Comments on the Waitable Timer Example

There are four combinations based on timer type and whether you wait on the handle or use a completion routine. Program 14–3 illustrates using a completion routine and a synchronization timer. The four combinations can be tested using the version of `TimeBeep.c` in the *Examples* file by changing some comments.

Caution: The beep sound may be annoying, so you might want to test this program without anyone else nearby or adjust the frequency and duration.

Threadpool Timers

Alternatively, you can use a different type of timer, specifying that the timer callback function is to be executed within a thread pool (see Chapter 9). `TimeBeep_TPT` is a simple modification to `TimeBeep`, and it shows how to use `CreateThreadpoolTimer` and `SetThreadpoolTimer`. The new program requires a `FILETIME` structure for the timer due time, whereas the waitable timer used a `LARGE_INTEGER`.

I/O Completion Ports

I/O completion ports combine features of both overlapped I/O and independent threads and are most useful in server programs. To see the requirement for this, consider the servers that we built in Chapters 11 and 12 (and converted to Windows Services in Chapter 13), where each client is supported by a distinct worker thread associated with a socket or named pipe instance. This solution works well when the number of clients is not large.

Consider what would happen, however, if there were 1,000 clients. The current model would then require 1,000 threads, each with a substantial amount of virtual memory space. For example, by default, each thread will consume 1MB of stack space, so 1,000 threads would require 1GB of virtual address space, and thread context switches could increase page fault delays.[1] Furthermore, the threads would contend for shared resources both in the executive and in the process, and the timing data in Chapter 9 showed the performance degradation that can result. Therefore, there is a requirement to allow a small pool of worker threads to serve a large number of clients. Chapter 9 used an NT6 thread pool to address this same problem.

I/O completion ports provide a solution on all Windows versions by allowing you to create a limited number of server threads in a thread pool while having a very large number of named pipe handles (or sockets), each associated with a dif-

[1]This problem is less severe, but should not be ignored, on systems with large amounts of memory.

ferent client. Handles are not paired with individual worker server threads; rather, a server thread can process data on any handle that has available data.

An I/O completion port, then, is a set of overlapped handles, and threads wait on the port. When a read or write on one of the handles is complete, one thread is awakened and given the data and the results of the I/O operation. The thread can then process the data and wait on the port again.

The first task is to create an I/O completion port and add overlapped handles to the port.

Managing I/O Completion Ports

A single function, `CreateIoCompletionPort`, both creates the port and adds handles. Since this one function must perform two tasks, the parameter usage is correspondingly complex.

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

An I/O completion port is a collection of file handles opened in `OVERLAPPED` mode. `FileHandle` is an overlapped handle to add to the port. If the value is `INVALID_HANDLE_VALUE`, a new I/O completion port is created and returned by the function. The next parameter, `ExistingCompletionPort`, must be `NULL` in this case.

`ExistingCompletionPort` is the port created on the first call, and it indicates the port to which the handle in the first parameter is to be added. The function also returns the port handle when the function is successful; `NULL` indicates failure.

`CompletionKey` specifies the key that will be included in the completion packet for `FileHandle`. The key could be a pointer to a structure containing information such as an operation type, a handle, and a pointer to the data buffer. Alternatively, the key could be an index to a table of structures, although this is less flexible.

`NumberOfConcurrentThreads` indicates the maximum number of threads allowed to execute concurrently. Any threads in excess of this number that are waiting on the port will remain blocked even if there is a handle with available data. If this parameter is `0`, the number of processors in the system is the limit. The value is ignored except when `ExistingCompletionPort` is `NULL` (that is, the port is created, not when handles are added).

An unlimited number of overlapped handles can be associated with an I/O completion port. Call `CreateIoCompletionPort` initially to create the port and to specify the maximum number of threads. Call the function again for every overlapped handle that is to be associated with the port. There is no way to remove a handle from a completion port; the handle and completion port are associated permanently.

The handles associated with a port should not be used with `ReadFileEx` or `WriteFileEx` functions. The Microsoft documentation suggests that the files or other objects not be shared using other open handles.

Waiting on an I/O Completion Port

Use `ReadFile` and `WriteFile`, along with overlapped structures (no event handle is necessary), to perform I/O on the handles associated with a port. The I/O operation is then queued on the completion port.

A thread waits for a queued overlapped completion not by waiting on an event but by calling `GetQueuedCompletionStatus`, specifying the completion port. Upon completion, the function returns a key that was specified when the handle (the one whose operation has completed) was initially added to the port with `CreateIoCompletionPort`. This key can specify the identity of the actual handle for the completed operation and other information associated with the I/O operation.

Notice that the Windows thread that initiated the read or write is not necessarily the thread that will receive the completion notification; any waiting thread can receive completion notification. Therefore, the receiving thread can identify the handle of the completed operation from the completion key.

Never hold a lock (mutex, `CRITICAL_SECTION`, etc.) when you call `GetQueuedCompletionStatus`, because the thread that releases the lock is probably not the some thread that acquired it. Owning the lock would not be a good idea in any case as there is an indefinite wait before the completion notification.

There is also a time-out associated with the wait.

```
BOOL GetQueuedCompletionStatus (
   HANDLE CompletionPort,
   LPDWORD lpNumberOfBytesTransferred,
   PULONG_PTR lpCompletionKey,
   LPOVERLAPPED *lpOverlapped,
   DWORD dwMilliseconds);
```

It is sometimes convenient (as in an additional example, `cciMTCP`, in the *Examples* file) to have the operation not be queued on the I/O completion port,

making the operation synchronous. In such a case, a thread can wait on the overlapped event. In order to specify that an overlapped operation should *not* be queued on the completion port, you must set the low-order bit in the overlapped structure's event handle; then you can wait on the event for that specific operation. This is an interesting design, but MSDN does document it, although not prominently.

Posting to an I/O Completion Port

A thread can post a completion event, with a key, to a port to satisfy an outstanding call to `GetQueuedCompletionStatus`. The `PostQueuedCompletionStatus` function supplies all the required information.

```
BOOL PostQueuedCompletionStatus (
   HANDLE CompletionPort,
   DWORD dwNumberOfBytesTransferred,
   ULONG_PTR dwCompletionKey,
   LPOVERLAPPED lpOverlapped);
```

One common technique is to provide a bogus key value, such as `-1`, to wake up waiting threads, even though no operation has completed. Waiting threads should test for bogus key values, and this technique can be used, for example, to signal a thread to shut down.

Alternatives to I/O Completion Ports

Chapter 9 showed how a semaphore can be used to limit the number of ready threads, and this technique is effective in maintaining throughput when many threads compete for limited resources.

We could use the same technique with `serverSK` (Program 12–2) and `serverNP` (Program 11–3). All that is required is to wait on the semaphore after the read request completes, perform the request, create the response, and release the semaphore before writing the response. This solution is much simpler than the I/O completion port example in the next section. One problem with this solution is that there may be a large number of threads, each with its own stack space, which will consume virtual memory. The problem can be partly alleviated by carefully measuring the amount of stack space required. Exercise 14–7 involves experimentation with this alternative solution, and there is an example implementation in the *Examples* file. I/O completion ports also have the advantage that the scheduler posts

the completion to the thread most recently executed, as that thread's memory is most likely still in the cache or at least does not need to be paged in.

There is yet another possibility when creating scalable servers. A limited number of worker threads can take work item packets from a queue (see Chapter 10). The incoming work items can be placed in the queue by one or more boss threads, as in Program 10–5.

Example: A Server Using I/O Completion Ports

`serverCP` (Program 14–4) modifies `serverNP` (Program 11–3) to use I/O completion ports. This server creates a small server thread pool and a larger pool of overlapped pipe handles along with a completion key for each handle. The overlapped handles are added to the completion port and a `ConnectNamedPipe` call is issued. The server threads wait for completions associated with both client connections and read operations. After a read is detected, the associated client request is processed and returned to the client (`clientNP` from Chapter 11).

`serverCP`'s design prevents server threads from blocking during I/O operations or request processing (through an external process). Each client pipe goes through a set of states (see the `enum CP_CLIENT_PIPE_STATE` type in the listing), and different server threads may process the pipe through stages of the state cycle. The states, which are maintained in a per-pipe `CP_KEY` structure, proceed as follows:

- `connected` — The pipe is connected with a server thread.
- `requestRead` — The server thread reads a request from the client and starts the process from a separate "compute" thread, which calls `PostQueuedCompletionStatus` when the process completes. The server thread does not block, since the process management is in the compute thread.
- `computed` — The server thread reads the first temporary file record with the response's first record, and the server thread then writes the record to the client.
- `responding` — The server thread sends additional response records, one at a time, returning to the `responding` state until the last response record is sent to the client.
- `respondLast` — The server thread sends a terminating empty record to the client.

The program listing does not show familiar functions such as the server mailslot broadcast thread function.

Program 14–4 `serverCP`: A Server Using a Completion Port

```
/*  Chapter 14. ServerCP.
 * Command line server. Named pipe version, COMPLETION PORT example
 *  Assume MAX_SERVER_TH <= 64 (because of WaitForMultipleObject) */

#include "Everything.h"
#include "ClientServer.h" /* Request, response message definitions */

typedef struct {    /* Argument to a server thread. */
   HANDLE hCompPort;/* Completion port handle. */
   DWORD threadNum;
} SERVER_THREAD_ARG;
typedef SERVER_THREAD_ARG *LPSERVER_THREAD_ARG;

enum CP_CLIENT_PIPE_STATE { connected, requestRead, computed,
                            responding, respondLast };
/* Argument structure for each named pipe instance */
typedef struct {/* Completion port keys refer to these structures */
      HANDLE hCompPort;
      HANDLE hNp;/* which represent outstanding ReadFile  */
      HANDLE hTempFile;
      FILE *tFp;/* Used by server process to hold result */
      TCHAR tmpFileName[MAX_PATH]; /* Temp file name for respnse. */
      REQUESTrequest;/* and ConnectNamedPipe operations */
      DWORD nBytes;
      enum CP_CLIENT_PIPE_STATE npState;
      LPOVERLAPPED pOverLap;
} CP_KEY;

OVERLAPPED overLap;
volatile static int shutDown = 0;
static DWORD WINAPI Server (LPSERVER_THREAD_ARG);
static DWORD WINAPI ServerBroadcast (LPLONG);
static BOOL  WINAPI Handler (DWORD);
static DWORD WINAPI ComputeThread (PVOID);

static CP_KEY Key[MAX_CLIENTS_CP];

_tmain (int argc, LPTSTR argv[])
{
   HANDLE hCompPort, hMonitor, hSrvrThread[MAX_CLIENTS];
   DWORD iNp, iTh;
   SECURITY_ATTRIBUTES tempFileSA = {sizeof (SECURITY_ATTRIBUTES),
                                     NULL, TRUE};
   SERVER_THREAD_ARG ThArgs[MAX_SERVER_TH]; /* MAX_SERVER_TH <= 64 */
   OVERLAPPED ov = {0};

   /* Console control handler to permit server shutDown */
```

```
SetConsoleCtrlHandler (Handler, TRUE);

/* Create a thread broadcast pipe name periodically. */
hMonitor = (HANDLE) _beginthreadex (NULL, 0, ServerBroadcast, NULL,
                    0, NULL);

hCompPort = CreateIoCompletionPort (INVALID_HANDLE_VALUE, NULL, 0,
                MAX_SERVER_TH);

/* Create an overlapped named pipe for every potential client, */
/* add to the completion port */
/* Assume that the maximum number of clients far exceeds */
/* the number of server threads*/
for (iNp = 0; iNp < MAX_CLIENTS_CP; iNp++) {
   memset (&Key[iNp], 0, sizeof(CP_KEY));
   Key[iNp].hCompPort = hCompPort;
   Key[iNp].hNp =  CreateNamedPipe ( SERVER_PIPE,
      PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED,
      PIPE_READMODE_MESSAGE | PIPE_TYPE_MESSAGE,
      MAX_CLIENTS_CP, 0, 0, INFINITE, &tempFileSA);
   GetTempFileName (_T("."), _T "CLP"), 0, Key[iNp].tmpFileName);
   Key[iNp].pOverLap = &overLap;
   /* Add the named pipe instance to the completion port */
   CreateIoCompletionPort (Key[iNp].hNp, hCompPort,
            (ULONG_PTR)&Key[iNp], 0);
   ConnectNamedPipe (Key[iNp].hNp, &ov);
   Key[iNp].npState = connected;
}

/* Create server worker threads and a temp file name for each.*/
for (iTh = 0; iTh < MAX_SERVER_TH; iTh++) {
   ThArgs[iTh].hCompPort = hCompPort;
   ThArgs[iTh].threadNum = iTh;
   hSrvrThread[iTh] = (HANDLE)_beginthreadex (NULL, 0, Server,
                       &ThArgs[iTh], 0, NULL);
}
_tprintf (_T("All server threads running.\n"));

/* Wait for all server threads (<= 64) to terminate. */
WaitForMultipleObjects (MAX_SERVER_TH, hSrvrThread,
            TRUE, INFINITE);
WaitForSingleObject (hMonitor, INFINITE);
_tprintf (_T ("Monitor and server threads have shut down.\n"));

CloseHandle (hMonitor);
for (iTh = 0; iTh < MAX_SERVER_TH; iTh++) {
   /* Close pipe handles */
   CloseHandle (hSrvrThread[iTh]);
}
```

```
    CloseHandle (hCompPort);

    return 0;
}

static DWORD WINAPI Server (LPSERVER_THREAD_ARG pThArg)
/* Server thread function. . */
{
    DWORD nXfer;
    BOOL disconnect = FALSE;
    CP_KEY *pKey;
    RESPONSE response;
    OVERLAPPED serverOv = {0}, * pOv = NULL;

    /* Main server thread loop to process completions */
    while (!disconnect && !shutDown) {
        if (!GetQueuedCompletionStatus (pThArg->hCompPort, &nXfer,
                (PULONG_PTR)&pKey, &pOv, INFINITE))
        {
            DWORD errCode = GetLastError();
            if (errCode == ERROR_OPERATION_ABORTED) continue;
            if (errCode != ERROR_MORE_DATA)
                ReportError (_T("GetQueuedCompletionStatus error."),
                    0, TRUE);
        }
        if (shutDown) continue;

        /* npState: connected, requestRead, ... */
        switch (pKey->npState) {
            case connected:
                {  /* A connection has completed, read a request */
                    /* Open temp results file for this connection. */

                    _tcscpy (pKey->request.record, _T(""));
                    pKey->request.rqLen = 0;
                    pKey->npState = requestRead;
                    disconnect = !ReadFile (pKey->hNp, &(pKey->request),
                            RQ_SIZE, &(pKey->nBytes), &serverOv)
                        && GetLastError() != ERROR_IO_PENDING;
                    continue;
                }
            case requestRead:
                { /* A read has completed. process the request */
                    /* Thread to process request asynchronously. */
                    /* This server is free to process other requests. */
                    HANDLE hComputeThread;
                    DWORD computeExitCode;
                    hComputeThread = (HANDLE)_beginthreadex (NULL, 0,
                            ComputeThread, pKey, 0, NULL);
                    if (NULL == hComputeThread) continue;
```

```
            WaitForSingleObject(hComputeThread, INFINITE);
            GetExitCodeThread (hComputeThread, &computeExitCode);
            CloseHandle (hComputeThread);

            pKey->npState = computed;
            if (computeExitCode != 0)
            {
                pKey->npState = respondLast;
            }

            PostQueuedCompletionStatus (pKey->hCompPort, 0,
                (ULONG_PTR)pKey, pKey->pOverLap);
            continue;
        }
    case computed:
        {
            /* Results are in the temp file */
            /* Respond a line at a time. It is convenient to use
               C library line-oriented routines at this point. */
            pKey->tFp = _tfopen (pKey->tmpFileName, _T ("r"));
            pKey->npState = responding;
            if (_fgetts (response.record, MAX_RQRS_LEN,
                      pKey->tFp) != NULL) {
                response.rsLen = strlen(response.record) + 1;
                disconnect = !WriteFile (pKey->hNp, &response,
                      response.rsLen + sizeof(response.rsLen),
                      &nXfer, &serverOv)
                   && GetLastError() != ERROR_IO_PENDING;
            } else {
                /* Bad read; post completion; go to next state */
                pKey->npState = respondLast;
                PostQueuedCompletionStatus (pKey->hCompPort, 0,
                      (ULONG_PTR)pKey, pKey->pOverLap);
            }
            continue;
        }
    case responding:
        {
            /* In the process of responding a record at a time */
            /* Continue in this state until no more records */
            if (_fgetts (response.record, MAX_RQRS_LEN,
                      pKey->tFp) != NULL) {
                response.rsLen = strlen(response.record) + 1;
                disconnect = !WriteFile (pKey->hNp, &response,
                      response.rsLen + sizeof(response.rsLen),
                      &nXfer, &serverOv)
                   && GetLastError() != ERROR_IO_PENDING;
            }
```

```
                else {
                    pKey->npState = respondLast;
                    PostQueuedCompletionStatus (pKey->hCompPort, 0,
                        (ULONG_PTR)pKey, pKey->pOverLap);
                }
                continue;
            }
        case respondLast:
            {
                /* Send end of response indicator. */
                /* Stay connected. */
                pKey->npState = connected;
                _tcscpy (response.record, _T(""));
                response.rsLen = 0;
                disconnect = !WriteFile (pKey->hNp, &response,
                        sizeof(response.rsLen), &nXfer, &serverOv)
                    && GetLastError() != ERROR_IO_PENDING;
                continue;
            }
        default:
            {
                /* No recovery attempted (a good exercise!) */
                return 1;
            }
    }

    FlushFileBuffers (pKey->hNp);
    DisconnectNamedPipe (pKey->hNp);
    if (disconnect) {
        ConnectNamedPipe (pKey->hNp, &serverOv);
        pKey->npState = connected;
    } else {
        _tprintf (_T("Thread %d shutting down\n"),
                pThArg->threadNum);
        /* End of command processing. Free resources; thread exit */
        DeleteFile (pKey->tmpFileName);
        return 0;
    }
  }
  return 0;
}

static DWORD WINAPI ComputeThread (PVOID pArg)
{
  PROCESS_INFORMATION procInfo;
  STARTUPINFO startInfo;
  CP_KEY *pKey = (CP_KEY *)pArg;
  SECURITY_ATTRIBUTES tempSA =
        {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};
  GetStartupInfo (&startInfo);
```

```
    /* Open temp file, erasing previous contents */
    pKey->hTempFile =
        CreateFile (pKey->tmpFileName, GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE, &tempSA,
        CREATE_ALWAYS, FILE_ATTRIBUTE_TEMPORARY, NULL);

    if (pKey->hTempFile != INVALID_HANDLE_VALUE) {
        startInfo.hStdOutput = pKey->hTempFile;
        startInfo.hStdError = pKey->hTempFile;
        startInfo.hStdInput = GetStdHandle (STD_INPUT_HANDLE);
        startInfo.dwFlags = STARTF_USESTDHANDLES;
        CreateProcess (NULL, pKey->request.record, NULL, NULL,
              TRUE, /* Inherit handles. */
              0, NULL, NULL, &startInfo, &procInfo);

        /* Server process is running */
        CloseHandle (procInfo.hThread);
        WaitForSingleObject (procInfo.hProcess, INFINITE);
        CloseHandle (procInfo.hProcess);
        CloseHandle(pKey->hTempFile);
    } else { /* To do: Come here if CreateProcess fails. */
        ReportError (_T("Compute thread failed."));
        return 1;
    }

    return 0;
}

static DWORD WINAPI ServerBroadcast (LPLONG pNull)
{
    MS_MESSAGE MsNotify;
    DWORD nXfer, iNp;
    HANDLE hMsFile;

    /* Open the mailslot for the MS "client" writer. */
    while (!shutDown) { /* Run as long as there are server threads */
        /* Wait for another client to open a mailslot. */
        Sleep (CS_TIMEOUT);
        hMsFile = CreateFile (MS_CLTNAME, GENERIC_WRITE | GENERIC_READ,
              FILE_SHARE_READ | FILE_SHARE_WRITE,
              NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
        if (hMsFile == INVALID_HANDLE_VALUE) continue;

        /* Send out the message to the mailslot. */

        MsNotify.msStatus = 0;
        MsNotify.msUtilization = 0;
        _tcscpy (MsNotify.msName, SERVER_PIPE);
        if (!WriteFile (hMsFile, &MsNotify, MSM_SIZE, &nXfer, NULL))
            ReportError (_T ("Server MS Write error."), 13, TRUE);
```

```
        CloseHandle (hMsFile);
    }

    _tprintf (_T("Cancel all outstanding I/O operations.\n"));
    for (iNp = 0; iNp < MAX_CLIENTS_CP; iNp++) {
        CancelIoEx (Key[iNp].hNp, NULL);
    }
    _tprintf (_T ("Shuting down monitor thread.\n"));

    _endthreadex (0);
    return 0;
}

BOOL WINAPI Handler (DWORD CtrlEvent)
{
    /* Same as in serverNP, but it posts a completion */
}
```

Summary

Windows has three methods for performing asynchronous I/O; there are examples of all three, along with performance results, throughout the book to help you decide which to use on the basis of programming simplicity and performance.

Threads provide the most general and simplest technique. Each thread is responsible for a sequence of one or more sequential, blocking I/O operations. Furthermore, each thread should have its own file or pipe handle.

Overlapped I/O allows a single thread to perform asynchronous operations on a single file handle, but there must be an event handle, rather than a thread and file handle pair, for each operation. Wait specifically for each I/O operation to complete and then perform any required cleanup or sequencing operations.

Extended I/O, on the other hand, automatically invokes the completion code, and it does not require additional events.

The one indispensable advantage provided by overlapped I/O is the ability to create I/O completion ports as mentioned previously and illustrated by a program, `cciMTCP.c`, in the *Examples* file. A single server thread can serve multiple clients, which is important if there are thousands of clients; there would not be enough memory for the equivalent number of servers.

UNIX supports threads through Pthreads, as discussed previously.

System V UNIX limits asynchronous I/O to streams and cannot be used for file or pipe operations.