERLAPPED structure's hEvent

Using Event Object

- The system sets the state of the event object to **nonsignaled** when a call to the I/O function returns before the operation has been completed.
- The system sets the state of the event object to **signaled** when the operation has been completed



IOBYEVENT.C – main()

```
// Global variables
// Need to keep the events in their own array
// so we can wait on them.
HANDLE ghEvents[MAX_REQUESTS];
// Keep track of each individual I/O operation
OVERLAPPED gOverlapped[MAX_REQUESTS];
```

Define event objects and
OVERLAPPED Structure as global

```
int main() {
```

```
    ghFile = CreateFile( szPath,
        GENERIC_READ,
        FILE_SHARE_READ|FILE_SHARE_WRITE,
        NULL,
        OPEN_EXISTING,
        FILE_FLAG_OVERLAPPED,
        NULL
    );
```

Create file handle for overlapped
operation

```
    for (i=0; i<MAX_REQUESTS; i++)
    {
        QueueRequest(i, i*16384, READ_SIZE);
    }
```

Read READ_SIZE bytes from
i*6384 position for
MAX_REQUEST times

```
    // Wait for all the operations to complete.
    MTVERIFY( WaitForMultipleObjects(
        MAX_REQUESTS, ghEvents, TRUE, INFINITE
    ) != WAIT_FAILED );
```

Wait for all the event objects to be
signaled, i.e. all file read operations
are complete.

```
}
```

IOBYEVENT.C – QueueRequest() 1/2

```
//The function "QueueRequest()" try overlapped file read operation.
//If there's unrecoverable error or recoverable errors repeated more
//than MAX_TRY_COUNT, then return -1
```

```
int QueueRequest(int nIndex, DWORD dwLocation, DWORD dwAmount)
```

```
{
    MTVERIFY(
        ghEvents[nIndex] = CreateEvent(
            NULL, // No security
            TRUE, // Manual reset - extremely important!
            FALSE, // Initially set Event to non-signaled state
            NULL // No name
        )
    );
```

Create manual reset event objects

```
    gOverlapped[nIndex].hEvent = ghEvents[nIndex];
    gOverlapped[nIndex].Offset = dwLocation;
```

Prepare OVERLAPPED Structure

```
//Continued to the next page
```

IOBYEVENT.C – QueueRequest() 2/2

```
for (i=0; i<MAX_TRY_COUNT; i++) {
```

```
    rc = ReadFile(  
        ghFile,  
        gBuffers[nIndex],  
        dwAmount,  
        &dwNumread,  
        &gOverlapped[nIndex]  
    );
```

Start overlapped read operation

```
    // Handle success  
    if (rc) {  
        printf("Read #%d completed immediately.\n", nIndex);  
        return TRUE;  
    }
```

If the read operation completes immediately, then return from the function

```
    err = GetLastError();  
    // Handle the error that isn't an error. rc is zero here.  
    if (err == ERROR_IO_PENDING) {  
        // asynchronous i/o is still in progress  
        printf("Read #%d queued for overlapped I/O.\n", nIndex);  
        return TRUE;  
    }
```

If the read operation is still running, return from the function

```
    // Handle recoverable error  
    if (err == ERROR_INVALID_USER_BUFFER ||  
        err == ERROR_NOT_ENOUGH_QUOTA ||  
        err == ERROR_NOT_ENOUGH_MEMORY) {  
        Sleep(50); // Wait around and try later  
        continue;  
    }
```

If the error is recoverable, then wait 50msec and try the read operation again.

```
    // Give up on fatal error.  
    break;
```

If the error is unrecoverable, then go out of the loop

```
    printf("ReadFile failed.\n");  
    return -1;  
}
```

If there is unrecoverable error or if there are more than MAX_TRY_COUNT, then return -1

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-29

6.5 Asynchronous Procedure Calls(APCs)

- Problems of using event objects:
 - WaitForMultipleObject() ->
MAXIMUM_WAIT_OBJECTS = 64
 - Constantly trying to figure out how to react base on which handle was signaled. (WaitForMultipleObject())
- Solution : APC
 - "Extended I/O" with completion routines
 - Using the "Ex(extended)" versions of ReadFile(), WriteFile() ([Read FileEx\(\)](#), [Write FileEx\(\)...](#))
 - Take an additional parameter that specifies a callback routine (I/O completion routine)
 - The system calls the callback routine when an overlapped I/O operation finishes.

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-30

ReadFileEx

- **BOOL ReadFileEx**(
 HANDLE *hFile*, // handle to file
 LPVOID *lpBuffer*, // data buffer
 DWORD *nNumberOfBytesToRead*,
 // number of bytes to read
 LPOVERLAPPED *lpOverlapped*, // offset
 LPOVERLAPPED_COMPLETION_ROUTINE
 lpCompletionRoutine // completion routine);
- **ReadFileEx()** designed only for asynchronous operation
- **hEvent** of the **OVERLAPPED** structure are ignored

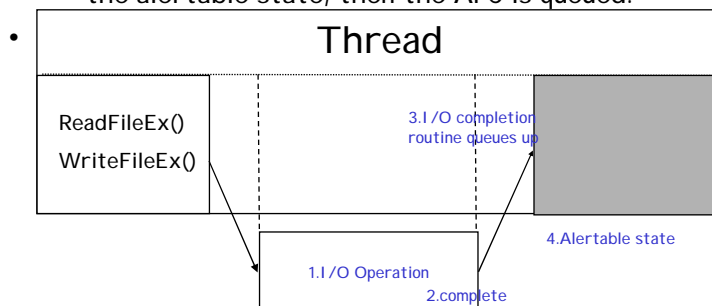
2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-31

Alertable State

- **APCs can only be called when the application is in an alertable state.**
 - `SleepEx()`, `WaitForSingleObjectEx()`, `WaitForMultipleObjectEx()`, `MsgWaitForMultipleObjectEx()`, `SignalObjectAndWait()`
- **Each thread has its own APC Queue**
 - If I/O operation is completed and the thread is not in the alertable state, then the APC is queued.



2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-32

6.5 Asynchronous Procedure Calls(APCs)

- VOID WINAPI [FileIoCompletionRoutine](#)(
 DWORD dwErrorCode,
 DWORD dwNumberOfBytesTransferred,
 LPOVERLAPPED lpOverlapped
)

6.5 Asynchronous Procedure Calls(APCs)

- Step:
 1. CreateEvent()
 2. CreateFile()
 3. ReadFileEx()
 4. WaitForSingleObjectEx()
 5. FileIoCompletionRoutine()
- Let's See Example (Listing 6-3)
- ?? How do I use a not static C++ member function as an I/O completion routine?
 - (This is similar to the problem of starting a thread in a C++ member function)

WaitForSingleObjectEx

- The **WaitForSingleObjectEx** function returns when one of the following occurs:
 - The specified object is in the signaled state.
 - An I/O completion routine or asynchronous procedure call (APC) is queued to the thread.
 - The time-out interval elapses.
- **DWORD WaitForSingleObjectEx(HANDLE *hHandle*, // handle to object
DWORD *dwMilliseconds*, // time-out interval
BOOL *bAlertable* // alertable option);**

WaitForSingleObjectEx

- *bAlertable*
 - The wait function returns and the completion routine is called only if *bAlertable* is TRUE, and the calling thread is the thread that initiated the read or write operation.
- Return Values

Value	Meaning
WAIT_OBJECT_0	The state of the specified object is signaled.
WAIT_IO_COMPLETION	The wait was ended by one or more user-mode asynchronous procedure calls (APC) queued to the thread.
WAIT_TIMEOUT	The time-out interval elapsed, and the object's state is nonsignaled.
WAIT_FAILED	The function fails. To get extended error information, call GetLastError .
WAIT_ABANDONED	Mutex

IOBYAPC.C – main()

```
// Open the file for overlapped reads
ghFile = CreateFile( szPath,
    GENERIC_READ,
    FILE_SHARE_READ|FILE_SHARE_WRITE,
    NULL,
    OPEN_EXISTING,
    FILE_FLAG_OVERLAPPED,
    NULL
);
```

Create file handle for overlapped operation

```
// Queue up a few requests
for (i=0; i<MAX_REQUESTS; i++)
{
    // Read some bytes every few K
    QueueRequest(i, i*16384, READ_SIZE);
}

printf("QUEUED!\n");
```

Read READ_SIZE bytes from i*6384 position for MAX_REQUEST times

```
// Wait for all the operations to complete.
for (;;)
{
    DWORD rc;
```

```
rc = WaitForSingleObjectEx(ghEvent, INFINITE, TRUE);
```

Enter alertable state

```
if (rc == WAIT_OBJECT_0)
    break;
```

If the event object is signaled, then go out of the loop

```
MTVERIFY(rc == WAIT_IO_COMPLETION);
```

If the wait was ended by APC queued by the thread, then continue the loop

```
}
CloseHandle(ghFile);
return EXIT_SUCCESS;
}
```

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-37

IOBYAPC.C – QueueRequest()

//The only difference is ReadFileEx()

```
int QueueRequest(int nIndex, DWORD dwLocation, DWORD
    dwAmount)
{
    int i;
    BOOL rc;
    DWORD err;

    gOverlapped[nIndex].hEvent = (HANDLE)nIndex;
    gOverlapped[nIndex].Offset = dwLocation;

    for (i=0; i<MAX_TRY_COUNT; i++)
    {
        rc = ReadFileEx(
            ghFile,
            gBuffers[nIndex],
            dwAmount,
            &gOverlapped[nIndex],
            FileIOCompletionRoutine
        );

        // Handle success
        if (rc)
        {
            // asynchronous i/o is still in progress
            printf("Read #%d queued for overlapped I/O.\n", nIndex);
            return TRUE;
        }
    }
}
```

```
err = GetLastError();
```

```
// Handle recoverable error
if (err == ERROR_INVALID_USER_BUFFER ||
    err == ERROR_NOT_ENOUGH_QUOTA ||
    err == ERROR_NOT_ENOUGH_MEMORY)
{
    Sleep(50); // Wait around and try later
    continue;
}
```

```
// Give up on fatal error.
break;
}
```

```
printf("ReadFileEx failed.\n");
return -1;
}
```

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-38

IOBYAPC.C – FileIOCompletionRoutine()

```
VOID WINAPI FileIOCompletionRoutine(  
    DWORD dwErrorCode, // completion code  
    DWORD dwNumberOfBytesTransferred, // number of bytes transferred  
    LPOVERLAPPED lpOverlapped // pointer to structure with I/O information  
)  
{  
    // The event handle is really the user defined data  
    int nIndex = (int)(lpOverlapped->hEvent);  
    printf("Read #%d returned %d. %d bytes were read.\n",  
        nIndex,  
        dwErrorCode,  
        dwNumberOfBytesTransferred);  
  
    if (++nCompletionCount == MAX_REQUESTS)  
        SetEvent(ghEvent); // Cause the wait to terminate  
}
```

If this is the last APC queued by the user,
then set the event object.

6.6 Drawbacks of Overlapped I/O with Files

- Windows NT decides whether or not to queue an operation based on the size of the request. (64k)
(ex) transfer rate 5MB/sec
 8KB -> block : 10ms (:seek the head)
 -> transfer : 1.8ms
 Nowadays : very different (should be 311k , 360k)
- In a Web server, using overlapped I/O would decrease overall performance.
 - -> bypass the virtual memory manager
 - -> go directly to the file system
(CreateFile() with FILE_FLAG_NO_BUFFERING)

6.6 Drawbacks of Overlapped I/O with Files

IBM Deskstar 75GXP and Deskstar 40GV at a glance		
Model	Deskstar 75GXP	Deskstar 40GV
	DTLA-307075/307060f 307045/307030f 307020/307015	DTLA-305040/305030f 305020
Configuration		
Interface	ATA	ATA
Capacity (GB)	75/60/45/30/20/15	40/30/20
Sector size (bytes)	512	512
Recording zone	15	15
User cylinders (physical)	27,724	34,326
Data heads (physical)	10/8/6/4/3/2	4/3/2
Data disks	5/4/3/2/2/1	2/2/1
Max. areal density (Gbits/sq. inch)	11.0	14.5
Max. recording density (BPI)	391,000	415,000
Track density (TPI)	28,350	35,000
Performance		
Data buffer	2 MB ²	512 KB ²
Rotational speed (RPM)	7200	5400
Latency (average ms)	4.17	5.56
Media transfer rate (max MBits/sec)	444	372
Interface transfer rate (max MB/sec)	100	100
Sustained data rate (MB/sec)	37	32
Seek time ³ (read typical)		
Average (ms)	8.5	9.5
Track-to-track (ms)	1.2	1.6
Full-track (ms)	15.0	16.0

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-41

6.7 I/O Completion Ports

- Problems of APCs with the overlapped I/O
 - Several I/O APIs doesn't supports APCs.
 - Only the thread that started the overlapped request can service the callback.
- Solution : In NT 3.5~, **I/O Completion ports**
 - Completion ports solve all of the problems we have seen so far
 - There is no limit to the number of handles.
 - It allows one thread to queue a request and another thread to service it.
 - implicitly support scalable architectures.

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-42

6.7 I/O Completion Ports

- **Server Threading Models**
 - There are three basic ways of deciding how many threads to use in a server
 - One Single Thread
 - One Single per Client
 - One Active Thread per Processor

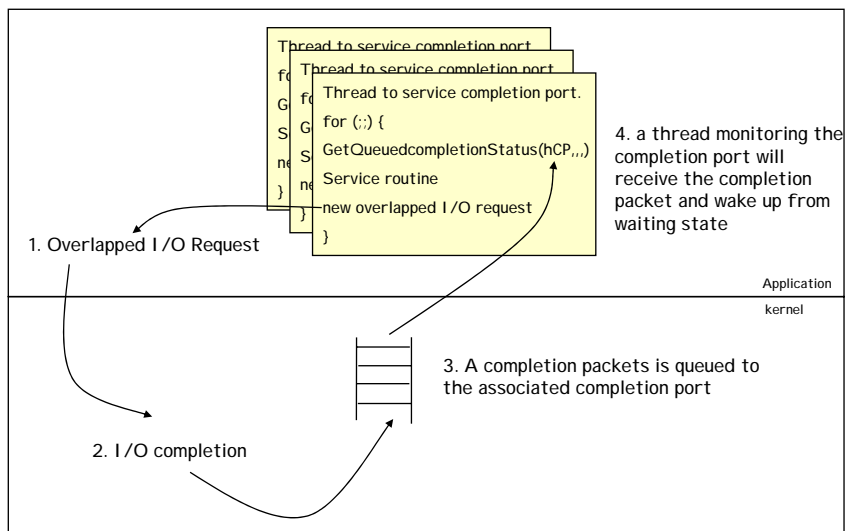
=> Use I/O Completion Ports !
 - **I/O Completion Ports** is a very special kind of **kernel object** that coordinates how a pool of threads services overlapped request, even across multiple processors.
- **What completion port do??**
 - The way an I/O completion port works is radically different from any of the others.

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-43

Operating Scenario



2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-44

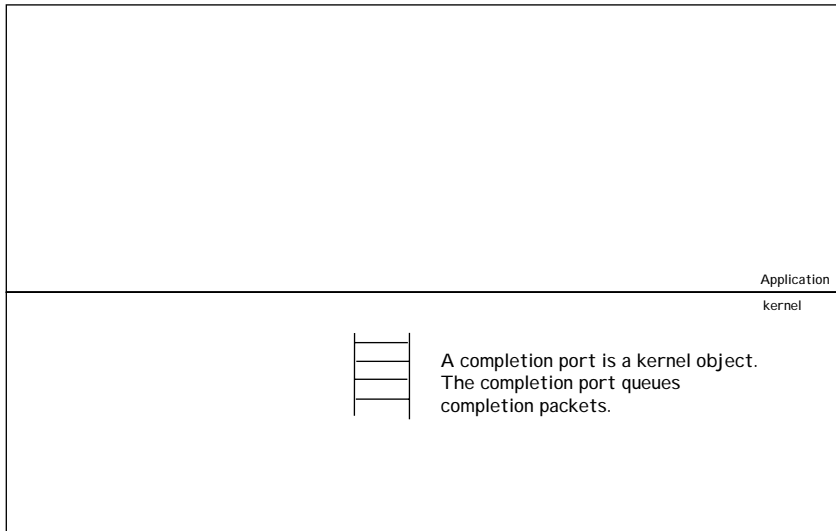
I/O Completion Port: Overview

- I/O completion ports are used with overlapped I/O.
 - The I/O should be created with `FILE_FLAG_OVERLAPPED` flag.
- I/O completion port is a kernel object
 - Use a handle to access the completion port.
- The `CreateIoCompletionPort` function associates an I/O completion port with one or more file handles.
- When an overlapped I/O operation started on a file handle associated with a completion port is completed, an I/O completion packet is queued to the port.
- This can be used to combine the synchronization point for multiple file handles into a single completion port object.
- A thread uses the `GetQueuedCompletionStatus` function to wait for a completion packet to be queued to the completion port, rather than waiting directly for the asynchronous I/O to complete.

Setup Procedure

1. Create the I/O Completion port.
2. Associate file handle with it.
3. Create a pool of threads.
4. Make each thread wait on the completion port.
5. Start issuing overlapped I/O requests with the file handles.

1. Creating a Completion Port



2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-47

1. Creating an I/O Completion Port

```
HANDLE CreateIoCompletionPort(  
    HANDLE    FileHandle,  
    HANDLE    ExistingCompletionPort,  
    DWORD     CompletionKey,  
    DWORD     NumberOfConcurrentThreads  
)
```

Return value :

succeed -> return a handle to the I/O completion port

Fail: Null -> GetLastError()

Note

For creating a completion port

FileHandle : INVALID_HANDLE_VALUE

ExistingCompletionPort : NULL

2005-11-16

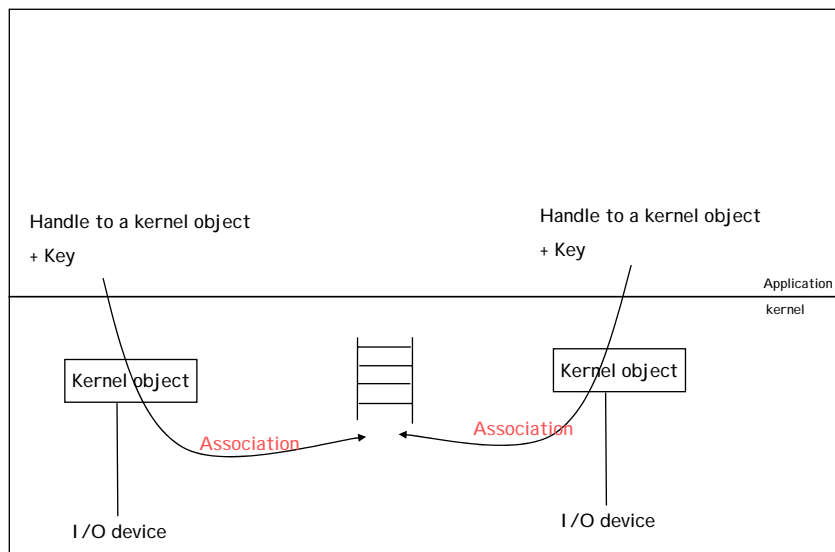
Seong Jong Choi

Multithreading_CHAPTER_6-48

1. Creating an I/O Completion Port

```
ghCompletionPort = CreateIoCompletionPort(  
    INVALID_HANDLE_VALUE,  
    NULL, // No prior port  
    0,    // No key  
    0     // Use default # of threads  
);  
if (ghCompletionPort == NULL)  
    FatalError("CreateIoCompletionPort() failed");
```

2. Associating File Handles



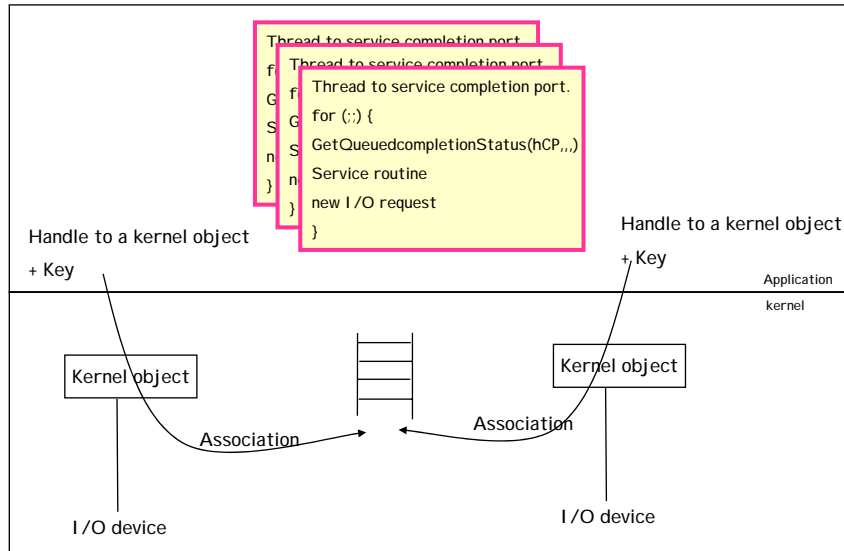
2. Associating File Handles

- Calling `CreateIoCompletionPort()` again once for each new file handle to be associated.
- 2nd argument : the handle returned by first calls

```
CreateIoCompletionPort(  
    (HANDLE) newsocket, //handle to the File  
    ghCompletionPort, //handle to the CP  
    (DWORD) pKey, // key pointer  
    0 // Use default # of threads  
);
```

- CompletionKey
 - Pointer to a variable that receives the completion key value associated with the file handle whose I/O operation has completed. A completion key is a per-file key that is specified in a call to `CreateIoCompletionPort`.

3. Creating a Pool of Threads



2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-53

3. Creating a Pool of Threads

Thread Currently running

- + Thread blocked (on disk I/O or a Wait...() call)
- + Thread waiting on the completion port

= Number of Threads in pool

- Number Threads to create = 2 * processor + 2

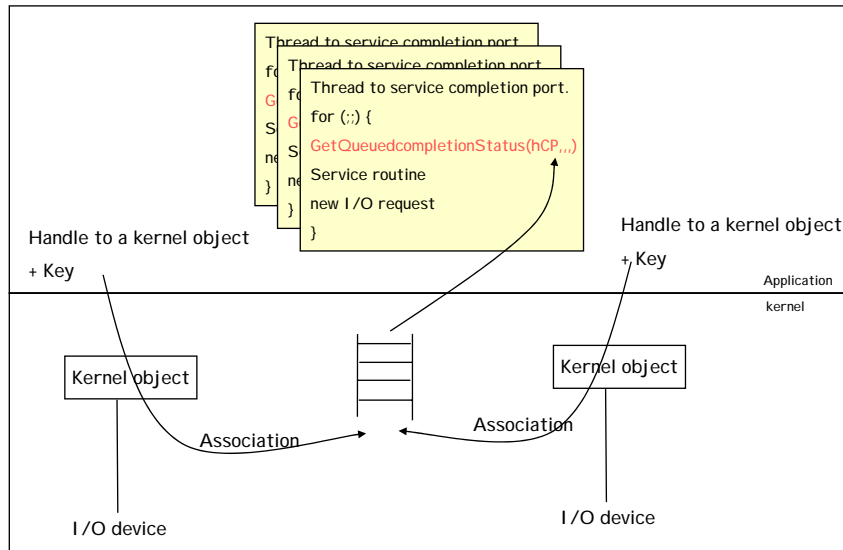
(chap 8, chap 10)

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-54

4. Make each thread wait on the completion port.



2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-55

4. Make each thread wait on the completion port.

- `GetQueuedCompletionStatus()`
 - The `GetQueuedCompletionStatus()` function attempts to dequeue an I/O completion packet from a specified I/O completion port.
 - If there is no completion packet queued, the function waits for a pending I/O operation associated with the completion port to complete.

```
BOOL GetQueuedCompletionStatus(  
    HANDLE CompletionPort,    // handle to completion port  
    LPDWORD lpNumberOfBytes,  // bytes transferred  
    PULONG_PTR lpCompletionKey, // file completion key  
    LPOVERLAPPED *lpOverlapped, // buffer  
    DWORD dwMilliseconds    // optional timeout value  
);
```

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-56

5. Issuing overlapped I/O request with a File Handle

Possible Functions

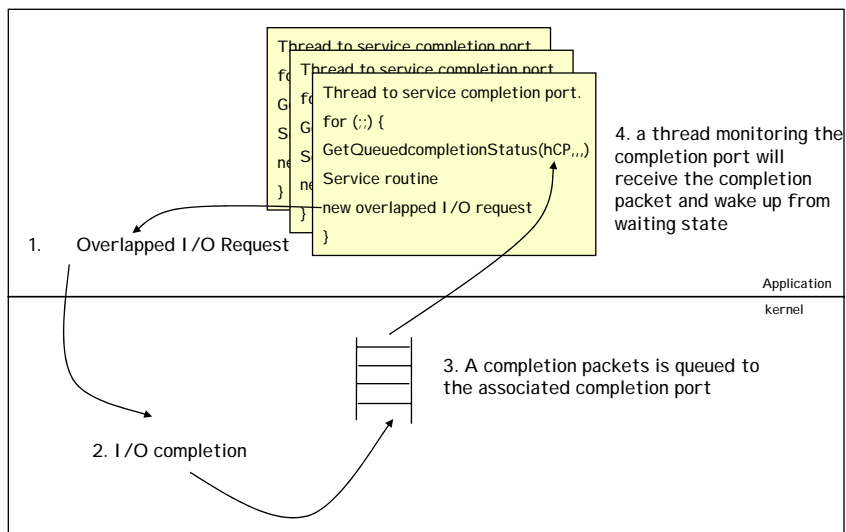
```
ConnectNamePipe()  
DeviceIoControl()  
LockFileEx()  
ReadFile()  
TransactNamePipe()  
WaitCommEvent()  
WriteFile()
```

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-57

Operating Scenario



2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-58

I/O Completion Port: Overview

- I/O completion ports are used with overlapped I/O.
 - The I/O should be created with `FILE_FLAG_OVERLAPPED` flag.
- I/O completion port is a kernel object
 - Use a handle to access the completion port.
- The `CreateIoCompletionPort` function associates an I/O completion port with one or more file handles.
- When an overlapped I/O operation started on a file handle associated with a completion port is completed, an I/O completion packet is queued to the port.
- This can be used to combine the synchronization point for multiple file handles into a single completion port object.
- A thread uses the `GetQueuedCompletionStatus` function to wait for a completion packet to be queued to the completion port, rather than waiting directly for the asynchronous I/O to complete.

Disabling Completion Packets

- You do not want the I/O completion port to be notified when the operation finishes.
- Use the signaled event object mechanism.

```
OVERLAPPED Overlap;  
HANDLE hFile;  
Char      buffer;  
DWORD    dwBytesWritten;  
Memset(&overlap, 0, sizeof(OVERLAPPED));  
overlap.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);  
overlap.hEvent = (HANDLE)((DWORD)hEvent | 0x1);  
WriteFile(hFile, buffer, 128, dwBytesWritten, &overlap);
```

6.8 Using Overlapped I/O with Sockets

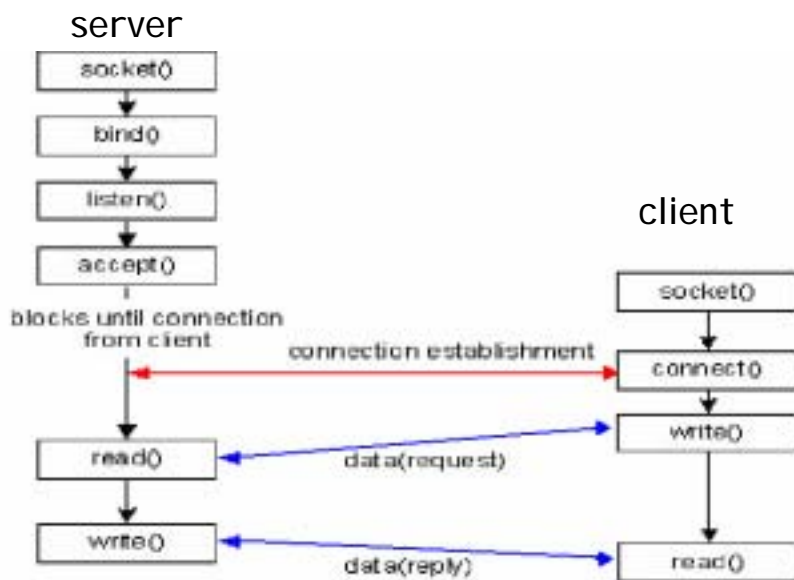
- The sample ECHO on the CD-ROM demonstrates using an I/O completion port to do the equivalent of standard TCP port 7, which echoes back every thing that is written to it.
- ECHOSRV is a server that listens on arbitrary TCP port
- ECHOCLI is the client that takes whatever you type, send ot to the server, and then print out the response
- -Completion ports are the only way to obtain high throughput on a Windows NT machine talking to the network.
 - Windows NT supports the *select()* call that is the standard way of supporting multiple simultaneous connections
 - But it has a problem with high performance.

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-61

6.8 Using Overlapped I/O with Sockets



2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-62

6.8 Using Overlapped I/O with Sockets

- Example (Listing 6-4)

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-63

Echosrv-main()

```
int main(int argc, char *argv[])
{
    ...
    /* Create/bind/listen socket
ghCompletionPort = CreateIoCompletionPort(
    INVALID_HANDLE_VALUE,
    NULL, // No prior port
    0, // No key
    0 // Use default # of threads
);
if (ghCompletionPort == NULL)
    FatalError("CreateIoCompletionPort() failed");
CreateWorkerThreads(ghCompletionPort);

// Loop forever accepting requests new connections
// and starting reading from them.
for (;;)
{
    struct ContextKey *pKey;

    clientAddressLength = sizeof(clientAddress);
    newsocket = accept(listener,
        (struct sockaddr *)&clientAddress,
        &clientAddressLength);

    // Create a context key and initialize it.
    pKey = calloc(1, sizeof(struct ContextKey));
    pKey->sock = newsocket;
    pKey->ovOut.hEvent = CreateEvent(NULL, TRUE, FALSE,
        NULL);

    // disable completion packet for write operation
    pKey->ovOut.hEvent = (HANDLE)((DWORD)pKey->
        >ovOut.hEvent | 0x1);

    // Associate the socket with the completion port
    CreateIoCompletionPort(
        (HANDLE)newsocket,
        ghCompletionPort,
        (DWORD)pKey, // No key
        0 // Use default # of threads
    );

    // Kick off the first read
    IssueRead(pKey);

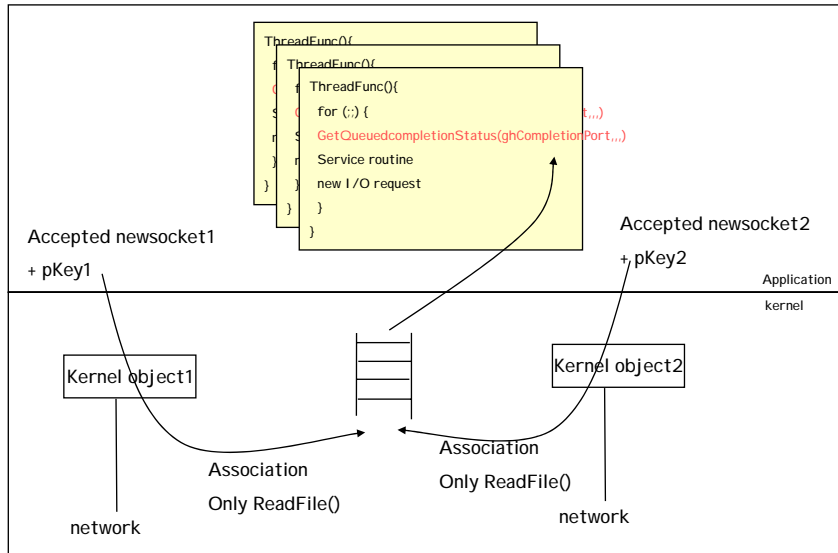
    return 0;
}
```

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-64

I/O Completion Port Setup with 2 clients



2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-65

Echosrv-IssueRead()

```
/*
 * Call ReadFile to start an overlapped request
 * on a socket. Make sure we handle errors
 * that are recoverable.
 */
void IssueRead(struct ContextKey *pCntx)
{
    int i = 0;
    BOOL bResult;
    int err;
    int numRead;
```

```
while (++i)
{
    // Request a single character
    bResult = ReadFile(
        (HANDLE)pCntx->sock,
        pCntx->InBuffer,
        1,
        &numRead,
        &pCntx->ovl);
}
```

```
// It succeeded immediately, but do not process it
// here, wait for the completion packet.
if (bResult)
    return;
```

```
err = GetLastError();
// This is what we want to happen, it's not an error
if (err == ERROR_IO_PENDING)
    return;
```

```
// Handle recoverable error
if (err == ERROR_INVALID_USER_BUFFER ||
    err == ERROR_NOT_ENOUGH_QUOTA ||
    err == ERROR_NOT_ENOUGH_MEMORY)
{
    if (i == 5) // I just picked a number
    {
        Sleep(50); // Wait around and try later
        continue;
    }

    FatalError("IssueRead - System ran out of non-paged
    space");
}
break;
```

```
}
printf(stderr, "IssueRead - ReadFile failed.\n");
}
```

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-66

Echosrv-ThreadFunc()

```
// Each worker thread starts here.
DWORD WINAPI ThreadFunc(LPVOID pVoid)
{
    BOOL bResult;
    DWORD dwNumRead;
    struct ContextKey *pCntx;
    LPOVERLAPPED lpOverlapped;

    UNREFERENCED_PARAMETER(pVoid);

    // Loop forever on getting packets from
    // the I/O completion port.
    for (;;)
    {
        bResult = GetQueuedCompletionStatus(
            ghCompletionPort,
            &dwNumRead,
            &(DWORD)pCntx,
            &lpOverlapped,
            INFINITE
        );

        if (bResult == FALSE
            && lpOverlapped == NULL)
        {
            FatalError(
                "ThreadFunc - Illegal call to
                GetQueuedCompletionStatus");
        }

        else if (bResult == FALSE
            && lpOverlapped != NULL)
        {
            // This happens occasionally instead of
            // end-of-file. Not sure why.
            closesocket(pCntx->sock);
            free(pCntx);
            fprintf(stderr,
                "ThreadFunc - I/O operation failed\n");
        }

        else if (dwNumRead == 0)
        {
            closesocket(pCntx->sock);
            free(pCntx);
            fprintf(stderr, "ThreadFunc - End of file.\n");
        }

        // Got a valid data block
        // Save the data to our buffer and write it
        // all back out (echo it) if we have see a '\n'
        else
        {
            // Figure out where in the buffer to save the character
            char *pch = &pCntx->OutBuffer[pCntx->nOutBufIndex++];
            *pch++ = pCntx->InBuffer[0];
            *pch = '\0'; // For debugging, WriteFile doesn't care
            if (pCntx->nInBuffer[0] == '\n')
            {
                WriteFile(
                    (HANDLE)(pCntx->sock),
                    pCntx->OutBuffer,
                    pCntx->nOutBufIndex,
                    &pCntx->dwWritten,
                    &pCntx->ovOut
                );
                pCntx->nOutBufIndex = 0;
                fprintf(stderr, "Echo on socket %x.\n", pCntx->sock);
            }

            // Start a new read
            IssueRead(pCntx);
        }
    }

    return 0;
}
```

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-67

6.9 Summary

- This chapter has provided a whirlwind tour of overlapped I/O, a technique for doing I/O asynchronously that often avoids the need for using multiple threads.
- Overlapped I/O can be done using signaled file handles, signaled event objects, asynchronous procedure calls (APCs), and I/O completion ports.
- I/O completion ports are very important because they are the preferred mechanism for creating high-performance servers that are easily scalable.

2005-11-16

Seong Jong Choi

Multithreading_CHAPTER_6-68