A simple application using I/O Completion Ports and WinSock - CodeProject

codeproject.com 2017년 8월 16일 업데이트됨

A simple application using I/O Completion Ports and WinSock

An article on implementing I/O completion ports using .

Introduction

The primary objective of this code submission is to provide source code which will demonstrate the use of IOCP using WinSock. This code submission tries to highlight the use of IOCP using a very easy to understand source code: the client and the server perform very simple operations; basically, they send and receive simple string messages. The IOCP client will allow you to stress test the server. You can send a message and provide input on how many times the message should be sent to the server.

Additionally, I have included *socket1.zip*, this code submission contains a one-to-one client and server implementation, and doesn't use IOCP. Similarly, *socket2.zip* contains a multi-threaded socket based server that creates a thread for every client connection, and is not a good design. I have included these additional codes so that the reader can compare these implementations with the IOCP implementation. This will provide additional insights to the understanding of the use of IOCP using WinSock.

Background

I was considering the use of IOCP in my current project. I found IOCP using WinSock to be a very useful, robust, and scalable mechanism. IOCP allows an application to use a pool of threads that are created to process asynchronous I/O requests. This prevents the application from creating one thread per client which can have severe performance issues (*socket2.zip* contains a one thread per client implementation).

Using the code

I have provided four zip files:

- *Socket1.zip* contains a simple one-to-one client and server implementation using WinSock. This is a pretty straightforward implementation; I won't be discussing this code.
- *Socket2.zip* contains a multi-threaded server, one thread per client; the client code

remains the same and is not discussed.

- *ServerIOCP.zip* contains a multi-threaded server that uses IOCP, and is the focus of this article.
- *ClientIOCP.zip* contains the IOCP client; the IOCP client can stress-test the IOCP server.
- *IOCPExecutables.zip* contains the IOCP client and server executables.

All of the code submissions are console based applications.

Multi-threaded server without IOCP

The implementation of this server can be found in *socket2.zip*.

Once a listening socket is created, a call is made to the `AcceptConnections()` function with the listening socket as the input parameter:

Hide Copy Code

```
//Make the socket a listening socket
if (SOCKET_ERROR == listen(ListenSocket,SOMAXCONN))
{
    closesocket(ListenSocket);
    printf("\nError occurred while listening.");
    goto error;
}
else
{
    printf("\nlisten() successful.");
}

//This function will take are of multiple clients using threads
AcceptConnections(ListenSocket);
```

The `AcceptConnections()` function will accept incoming client connections, and will spawn a thread for every new client connection:

Hide Shrink ▲ Copy Code

```
//This function will loop on while creating
//a new thread for each client connection
void AcceptConnections(SOCKET ListenSocket)
{
     sockaddr_in ClientAddress;
     int nClientLength = sizeof(ClientAddress);

     //Infinite, no graceful shutdown of server implemented,
     //preferably server should be implemented as a service
     //Events can also be used for graceful shutdown
     while (1)
     {
          //Accept remote connection attempt from the client
          SOCKET Socket = accept(ListenSocket,
                 (sockaddr*)&ClientAddress, &nClientLength);
```

```
            if (INVALID_SOCKET == Socket)
            {
                 printf("\nError occurred while accepting"
                        " socket: %ld.", WSAGetLastError());
            }

            //Display Client's IP
            printf("\nClient connected from: %s",
                   inet_ntoa(ClientAddress.sin_addr));

            DWORD nThreadID;

            //Spawn one thread for each client
            //connection, not a wise idea.
            //One should limit the number of threads
            //or use I/O completion port
            CreateThread(0, 0, AcceptHandler,
                   (void*)Socket, 0, &nThreadID);
      }
}
```

The `AcceptHandler()` function is a thread function. It will take an accepted socket as the input, and will perform client related I/O operations on it:

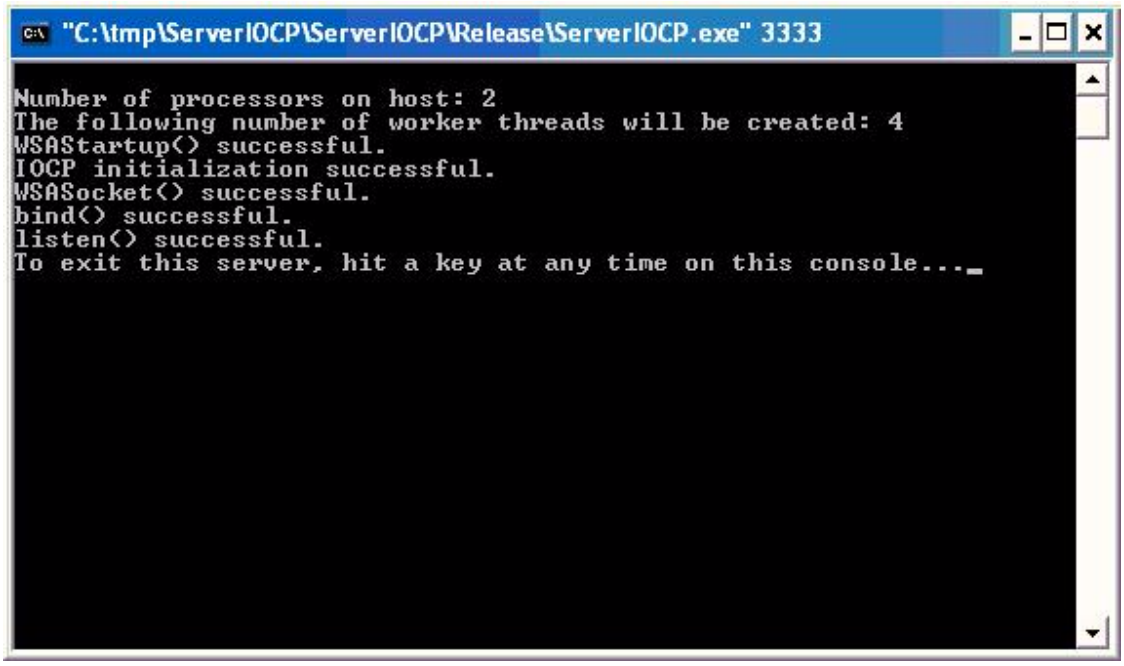
Hide Shrink ▲ Copy Code

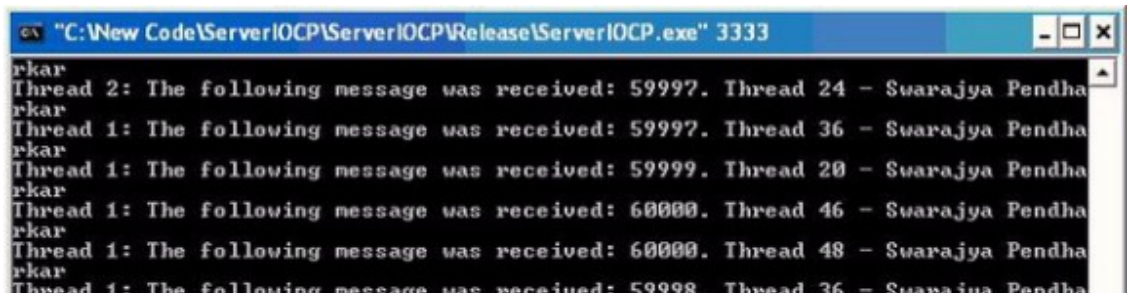
```
//Thread procedure one thread will be created for each client.
DWORD WINAPI AcceptHandler(void* Socket)
{
      SOCKET RemoteSocket = (SOCKET)Socket;

      char szBuffer[256];

      //Cleanup and Init with 0 the szBuffer
      ZeroMemory(szBuffer,256);

      int nBytesSent;
      int nBytesRecv;

      //Receive data from a connected or bound socket
      nBytesRecv = recv(RemoteSocket, szBuffer, 255, 0 );

      if (SOCKET_ERROR == nBytesRecv)
      {
            closesocket(RemoteSocket);
            printf("\nError occurred while receiving from socket.");
            return 1; //error
      }
      else
      {
            printf("\nrecv() successful.");
```

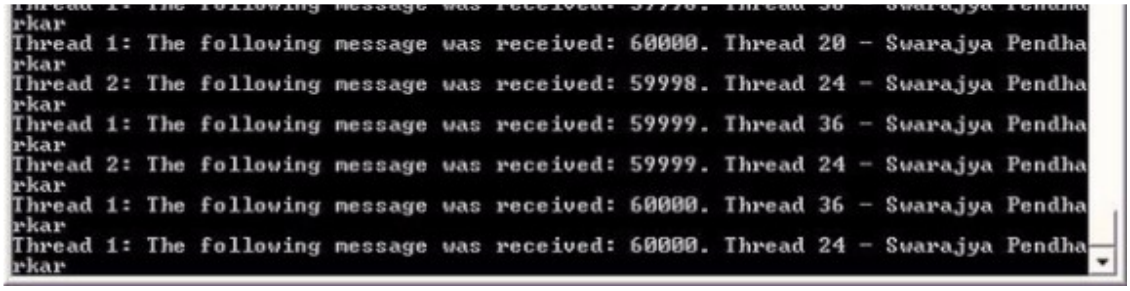
```
        }

        //Display the message received on console
        printf("\nThe following message was received: %s", szBuffer);

        //Send data on a connected socket to the client
        nBytesSent = send(RemoteSocket, ACK_MESG_RECV ,
                                        strlen(ACK_MESG_RECV), 0);

        if (SOCKET_ERROR == nBytesSent)
        {
             closesocket(RemoteSocket);
             printf("\nError occurred while writing to socket.");
             return 1; //error
        }
        else
        {
             printf("\nsend() successful.");
        }

        return 0; //success
}
```

This design is not scalable, and can have severe performance issues.

Multi-threaded server using IOCP







The implementation of this server can be found in *ServerIOCP.zip*.

The following set of APIs is used in the implementation of IOCP. A quick read or review on MSDN will help in a quick grasping of the following code:

- `CreateIoCompletionPort()`
- `GetQueuedCompletionStatus()`
- `PostQueuedCompletionStatus()`

The `InitializeIOCP()` function will initialize IOCP, and create a worker thread pool that will process the IOCP requests. Generally, two threads are created per processor. Many programs will find out the number of processors on the host, and will create *<Number of Processors * 2>* number of worker threads. This can be created as a configurable parameter. This will allow the user of the application to configure the number of threads in an attempt to fine-tune the application. In my code, two threads will be created per processor on the host.

Hide Copy Code

```
///Function to Initialize IOCPbool InitializeIOCP()
{
     //Create I/O completion port
     g_hIOCompletionPort =
       CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0 );

     if ( NULL == g_hIOCompletionPort)
     {
          printf("\nError occurred while creating IOCP: %d.",
                 WSAGetLastError());
          return false;
     }

     DWORD nThreadID;

     //Create worker threads
     for (int ii = 0; ii < g_nThreads; ii++)
     {
          g_phWorkerThreads[ii] = CreateThread(0, 0,
              WorkerThread, (void *)(ii+1), 0, &nThreadID);
     }

     return true;
}
```

IOCP is to be used with overlapped I/O. The following code shows how to create an

overlapped socket:

Hide Copy Code

```
//Overlapped I/O follows the model established
//in Windows and can be performed only on
//sockets created through the WSASocket function
ListenSocket = WSASocket(AF_INET, SOCK_STREAM, 0,
                      NULL, 0, WSA_FLAG_OVERLAPPED);
if (INVALID_SOCKET == ListenSocket)
{
     printf("\nError occurred while opening socket: %ld.",
                                       WSAGetLastError());
     goto error;
}
else
{
     printf("\nWSASocket() successful.");
}
```

To have a graceful shutdown of this server, I have used `WSAEventSelect()`. We will process an Accept event, rather than blocking on the `accept()` call. The creation of `WSAEVENT` is demonstrated below:

Hide Copy Code

```
g_hAcceptEvent = WSACreateEvent();

     if (WSA_INVALID_EVENT == g_hAcceptEvent)
     {
          printf("\nError occurred while WSACreateEvent().");
          goto error;
     }

     if (SOCKET_ERROR == WSAEventSelect(ListenSocket,
                       g_hAcceptEvent, FD_ACCEPT))
     {
          printf("\nError occurred while WSAEventSelect().");
          WSACloseEvent(g_hAcceptEvent);
          goto error;
     }
```

The `AcceptThread()` will keep looking for the Accept event.

Hide Copy Code

```
//This thread will look for accept event
DWORD WINAPI AcceptThread(LPVOID lParam)
{
     SOCKET ListenSocket = (SOCKET)lParam;

     WSANETWORKEVENTS WSAEvents;

     //Accept thread will be around to look for
```

```
      //accept event, until a Shutdown event is not Signaled.
      while(WAIT_OBJECT_0 !=
            WaitForSingleObject(g_hShutdownEvent, 0))
      {
           if (WSA_WAIT_TIMEOUT != WSAWaitForMultipleEvents(1,
               &g_hAcceptEvent, FALSE, WAIT_TIMEOUT_INTERVAL, FALSE))
           {
                WSAEnumNetworkEvents(ListenSocket,
                     g_hAcceptEvent, &WSAEvents);
                if ((WSAEvents.lNetworkEvents & FD_ACCEPT) &&
                    (0 == WSAEvents.iErrorCode[FD_ACCEPT_BIT]))
                {
                     //Process it.
                     AcceptConnection(ListenSocket);
                }
           }
      }

      return 0;
}
```

The `AcceptConnection()` will process the Accept event. It will also associate the socket to IOCP, and will post a `WSARecv()` on the socket to receive the incoming client data.

Hide Shrink ▲ Copy Code

```
//This function will process the accept event
void AcceptConnection(SOCKET ListenSocket)
{
     sockaddr_in ClientAddress;
     int nClientLength = sizeof(ClientAddress);

     //Accept remote connection attempt from the client
     SOCKET Socket = accept(ListenSocket,
          (sockaddr*)&ClientAddress, &nClientLength);

     if (INVALID_SOCKET == Socket)
     {
          WriteToConsole("\nError occurred while " +
                         "accepting socket: %ld.",
                         WSAGetLastError());
     }

     //Display Client's IP
     WriteToConsole("\nClient connected from: %s",
                    inet_ntoa(ClientAddress.sin_addr));

     //Create a new ClientContext for this newly accepted client
     CClientContext   *pClientContext  = new CClientContext;
```

```
        pClientContext->SetOpCode(OP_READ);
        pClientContext->SetSocket(Socket);

        //Store this object
        AddToClientList(pClientContext);

        if (true == AssociateWithIOCP(pClientContext))
        {
             //Once the data is successfully received, we will print it
             pClientContext->SetOpCode(OP_WRITE);

             WSABUF *p_wbuf = pClientContext->GetWSABUFPtr();
             OVERLAPPED *p_ol = pClientContext->GetOVERLAPPEDPtr();

             //Get data.
             DWORD dwFlags = 0;
             DWORD dwBytes = 0;

             //Post initial Recv
             //This is a right place to post a initial Recv
             //Posting a initial Recv in WorkerThread
             //will create scalability issues.
             int nBytesRecv = WSARecv(pClientContext->GetSocket(),
                              p_wbuf, 1, &dwBytes, &dwFlags, p_ol, NULL

             if ((SOCKET_ERROR == nBytesRecv) &&
                 (WSA_IO_PENDING != WSAGetLastError()))
             {
                  WriteToConsole("\nError in Initial Post.");
             }
        }
}
```

The `AssociateWithIOCP()` will associate the accepted socket to IOCP.

Hide Copy Code

```
bool AssociateWithIOCP(CClientContext   *pClientContext)
{
    //Associate the socket with IOCP
    HANDLE hTemp = CreateIoCompletionPort((HANDLE)pClientContext->Ge
        g_hIOCompletionPort, (DWORD)pClientContext, 0);

    if (NULL == hTemp)
    {
        WriteToConsole("\nError occurred while" +
            " executing CreateIoCompletionPort().");

        //Let's not work with this client
        RemoveFromClientListAndFreeMemory(pClientContext);
```

```
        return false;
    }

    return true;
}
```

The class `CClientContext` is used to store client related information, like the client socket, and it has a buffer that will be dedicated to overlapped I/O. We need to ensure that the buffer that is used in overlapped I/O is not updated when the overlapped I/O is underway.

Hide Shrink ▲ Copy Code

```
class CClientContext
//To store and manage client related information
{
private:

     OVERLAPPED        *m_pol;
     WSABUF            *m_pwbuf;

     int               m_nTotalBytes;
     int               m_nSentBytes;

     //accepted socket
     SOCKET            m_Socket;
     //will be used by the worker thread
     //to decide what operation to perform
     int               m_nOpCode;
     char              m_szBuffer[MAX_BUFFER_LEN];

public:

     //Get/Set calls
     void SetOpCode(int n)
     {
          m_nOpCode = n;
     }

     int GetOpCode()
     {
          return m_nOpCode;
     }

     void SetTotalBytes(int n)
     {
          m_nTotalBytes = n;
     }

     int GetTotalBytes()
     {
```

```
{
     return m_nTotalBytes;
}

void SetSentBytes(int n)
{
     m_nSentBytes = n;
}

void IncrSentBytes(int n)
{
     m_nSentBytes += n;
}

int GetSentBytes()
{
     return m_nSentBytes;
}

void SetSocket(SOCKET s)
{
     m_Socket = s;
}

SOCKET GetSocket()
{
     return m_Socket;
}

void SetBuffer(char *szBuffer)
{
     strcpy(m_szBuffer, szBuffer);
}

void GetBuffer(char *szBuffer)
{
     strcpy(szBuffer, m_szBuffer);
}

void ZeroBuffer()
{
     ZeroMemory(m_szBuffer, MAX_BUFFER_LEN);
}

void SetWSABUFLength(int nLength)
{
     m_pwbuf->len = nLength;
}
```

```
int GetWSABUFLength()
{
     return m_pwbuf->len;
}

WSABUF* GetWSABUFPtr()
{
     return m_pwbuf;
}

OVERLAPPED* GetOVERLAPPEDPtr()
{
     return m_pol;
}

void ResetWSABUF()
{
     ZeroBuffer();
     m_pwbuf->buf = m_szBuffer;
     m_pwbuf->len = MAX_BUFFER_LEN;
}

//Constructor
CClientContext()
{
     m_pol = new OVERLAPPED;
     m_pwbuf = new WSABUF;

     ZeroMemory(m_pol, sizeof(OVERLAPPED));

     m_Socket =  SOCKET_ERROR;

     ZeroMemory(m_szBuffer, MAX_BUFFER_LEN);

     m_pwbuf->buf = m_szBuffer;
     m_pwbuf->len = MAX_BUFFER_LEN;

     m_nOpCode = 0;
     m_nTotalBytes = 0;
     m_nSentBytes = 0;
}

//destructor
~CClientContext()
{
     //Wait for the pending operations to complete
     while (!HasOverlappedIoCompleted(m_pol))
     {
          Sleep(0);
```

```
            }

            closesocket(m_Socket);

            //Cleanup
            delete m_pol;
            delete m_pwbuf;
        }
};
```

Next is the worker thread function, `WorkerThread()`. This function will wait for requests from IOCP, and it will process them. Depending on the operation code supplied, it will perform the appropriate operation. The `WorkerThread()`, in turn, will make operation requests to IOCP by setting the appropriate operation code of `CClientContext`. These requests will be routed to one of the worker threads, including the requesting worker thread.

Hide Shrink ▲ Copy Code

```
//Worker thread will service IOCP requests
DWORD WINAPI WorkerThread(LPVOID lpParam)
{
    int nThreadNo = (int)lpParam;

    void *lpContext = NULL;
    OVERLAPPED       *pOverlapped = NULL;
    CClientContext   *pClientContext = NULL;
    DWORD            dwBytesTransfered = 0;
    int nBytesRecv = 0;
    int nBytesSent = 0;
    DWORD             dwBytes = 0, dwFlags = 0;

    //Worker thread will be around to process requests,
    //until a Shutdown event is not Signaled.
    while (WAIT_OBJECT_0 != WaitForSingleObject(g_hShutdownEvent, 0)
    {
        BOOL bReturn = GetQueuedCompletionStatus(
            g_hIOCompletionPort,
            &dwBytesTransfered,
            (LPDWORD)&lpContext,
            &pOverlapped,
            INFINITE);

        if (NULL == lpContext)
        {
            //We are shutting down
            break;
        }

        //Get the client context
        pClientContext = (CClientContext *)lpContext;
```

```
if ((FALSE == bReturn) || ((TRUE == bReturn) &&
(0 == dwBytesTransfered)))
{
    //Client connection gone, remove it.
    RemoveFromClientListAndFreeMemory(pClientContext);
    continue;
}

WSABUF *p_wbuf = pClientContext->GetWSABUFPtr();
OVERLAPPED *p_ol = pClientContext->GetOVERLAPPEDPtr();

switch (pClientContext->GetOpCode())
{
case OP_READ:

    pClientContext->IncrSentBytes(dwBytesTransfered);

    //Write operation was finished, see if all the data was s
    //Else post another write.
    if(pClientContext->GetSentBytes() <
    pClientContext->GetTotalBytes())
    {
        pClientContext->SetOpCode(OP_READ);

        p_wbuf->buf += pClientContext->GetSentBytes();
        p_wbuf->len = pClientContext->GetTotalBytes() -
            pClientContext->GetSentBytes();

        dwFlags = 0;

        //Overlapped send
        nBytesSent = WSASend(pClientContext->GetSocket(),
            p_wbuf, 1, &dwBytes, dwFlags, p_ol, NULL);

        if ((SOCKET_ERROR == nBytesSent) &&
        (WSA_IO_PENDING != WSAGetLastError()))
        {
            //Let's not work with this client
            RemoveFromClientListAndFreeMemory(pClientContext
        }
    }
    else
    {
        //Once the data is successfully received, we will pri
        pClientContext->SetOpCode(OP_WRITE);
        pClientContext->ResetWSABUF();

        dwFlags = 0;
```

```
            dwFlags = 0;

            //Get the data.
            nBytesRecv = WSARecv(pClientContext->GetSocket(), p_
                &dwBytes, &dwFlags, p_ol, NULL);

            if ((SOCKET_ERROR == nBytesRecv) &&
            (WSA_IO_PENDING != WSAGetLastError()))
            {
                WriteToConsole("\nThread %d: Error occurred"
                " while executing WSARecv().", nThreadNo);

                //Let's not work with this client
                RemoveFromClientListAndFreeMemory(pClientContext
            }
        }

        break;

    case OP_WRITE:

        char szBuffer[MAX_BUFFER_LEN];

        //Display the message we recevied
        pClientContext->GetBuffer(szBuffer);

        WriteToConsole("\nThread %d: The following message"
        " was received: %s", nThreadNo, szBuffer);

        //Send the message back to the client.
        pClientContext->SetOpCode(OP_READ);


        pClientContext->SetTotalBytes(dwBytesTransfered);
        pClientContext->SetSentBytes(0);

        p_wbuf->len  = dwBytesTransfered;

        dwFlags = 0;

        //Overlapped send
        nBytesSent = WSASend(pClientContext->GetSocket(), p_wbuf
            &dwBytes, dwFlags, p_ol, NULL);

        if ((SOCKET_ERROR == nBytesSent) &&
        (WSA_IO_PENDING != WSAGetLastError()))
        {
            WriteToConsole("\nThread %d: Error "
            "occurred while executing WSASend().", nThreadNo);
```

```
                //Let's not work with this client
                RemoveFromClientListAndFreeMemory(pClientContext);
            }

            break;

        default:
            //We should never be reaching here, under normal circums
            break;
        } // switch
    } // while

    return 0;
}
```

Some of the functions used in initialization and cleanup are shown below. Notice that `PostQueuedCompletionStatus()` is used in `CleanUp()` to help `WorkerThread()` get out of blocking calls to `GetQueuedCompletionStatus()`.

Hide Shrink ▲ Copy Code

```
bool Initialize()
{
     //Find out number of processors and threads
     g_nThreads = WORKER_THREADS_PER_PROCESSOR * GetNoOfProcessors()

     printf("\nNumber of processors on host: %d", GetNoOfProcessors(

     printf("\nThe following number of worker threads" +
            " will be created: %d", g_nThreads);

     //Allocate memory to store thread handless
     g_phWorkerThreads = new HANDLE[g_nThreads];

     //Initialize the Console Critical Section
     InitializeCriticalSection(&g_csConsole);

     //Initialize the Client List Critical Section
     InitializeCriticalSection(&g_csClientList);

     //Create shutdown event
     g_hShutdownEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

     // Initialize Winsock
     WSADATA wsaData;

     int nResult;
     nResult = WSAStartup(MAKEWORD(2,2), &wsaData);

     if (NO_ERROR != nResult)
```

```
        {
            printf("\nError occurred while executing WSAStartup().");
            return false; //error
        }
        else
        {
            printf("\nWSAStartup() successful.");
        }

        if (false == InitializeIOCP())
        {
            printf("\nError occurred while initializing IOCP");
            return false;
        }
        else
        {
            printf("\nIOCP initialization successful.");
        }

        return true;
    }

    //Function to Initialize IOCP
    bool InitializeIOCP()
    {
        //Create I/O completion port
        g_hIOCompletionPort =
           CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0 );

        if ( NULL == g_hIOCompletionPort)
        {
            printf("\nError occurred while creating IOCP: %d.",
                                            WSAGetLastError());
            return false;
        }

        DWORD nThreadID;

        //Create worker threads
        for (int ii = 0; ii < g_nThreads; ii++)
        {
            g_phWorkerThreads[ii] = CreateThread(0, 0, WorkerThread,
                                  (void *)(ii+1), 0, &nThreadID);
        }

        return true;
    }

    void CleanUp()
```

```
{
     //Ask all threads to start shutting down
     SetEvent(g_hShutdownEvent);

     //Let Accept thread go down
     WaitForSingleObject(g_hAcceptThread, INFINITE);

     for (int i = 0; i < g_nThreads; i++)
     {
          //Help threads get out of blocking - GetQueuedCompletionSta
          PostQueuedCompletionStatus(g_hIOCompletionPort, 0,
                                     (DWORD) NULL, NULL);
     }

     //Let Worker Threads shutdown
     WaitForMultipleObjects(g_nThreads, g_phWorkerThreads, TRUE, INF

     //We are done with this event
     WSACloseEvent(g_hAcceptEvent);

     //Cleanup dynamic memory allocations, if there are any.
     CleanClientList();
}

void DeInitialize()
{
     //Delete the Console Critical Section.
     DeleteCriticalSection(&g_csConsole);

     //Delete the Client List Critical Section.
     DeleteCriticalSection(&g_csClientList);

     //Cleanup IOCP.
     CloseHandle(g_hIOCompletionPort);

     //Clean up the event.
     CloseHandle(g_hShutdownEvent);

     //Clean up memory allocated for the storage of thread handles
     delete[] g_phWorkerThreads;

     //Cleanup Winsock
     WSACleanup();
}
```

To store client information, I am using a vector; and to manage the list, I am using the following set of calls:

Hide Shrink ▲ Copy Code

```
//Store client related information in a vector
```

```
void AddToClientList(CClientContext   *pClientContext)
{
     EnterCriticalSection(&g_csClientList);

     //Store these structures in vectors
     g_ClientContext.push_back(pClientContext);

     LeaveCriticalSection(&g_csClientList);
}

//This function will allow to remove one single client out of the li
void RemoveFromClientListAndFreeMemory(CClientContext   *pClientCont
{
     EnterCriticalSection(&g_csClientList);

     std::vector <cclientcontext>::iterator IterClientContext;

     //Remove the supplied ClientContext
     //from the list and release the memory
     for (IterClientContext = g_ClientContext.begin();
          IterClientContext != g_ClientContext.end();
          IterClientContext++)
     {
          if (pClientContext == *IterClientContext)
          {
               g_ClientContext.erase(IterClientContext);

               //i/o will be cancelled and socket
               //will be closed by destructor.
               delete pClientContext;
               break;
          }
     }

     LeaveCriticalSection(&g_csClientList);
}

//Clean up the list, this function
//will be executed at the time of shutdown
void CleanClientList()
{
     EnterCriticalSection(&g_csClientList);

     std::vector <cclientcontext>::iterator IterClientContext;

     for (IterClientContext = g_ClientContext.begin();
          IterClientContext != g_ClientContext.end( );
          IterClientContext++)
     {
```

```
            //i/o will be cancelled and socket
            //will be closed by destructor.
            delete *IterClientContext;
      }

      g_ClientContext.clear();

      LeaveCriticalSection(&g_csClientList);
}
```

I have created and used a `WriteToConsole()` function which will synchronize the console output that is sent by the worker threads; there will be a race condition for console, as I am using `printf()` to write to the console. It uses a critical section for synchronization:

Hide Copy Code

```
//Function to synchronize console output
//Threads need to be synchronized while they write to console.
//WriteConsole() API can be used, it is thread-safe, I think.
//I have created my own function.
void WriteToConsole(char *szFormat, ...)
{
      EnterCriticalSection(&g_csConsole);

      va_list args;
      va_start(args, szFormat);

      vprintf(szFormat, args );

      va_end(args);

      LeaveCriticalSection(&g_csConsole);
}
```

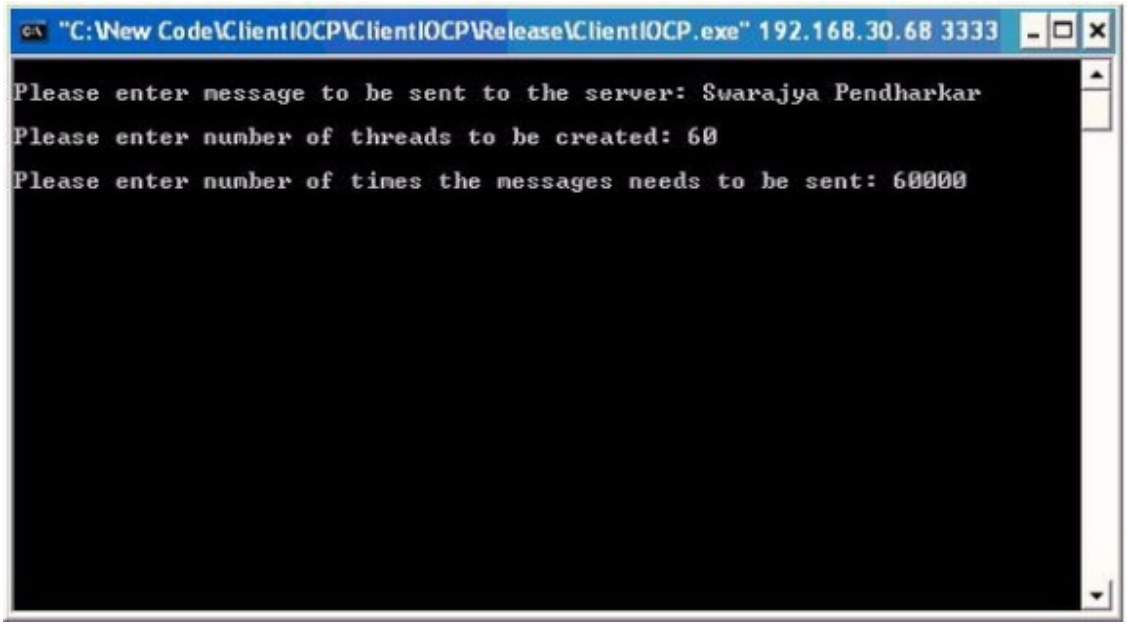
Finally, the code of `GetNoOfProcessors()`, a function that will get us the number of processors on the host:

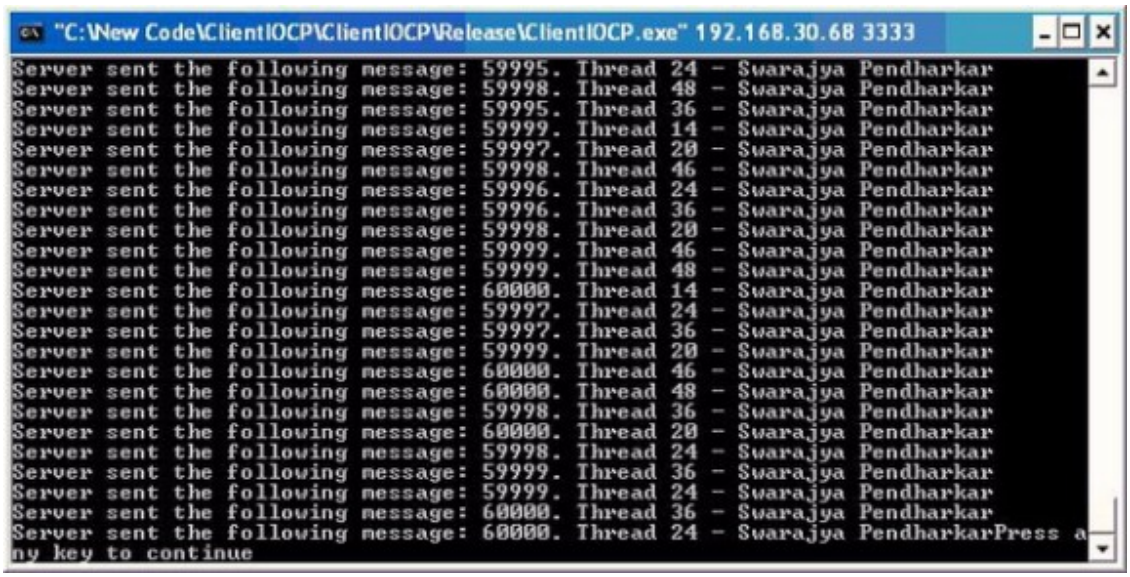
Hide Copy Code

```
//The use of static variable will ensure that
//we will make a call to GetSystemInfo()
//to find out number of processors,
//only if we don't have the information already.
//Repeated use of this function will be efficient.
int GetNoOfProcessors()
{
      static int nProcessors = 0;

      if (0 == nProcessors)
      {
            SYSTEM_INFO si;
```

```
            GetSystemInfo(&si);

            nProcessors = si.dwNumberOfProcessors;
        }

        return nProcessors;
}
```

IOCP client





The following is an IOCP client; it will let us stress-test the server. It uses traditional socket calls. I have used threads to put additional stress on the server.

Hide Shrink ▲ Copy Code

```
#include "stdafx.h"

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
```

```
#include <string.h>
#include <winsock2.h>

#include "ClientIOCP.h"

int main(int argc, char* argv[])
{
     //Validate the input
     if (argc < 3)
     {
          printf("\nUsage: %s hostname port.", argv[0]);
          return 1; //error
     }

     //Initialize Winsock
     WSADATA wsaData;

     int nResult = WSAStartup(MAKEWORD(2,2), &wsaData);

     if (NO_ERROR != nResult)
     {
          printf("\nError occurred while executing WSAStartup().");
          return 1; //error
     }

     //Initialize the Console Critical Section
     InitializeCriticalSection(&g_csConsole);

     int nPortNo = atoi(argv[2]);

     char szBuffer[MAX_BUFFER_LEN];
     int nNoOfThreads = 0;
     int nNoOfSends = 0;

     printf("\nPlease enter message to be sent to the server: ");

     //Read the message from server
     gets(szBuffer);

     printf("\nPlease enter number of threads to be created: ");

     //No. of times we will send the message to the server
     scanf("%d", &nNoOfThreads);

     printf("\nPlease enter number of times the" +
            " messages needs to be sent: ");

     //No. of times we will send the message to the server
     scanf("%d", &nNoOfSends);
```

```
HANDLE *p_hThreads = new HANDLE[nNoOfThreads];
ThreadInfo *pThreadInfo = new ThreadInfo[nNoOfThreads];

bool bConnectedSocketCreated = false;

DWORD nThreadID;

for (int ii = 0; ii < nNoOfThreads; ii++)
{
     bConnectedSocketCreated =
       CreateConnectedSocket(&(pThreadInfo[ii].m_Socket),
                             argv[1], nPortNo);

     if (!bConnectedSocketCreated)
     {
          //Clean up memory
          delete[] p_hThreads;
          delete[] pThreadInfo;

          //failed in creating of connected socket, error out.
          return 1;
     }

     //Populate ThreadInfo
     pThreadInfo[ii].m_nNoOfSends = nNoOfSends;
     pThreadInfo[ii].m_nThreadNo = ii+1;
     sprintf(pThreadInfo[ii].m_szBuffer,
             "Thread %d - %s", ii+1, szBuffer);

     //Create thread and start banging the server
     p_hThreads[ii] = CreateThread(0, 0, WorkerThread,
                      (void *)(&pThreadInfo[ii]), 0, &nThreadID
}

//Let Worker Threads shutdown
WaitForMultipleObjects(nNoOfThreads, p_hThreads, TRUE, INFINITE

//Close the sockets here
for (ii = 0; ii < nNoOfThreads; ii++)
{
     closesocket(pThreadInfo[ii].m_Socket);
}

//Clean up memory
delete[] p_hThreads;
delete[] pThreadInfo;

//Delete the Console Critical Section.
```

```
        DeleteCriticalSection(&g_csConsole);

        //Cleanup Winsock
        WSACleanup();
        return 0;
    }

    //vprintf() is not thread safe
    void WriteToConsole(char *szFormat, ...)
    {
        EnterCriticalSection(&g_csConsole);

        va_list args;
        va_start(args, szFormat);

        vprintf(szFormat, args );

        va_end(args);

        LeaveCriticalSection(&g_csConsole);
    }

    bool CreateConnectedSocket(SOCKET *pSocket, char *szHost, int nPortN
    {
        struct sockaddr_in ServerAddress;
        struct hostent *Server;

        //Create a socket
        *pSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

        if (INVALID_SOCKET == *pSocket)
        {
            WriteToConsole("\nError occurred while" +
                           " opening socket: %d.", WSAGetLastError());
            return false; //error
        }

        //Server name will be supplied as a commandline argument
        //Get the server details
        Server = gethostbyname(szHost);

        if (Server == NULL)
        {
            closesocket(*pSocket);
            WriteToConsole("\nError occurred no such host.");
            return false; //error
        }

        //Cleanup and Init with 0 the ServerAddress
```

```
        ZeroMemory((char *) &ServerAddress, sizeof(ServerAddress));

        ServerAddress.sin_family = AF_INET;

        //Assign the information received from gethostbyname()
        CopyMemory((char *)&ServerAddress.sin_addr.s_addr,
             (char *)Server->h_addr,
             Server->h_length);

        ServerAddress.sin_port = htons(nPortNo);

        //Establish connection with the server
        if (SOCKET_ERROR == connect(*pSocket,
            reinterpret_cast<const>(&ServerAddress),
            sizeof(ServerAddress)))
        {
             closesocket(*pSocket);
             WriteToConsole("\nError occurred while connecting.");
             return false; //error
        }

        return true;
    }

    DWORD WINAPI WorkerThread(LPVOID lpParam)
    {
        ThreadInfo *pThreadInfo = (ThreadInfo*)lpParam;

        char szTemp[MAX_BUFFER_LEN];

        int nBytesSent = 0;
        int nBytesRecv = 0;

        for (int ii = 0; ii < pThreadInfo->m_nNoOfSends; ii++)
        {
             sprintf(szTemp, "%d. %s", ii+1, pThreadInfo->m_szBuffer);

             //Send the message to the server, include the NULL as well
             nBytesSent = send(pThreadInfo->m_Socket, szTemp, strlen(sz

             if (SOCKET_ERROR == nBytesSent)
             {
                  WriteToConsole("\nError occurred while " +
                                 "writing to socket %ld.",
                                 WSAGetLastError());
                  return 1; //error
             }

             //Get the message from the server
```

```
            //Use the message from the server
            nBytesRecv = recv(pThreadInfo->m_Socket, szTemp, 255, 0);

            if (SOCKET_ERROR == nBytesRecv)
            {
                 WriteToConsole("\nError occurred while reading " +
                                "from socket %ld.", WSAGetLastError())
                 return 1; //error
            }

            //Display the server message
            WriteToConsole("\nServer sent the following message: %s",
       }

       return 0; //success
}
```

Points of interest

I/O completion port is a very powerful mechanism to create highly scalable and robust server applications, and needs to be used with overlapped I/O. It should be used with WinSock in socket based server applications serving multiple clients. IOCP will add a lot of value to the design.

History

- 24th March, 2006 – Implemented using overlapped I/O.
- April, 2006 – Spent time studying overlapped I/O. Made updates related to overlapped I/O.
- 29th May, 2006 – Changed the IOCP implementation, and implemented a graceful shutdown of the server.
- 9th June, 2006 – Added comments for easy understanding of the code, and minor code updates.
- 19th August, 2006 – Updated client and server for stress-testing.
- 22nd August, 2006 – Client is multi-threaded now. The client gets additional pounding power.
- 24th September, 2006 – The number of worker threads created on the IOCP server will be proportional to the number of processors on the host. Also, both client and server use new improved `WriteToConsole()`.
- 28th September, 2006 – Minor updates.
- 3rd February, 2007 - Fixed a defect as per input by xircon (CodeProject) and Visual C++ 2005 migration updates.
- March, 2007 - Cut down CPU usage by updating WSAWaitForMultipleEvents().