Network Programming for Windows 05:

Winsock I/O Methods

jintaeks@dongseo.ac.kr
Division of Digital Contents, DSU
May 22, 2019

Outline

- ✓ Socket Modes
- ✓ Socket I/O Models
- ✓ I/O Model Consideration

- ✓ Winsock features socket modes and socket I/O models to control how I/O is processed on a socket.
- ✓ A **socket mode** simply determines how Winsock functions behave when called with a socket.
  - Two socket modes: **blocking** and **non-blocking**.
- ✓ A **socket model** describes how an application manages and processes I/O on a socket.
  - **blocking**, **select()**, **WSAAsyncSelect()**, **WSAEventSelect()**, **overlapped I/O**, and **completion port**.

Socket Modes

- ✓ In **blocking mode**, Winsock calls that perform I/O, such as send() and recv() wait until the operation is complete before they return to the program.
- ✓ In **non-blocking mode**, the Winsock functions return immediately.

Blocking Mode

- ✓ The problem with this code is that the recv() function might never return.

```
SOCKET  sock;
char    buff[256];
int     done = 0, nBytes;
…

while (!done)
{
    nBytes = recv(sock, buff, 65);
    if (nBytes == SOCKET_ERROR)
    {
        printf("recv failed with error %d₩n", WSAGetLastError());
        return;
    }
    DoComputationOnData(buff);
}
```

✓ One method is to **separate the application into a reading thread and a computation thread**.

```
#define MAX_BUFFER_SIZE    4096

// Initialize critical section (data) and create
// an auto-reset event (hEvent) before creating the two threads
CRITICAL_SECTION data;
HANDLE           hEvent;

SOCKET           sock;
TCHAR            buff[MAX_BUFFER_SIZE];
int              done = 0;

// Create and connect sock
...
```

```
// Reader thread
void ReadThread(void)
{
    int nTotal = 0,
        nRead = 0,
        nLeft = 0,
        nBytes = 0;

    while (!done)     {
        nTotal = 0;
        nLeft = NUM_BYTES_REQUIRED;

        // However many bytes constitutes enough data for processing (i.e. non-zero)
        while (nTotal != NUM_BYTES_REQUIRED)        {
            EnterCriticalSection(&data);
            nRead = recv(sock, &(buff[MAX_BUFFER_SIZE - nBytes]), nLeft, 0);
            if (nRead == -1)            {
                printf("error₩n");
                ExitThread();
            }
            nTotal += nRead;
            nLeft -= nRead;

            nBytes += nRead;
            LeaveCriticalSection(&data);
        }
        SetEvent(hEvent);
    }
}
```

```
// Computation thread
void ProcessThread(void)
{
    WaitForSingleObject(hEvent);
    EnterCriticalSection(&data);
    DoSomeComputationOnData(buff);

    // Remove the processed data from the input
    // buffer, and shift the remaining data to the start of the array
    nBytes -= NUM_BYTES_REQUIRED;

    LeaveCriticalSection(&data);
}
```

Non-blocking Mode

- ✓ Winsock API calls that deal with sending and receiving data or connection management return immediately.

```
SOCKET         s;
unsigned long ul = 1;
int            nRet;

s = socket(AF_INET, SOCK_STREAM, 0);

nRet = ioctlsocket(s, FIONBIO, (unsigned long *)&ul);

if (nRet == SOCKET_ERROR) {
    // Failed to put the socket into non-blocking mode
}
```

- ✓ Because non-blocking calls frequently fail with the **WSAEWOULDBLOCK** error, you should check all return codes and be prepared for failure at any time.

Socket I/O Models

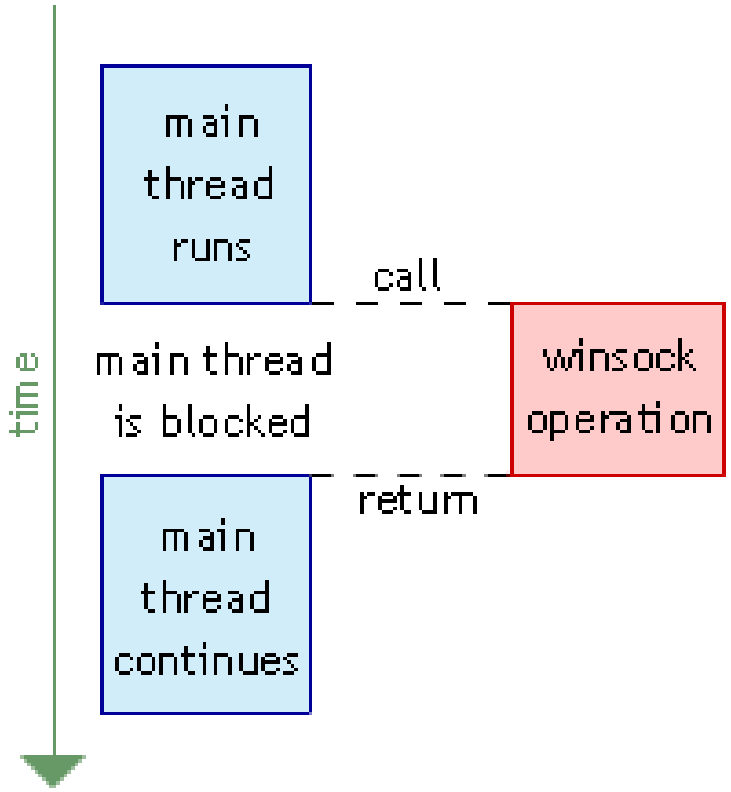
A **socket model** describes how an application manages and processes I/O on a socket.

① blocking
② select()
③ WSAAsyncSelect()
④ WSAEventSelect()
⑤ overlapped
⑥ completion port

Model	Blocking mode	Notification method		
		none	*on network event*	*on completion*
Blocking sockets	blocking	**x**		
Polling	non-blocking	**x**		
Select	both		blocking select	
WSAAsyncSelect	non-blocking		window message	
WSAEventSelect	non-blocking		event objects	
Overlapped I/O: blocking	N/A			blocking call
Overlapped I/O: polling	N/A	**x**		
Overlapped I/O: completion routines	N/A			callback function
Overlapped I/O: completion ports	N/A			completion port

~~The Blocking Model~~

- ✓ Straightforward model.
  - chapter 1
- ✓ Applications following this model typically use one or two **threads per socket connection** for handling I/O.
- ✓ Each thread will then issue blocking operations, such as send() and recv().
- ✓ For very simple applications and **rapid prototyping**, this model is very useful.



```
DWORD WINAPI ServerListenThread(LPVOID lpParam)
{
    CONNECTION_OBJ *ConnObj = NULL;
    HANDLE          hThread = NULL;
    SOCKET          s;
    int             rc;

    s = (SOCKET)lpParam;
...
 while (1)
        {
            // Wait for an incoming client connection
            ns = accept(s, (SOCKADDR *)&saAccept, &acceptlen);
            if (ns == INVALID_SOCKET)
            {
                fprintf(stderr, "accept failed: %d₩n", WSAGetLastError());
                return -1;
            }

            InterlockedIncrement(&gConnectedClients);
```

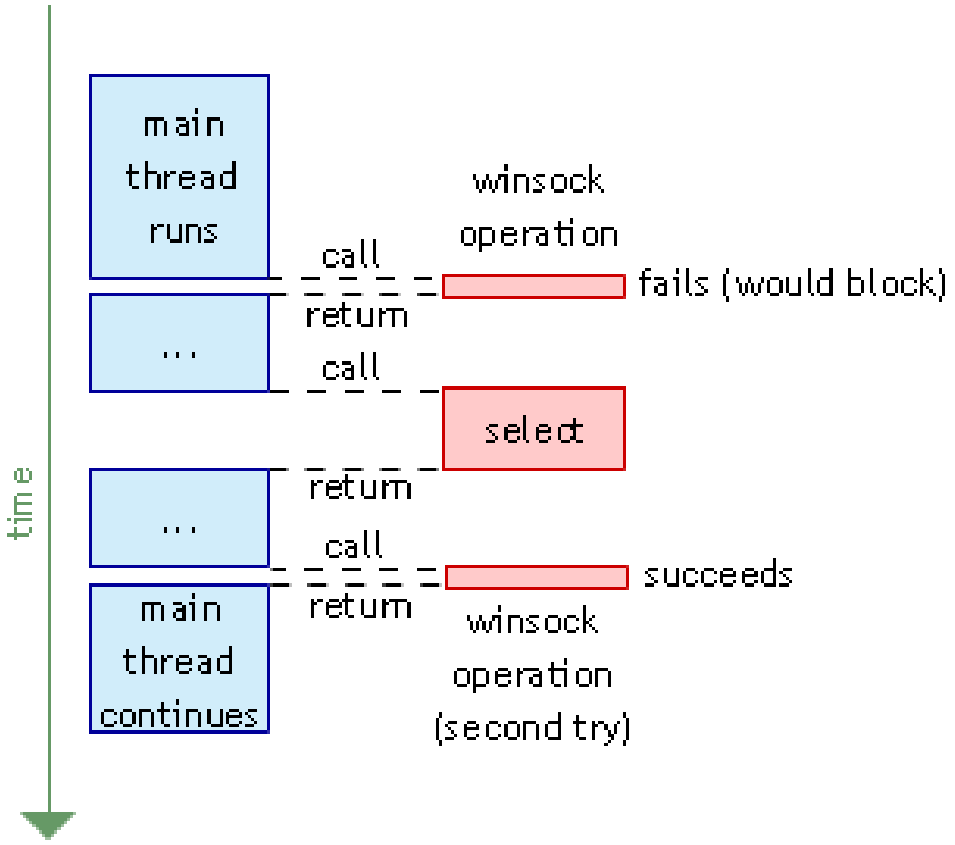
...

```
// Allocate a connection object for this client
ConnObj = GetConnectionObj(ns);

// Create a receiver thread for this connection
hThread = CreateThread(NULL, 0, ReceiveThread, (LPVOID)ConnObj, 0, NULL);
if (hThread == NULL)
{
    fprintf(stderr, "CreateThread failed: %d₩n", GetLastError());
    ExitThread(-1);
}
CloseHandle(hThread);

// Create a sender thread for this connection
hThread = CreateThread(NULL, 0, SendThread, (LPVOID)ConnObj, 0, NULL);
if (hThread == NULL)
{
    fprintf(stderr, "CreateThread failed: %d₩n", GetLastError());
    ExitThread(-1);
}
CloseHandle(hThread);
```

~~The select Model~~

- ✓ We call it the select model because it centers on using the **select**() function to manage I/O.
- ✓ The **select**() function can be used to determine if there is data on a socket and if a socket can be written to.

```
int select(
    int nfds,
    fd_set FAR * readfds,
    fd_set FAR * writefds,
    fd_set FAR * exceptfds,
    const struct timeval FAR * timeout
    );
```



```
SOCKET  s;
fd_set  fdread;
int     ret;

// Create a socket, and accept a connection

// Manage I/O on the socket
while (TRUE){
    // Always clear the read set before calling select()
    FD_ZERO(&fdread);
    // Add socket s to the read set
    FD_SET(s, &fdread);

    if ((ret = select(0, &fdread, NULL, NULL, NULL)) == SOCKET_ERROR)     {
        // Error condition
    }

    if (ret > 0)    {
        // For this simple case, select() should return the value 1.
        // An application dealing with more than one socket could get a value
        // greater than 1. At this point, your
        // application should check to see whether the socket is part of a set.
        if (FD_ISSET(s, &fdread))       {
            // A read event has occurred on socket s
        }
    }
}
```

Practice

- ✓ chapter05 → select client/server projects
- ✓ chapter05 → blocking server project

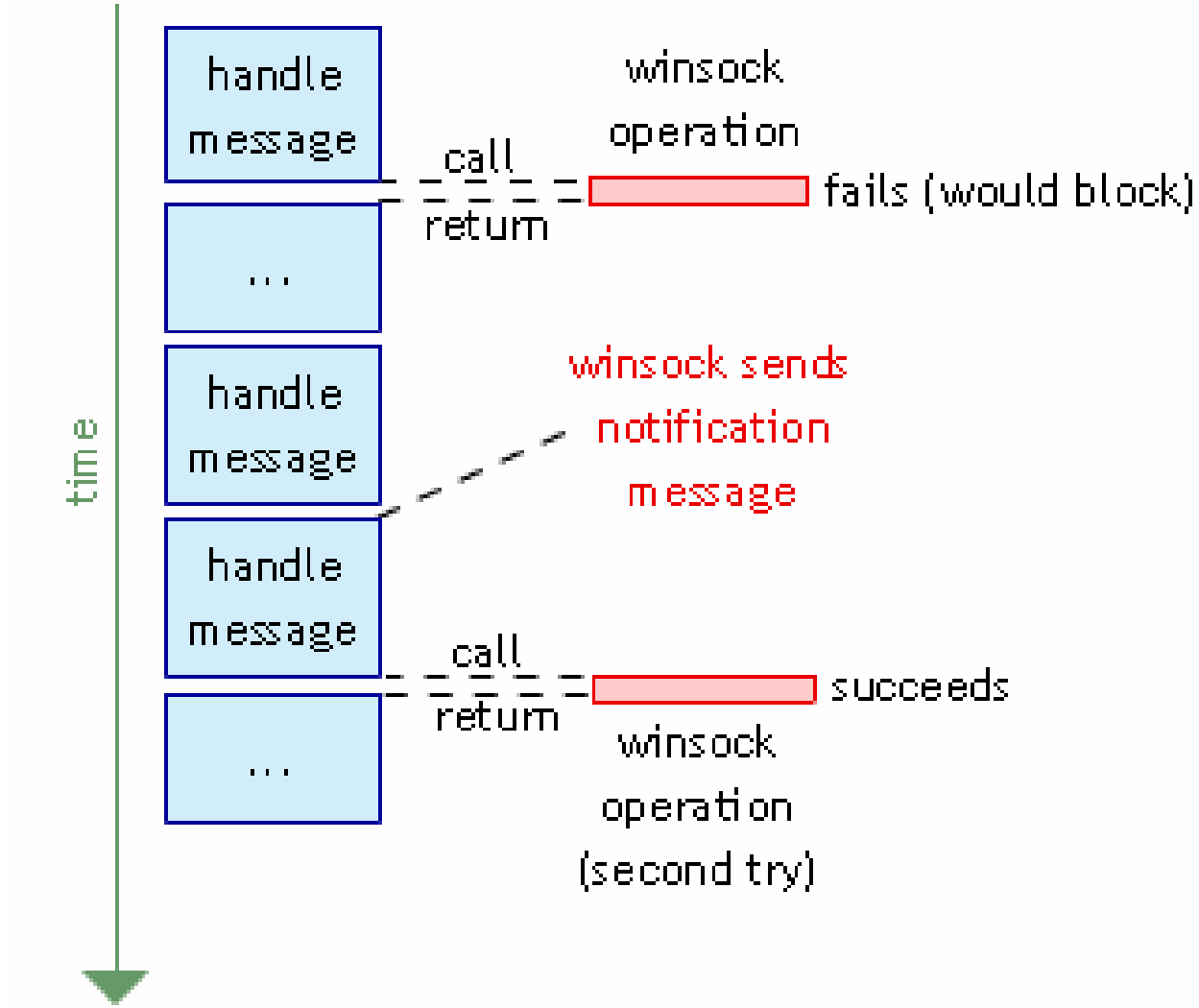
The WSAAsyncSelect() Model

- ✓ Asynchronous I/O model that allows an application to receive **Windows message–based** notification of network events on a socket.
- ✓ The WSAAsyncSelect() and WSAEventSelect() models **does not** provide asynchronous data transfer like the overlapped and completion port models.
- ✓ This model is also used by the Microsoft Foundation Class (**MFC**) **CSocket** object.

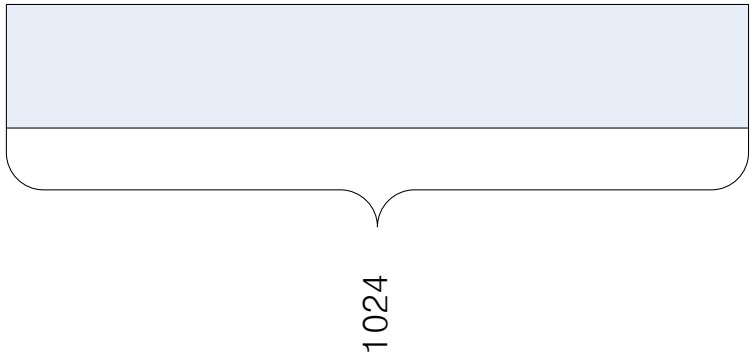
```
int WSAAsyncSelect(
    SOCKET s,
    HWND hWnd,
    unsigned int wMsg,
    long lEvent
    );
```

Practice

✓ chapter05 → AsyncSelectServer project



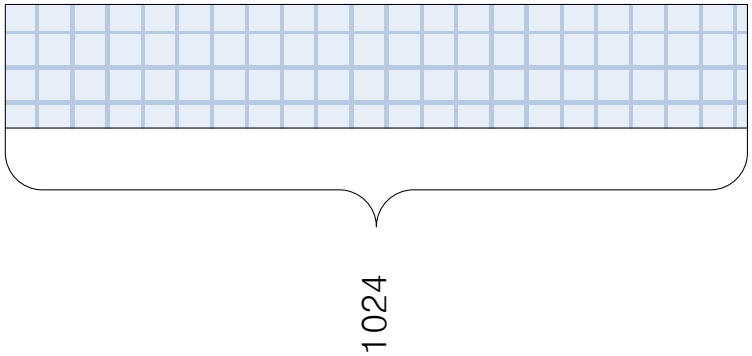
case 1



time

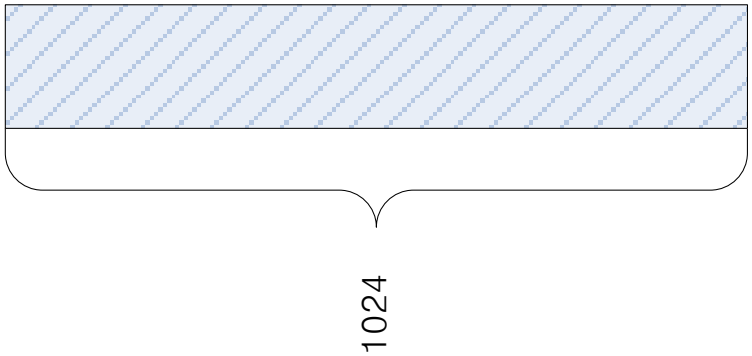
```
case FD_READ:
```

WSARecv(, 1024, ...)

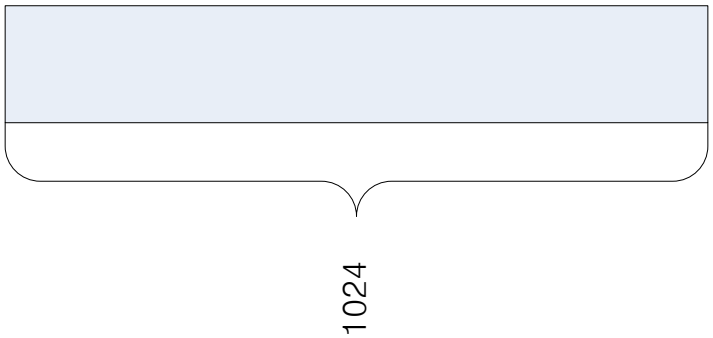



```
case FD_WRITE:
```

WSASend(, 1024, ...)

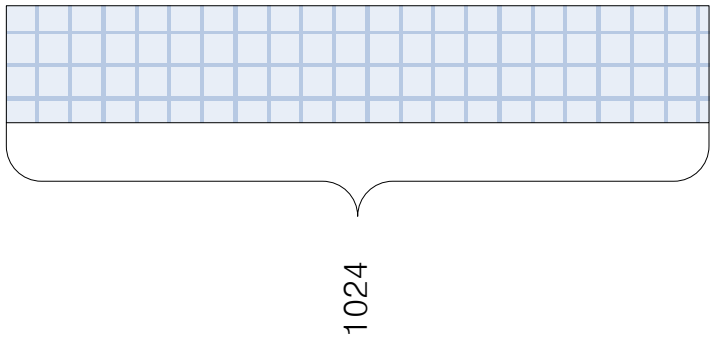


case 2

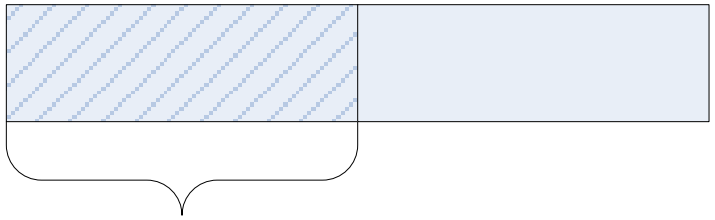


time

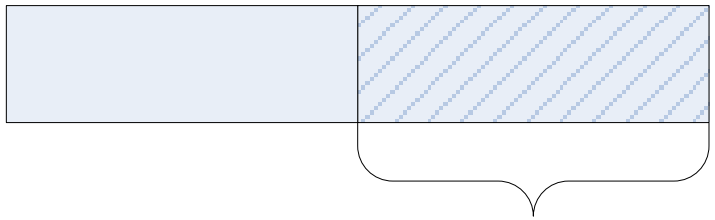
```
case FD_READ:
        WSARecv(, 1024, ...)
```



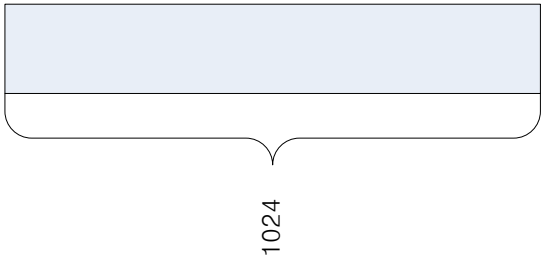

```
case FD_WRITE:
        WSASend(, 512, ...)
```




```
case FD_WRITE:
        WSASend(, 512, ...)
```

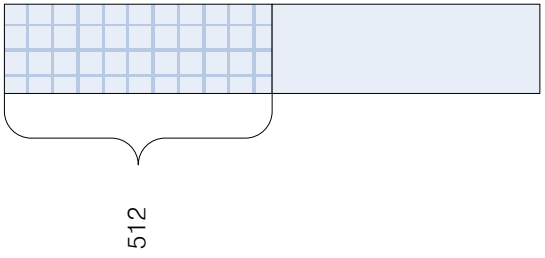


case 3

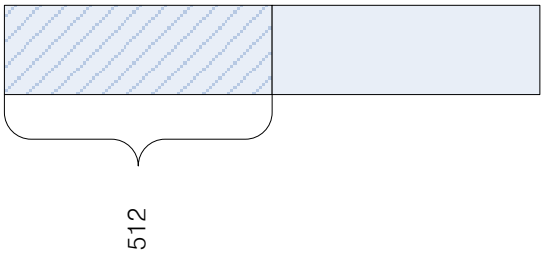


time

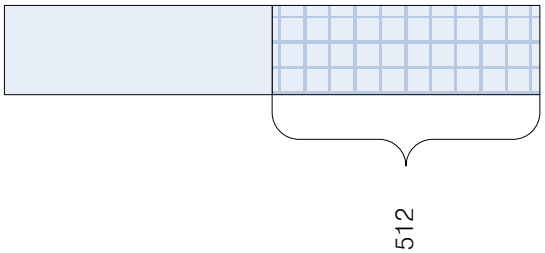
```
case FD_READ:
        WSARecv(, 512, ...)
```



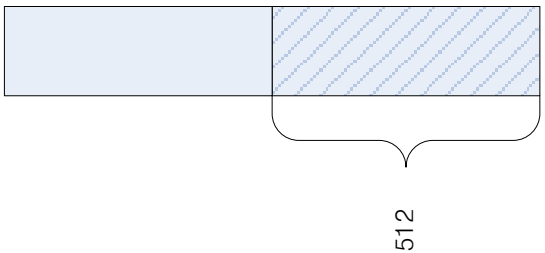

```
case FD_WRITE:
        WSASend(, 512, ...)
```



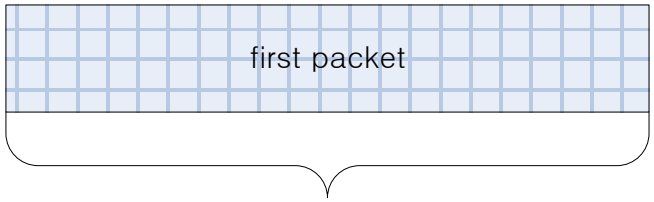

```
case FD_READ:
        WSARecv(, 512, ...)
```



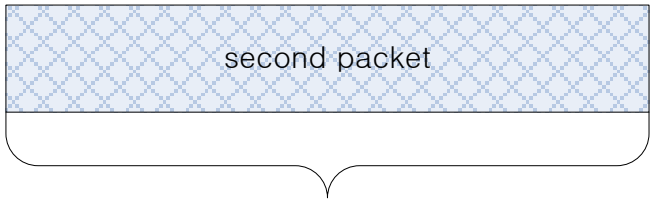

```
case FD_WRITE:
        WSASend(, 512, ...)
```



case 4



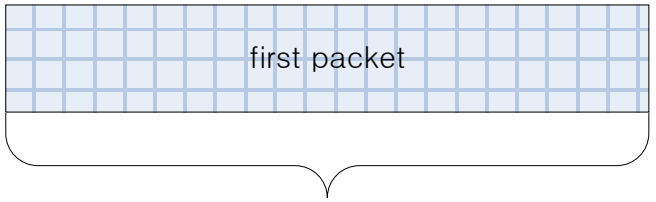
1024



1024

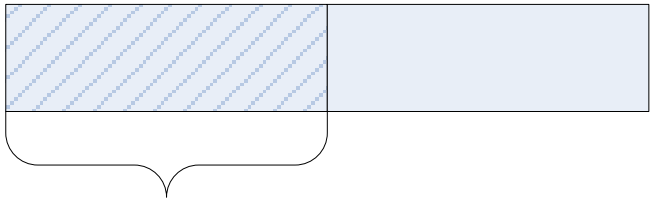
time

```
case FD_READ:
        WSARecv(, 1024, ...)
```



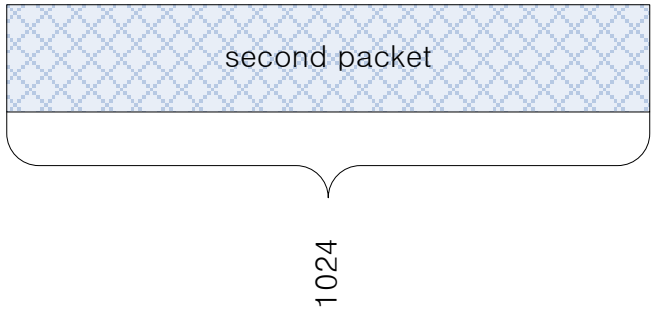
1024

```
case FD_WRITE:
        WSASend(, 512, ...)
```



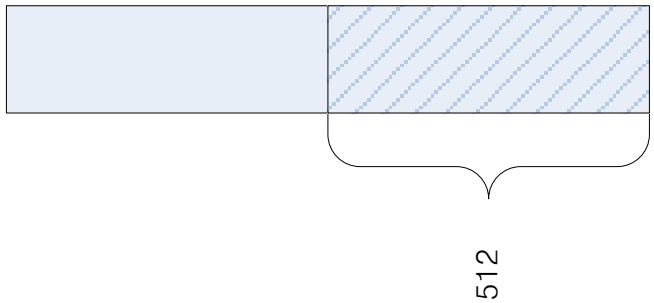
512

```
case FD_READ:
    SocketInfo->RecvPosted = TRUE;
    return 0;
case FD_WRITE:
```



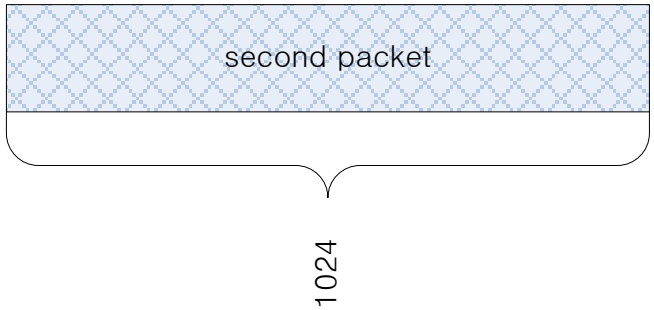

```
case FD_READ:
    SocketInfo->RecvPosted = TRUE;
    return 0;
case FD_WRITE:
```

WSASend(, 512, ...)




```
PostMessage(hwnd, WM_SOCKET,
    wParam, FD_READ);

case FD_READ:
```

WSARecv(, 1024, ...)



```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    SOCKET Accept;
    LPSOCKET_INFORMATION SocketInfo;
    DWORD RecvBytes;
    DWORD SendBytes;
    DWORD Flags;

    if (uMsg == WM_SOCKET)
    {
        if (WSAGETSELECTERROR(lParam))
        {
            printf("Socket failed with error %d\n", WSAGETSELECTERROR(lParam));
            FreeSocketInformation(wParam);
        }
        else
        {
            printf("Socket looks fine!\n");
            switch (WSAGETSELECTEVENT(lParam))
            {
            case FD_ACCEPT:
                if ((Accept = accept(wParam, NULL, NULL)) == INVALID_SOCKET)
                {
                    printf("accept() failed with error %d\n", WSAGetLastError());
                    break;
                }
                else
                    printf("accept() is OK!\n");

                // Create a socket information structure to associate with the socket for processing
I/O

                CreateSocketInformation(Accept);
                printf("Socket number %d connected\n", Accept);
                WSAAsyncSelect(Accept, hwnd, WM_SOCKET, FD_READ | FD_WRITE | FD_CLOSE);
                break;
```

```
case FD_READ:
        SocketInfo = GetSocketInformation(wParam);
        // Read data only if the receive buffer is empty
        if (SocketInfo->BytesRECV != 0)
        {
            SocketInfo->RecvPosted = TRUE;
            return 0;
        }
        else
        {
            SocketInfo->DataBuf.buf = SocketInfo->Buffer;
            SocketInfo->DataBuf.len = DATA_BUFSIZE;

            Flags = 0;
            if (WSARecv(SocketInfo->Socket, &(SocketInfo->DataBuf), 1, &RecvBytes,
                &Flags, NULL, NULL) == SOCKET_ERROR)
            {
                if (WSAGetLastError() != WSAEWOULDBLOCK)
                {
                    printf("WSARecv() failed with error %d\n", WSAGetLastError());
                    FreeSocketInformation(wParam);
                    return 0;
                }
            }
            else // No error so update the byte count
            {
                printf("WSARecv() is OK!\n");
                SocketInfo->BytesRECV = RecvBytes;
            }
        }
        // DO NOT BREAK HERE SINCE WE GOT A SUCCESSFUL RECV. Go ahead
        // and begin writing data to the client
    case FD_WRITE:
        SocketInfo = GetSocketInformation(wParam);
```

```
        case FD_WRITE:
            SocketInfo = GetSocketInformation(wParam);
            if (SocketInfo->BytesRECV > SocketInfo->BytesSEND)
            {
                SocketInfo->DataBuf.buf = SocketInfo->Buffer + SocketInfo->BytesSEND;
                SocketInfo->DataBuf.len = SocketInfo->BytesRECV - SocketInfo->BytesSEND;

                if (WSASend(SocketInfo->Socket, &(SocketInfo->DataBuf), 1, &SendBytes, 0,
                    NULL, NULL) == SOCKET_ERROR)
                {
                    if (WSAGetLastError() != WSAEWOULDBLOCK)
                    {
                        printf("WSASend() failed with error %d\n", WSAGetLastError());
                        FreeSocketInformation(wParam);
                        return 0;
                    }
                }
                else // No error so update the byte count
                {
                    printf("WSASend() is OK!\n");
                    SocketInfo->BytesSEND += SendBytes;
                }
            }
            if (SocketInfo->BytesSEND == SocketInfo->BytesRECV)
            {
                SocketInfo->BytesSEND = 0;
                SocketInfo->BytesRECV = 0;
                // If a RECV occurred during our SENDs then we need to post an FD_READ
notification on the socket
                if (SocketInfo->RecvPosted == TRUE)
                {
                    SocketInfo->RecvPosted = FALSE;
                    PostMessage(hwnd, WM_SOCKET, wParam, FD_READ);
```

~~The WSAEventSelect Model~~

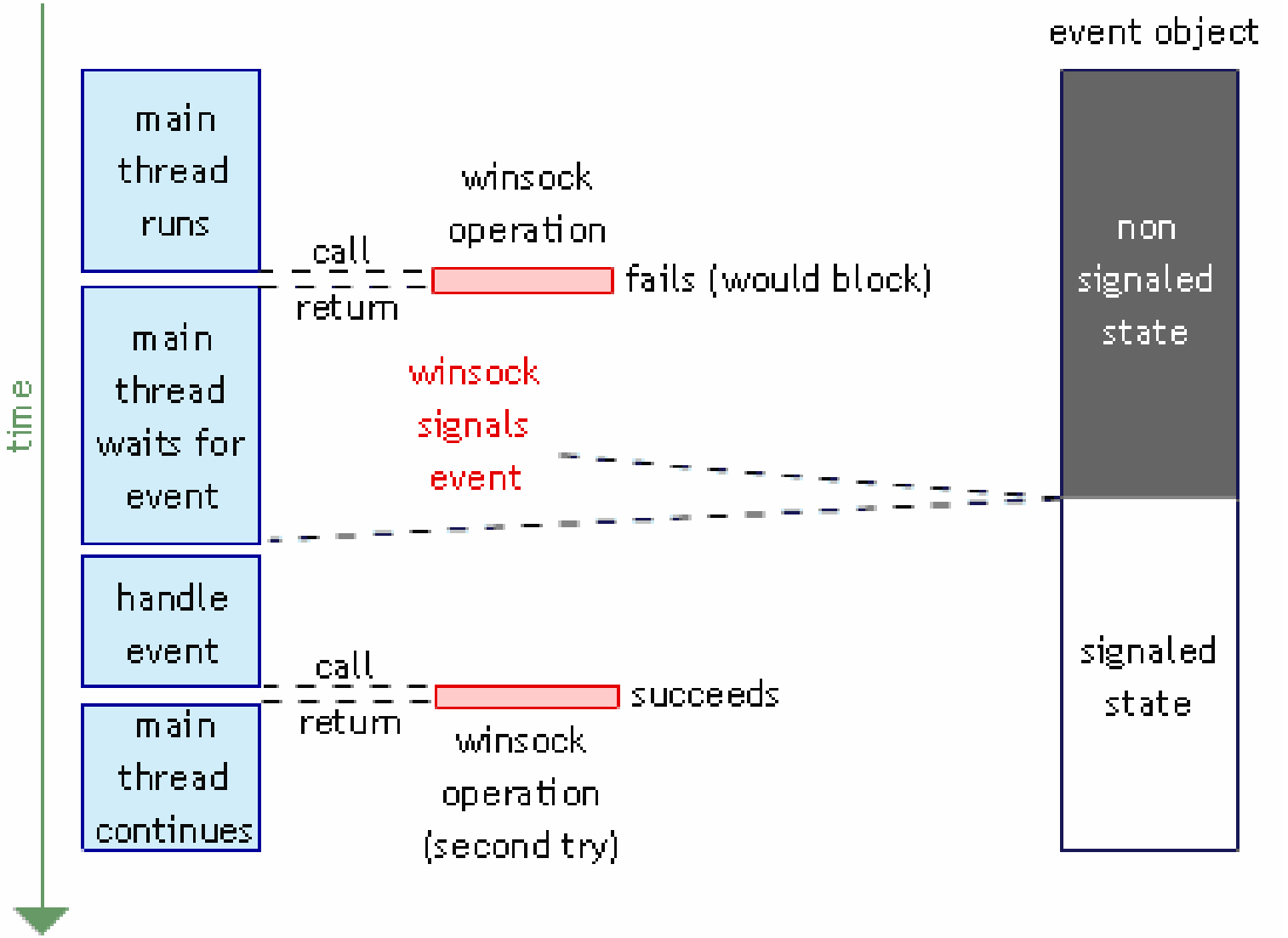
- ✓ Network events are notified via an **event object** handle instead of a window procedure.

```
int WSAEventSelect(
    SOCKET s,
    WSAEVENT hEventObject,
    long lNetworkEvents
    );
```

```
DWORD WSAWaitForMultipleEvents(
    DWORD cEvents,
    const WSAEVENT FAR * lphEvents,
    BOOL fWaitAll,
    DWORD dwTimeout,
    BOOL fAlertable
    );
```

~~Practice~~

✓ chapter05 → EventSelectServer project



The Overlapped Model

- ✓ The overlapped model's basic design allows your application to post one or more asynchronous I/O requests at a time using an overlapped data structure.
- ✓ The model's overall design is based on the **Windows overlapped I/O** mechanisms.
- ✓ To use the overlapped I/O model on a socket, you must first **create** a socket that has the **overlapped flag** set.
  - **if ((ListenSocket =** *WSASocket***(***AF_INET***, *SOCK_STREAM***, 0,** *NULL***, 0, *WSA_FLAG_OVERLAPPED*)) ==** *INVALID_SOCKET***)**
- ✓ To use overlapped I/O, each WSA function takes a **WSAOVERLAPPED** structure as a parameter.
  - **if (***WSARecv***(SI->Socket, &(SI->DataBuf), 1, &RecvBytes, &Flags, &(SI->Overlapped),** *NULL***) ==** *SOCKET_ERROR***)**

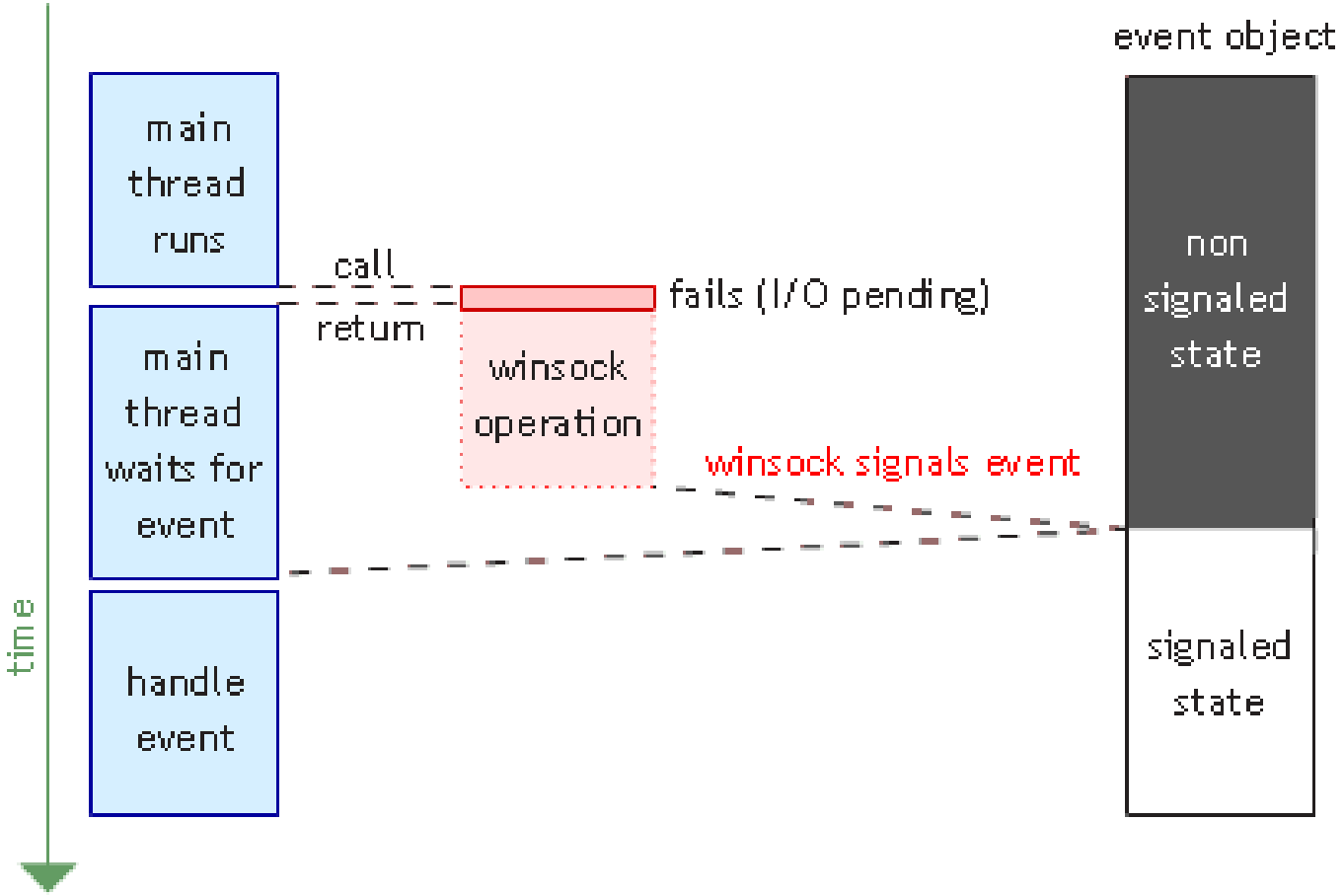
Event Notification

- ✓ The event notification method of overlapped I/O requires associating **Windows event objects** with **WSAOVERLAPPED** structures.
- ✓ When I/O calls such as WSASend() and WSARecv() are made using a WSAOVERLAPPED structure, they return immediately.

```
typedef struct WSAOVERLAPPED
{
    DWORD    Internal;
    DWORD    InternalHigh;
    DWORD    Offset;
    DWORD    OffsetHigh;
    WSAEVENT hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

```
BOOL WSAGetOverlappedResult(
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags
    );
```

- ✓ If the **WSAGetOverlappedResult**() function succeeds, the return value is TRUE.
- ✓ If the return value is FALSE, one of the following statements is true:
  - The overlapped I/O operation is still pending.
  - The overlapped operation completed, but with errors.


event object
main thread runs
call
return
fails (I/O pending)
main thread waits for event
winsock operation
winsock signals event
non signaled state
signaled state
handle event
time

1. **Create** a socket and begin **listening** for a connection on a specified port.
2. **Accept** an inbound connection.
3. Create a **WSAOVERLAPPED structure** for the accepted socket and assign an **event object** handle to the structure. Also assign the **event object handle** to an event array to be used later by the WSAWaitForMultipleEvents() function.
4. Post an asynchronous **WSARecv**() request on the socket by specifying the WSAOVERLAPPED structure as a parameter.

5. Call **WSAWaitForMultipleEvents**() using the **event array** and wait for the event associated with the overlapped call to become signaled.
6. Determine the **return status** of the overlapped call by using **WSAGetOverlappedResult**().
7. **Reset the event** object by using **WSAResetEvent**() with the event array and process the completed overlapped request.
8. **Post another overlapped WSARecv**() request on the socket.
9. Repeat steps 5–8.

```
DWORD WINAPI ProcessIO(LPVOID lpParameter)
```

```
#define PORT 5150
#define DATA_BUFSIZE 8192

typedef struct _SOCKET_INFORMATION {
    CHAR Buffer[DATA_BUFSIZE];
    WSABUF DataBuf;
    SOCKET Socket;
    WSAOVERLAPPED Overlapped;
    DWORD BytesSEND;
    DWORD BytesRECV;
} SOCKET_INFORMATION, *LPSOCKET_INFORMATION;

DWORD WINAPI ProcessIO(LPVOID lpParameter);

DWORD EventTotal = 0;
WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
LPSOCKET_INFORMATION SocketArray[WSA_MAXIMUM_WAIT_EVENTS];
CRITICAL_SECTION CriticalSection;

int main(int argc, char **argv)
{
    WSADATA wsaData;
    SOCKET ListenSocket, AcceptSocket;
    SOCKADDR_IN InternetAddr;
```

```
int main(int argc, char **argv)
{
    WSADATA wsaData;
    SOCKET ListenSocket, AcceptSocket;
    SOCKADDR_IN InternetAddr;
    DWORD Flags;
    DWORD ThreadId;
    DWORD RecvBytes;

    InitializeCriticalSection(&CriticalSection);

    WSAStartup((2, 2), &wsaData);
    printf("WSAStartup() looks nice!\n");
```

Step1

```
    ListenSocket = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
    printf("WSASocket() is OK lol!\n");

    InternetAddr.sin_family = AF_INET;
    InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    InternetAddr.sin_port = htons(PORT);

    bind(ListenSocket, (PSOCKADDR)&InternetAddr, sizeof(InternetAddr));
    printf("YOu see, bind() is working!\n");

    listen(ListenSocket, 5);
    printf("listen() is OK maa...\n");

    EventArray[0] = WSACreateEvent();
    printf("WSACreateEvent() is OK!\n");

    // Create a thread to service overlapped requests
    CreateThread(NULL, 0, ProcessIO, NULL, 0, &ThreadId);
    printf("Nothing to say, CreateThread() is OK!\n");

    EventTotal = 1;
```

```
EventTotal = 1;

    while (TRUE)
    {
        // Accept inbound connections
        AcceptSocket = accept(ListenSocket, NULL, NULL);
        printf("accept() is OK!\n");

        EnterCriticalSection(&CriticalSection);
        // Create a socket information structure to associate with the accepted socket
        SocketArray[EventTotal] = (LPSOCKET_INFORMATION)GlobalAlloc(GPTR,
sizeof(SOCKET_INFORMATION));
        printf("GlobalAlloc() for LPSOCKET_INFORMATION is pretty fine!\n");

        // Fill in the details of our accepted socket
        SocketArray[EventTotal]->Socket = AcceptSocket;
        ZeroMemory(&(SocketArray[EventTotal]->Overlapped), sizeof(OVERLAPPED));
        SocketArray[EventTotal]->BytesSEND = 0;
        SocketArray[EventTotal]->BytesRECV = 0;
        SocketArray[EventTotal]->DataBuf.len = DATA_BUFSIZE;
        SocketArray[EventTotal]->DataBuf.buf = SocketArray[EventTotal]->Buffer;

        SocketArray[EventTotal]->Overlapped.hEvent = EventArray[EventTotal] = WSACreateEvent();
        printf("WSACreateEvent() is OK!\n");

        // Post a WSARecv() request to to begin receiving data on the socket
        Flags = 0;
        WSARecv(SocketArray[EventTotal]->Socket,
            &(SocketArray[EventTotal]->DataBuf), 1, &RecvBytes, &Flags, &(SocketArray[EventTotal]-
>Overlapped), NULL);
        printf("WSARecv() should be working!\n");
```

Step2

Step3

```
        // Post a WSARecv() request to to begin receiving data on the socket
        Flags = 0;
        WSARecv(SocketArray[EventTotal]->Socket,
            &(SocketArray[EventTotal]->DataBuf), 1, &RecvBytes, &Flags, &(SocketArray[EventTotal]-
>Overlapped), NULL);
        printf("WSARecv() should be working!\n");

        EventTotal++;
        LeaveCriticalSection(&CriticalSection);

        // Signal the first event in the event array to tell the worker thread to
        // service an additional event in the event array
        WSASetEvent(EventArray[0]);
        printf("Don't worry, WSASetEvent() is OK!\n");
    }
}
```

Step4

```
DWORD WINAPI ProcessIO(LPVOID lpParameter)
{
    DWORD Index;
    DWORD Flags;
    LPSOCKET_INFORMATION SI;
    DWORD BytesTransferred;
    DWORD i;
    DWORD RecvBytes, SendBytes;

    // Process asynchronous WSASend, WSARecv requests
    while (TRUE)
    {
        Index = WSAWaitForMultipleEvents(EventTotal, EventArray, FALSE, WSA_INFINITE, FALSE);
        printf("WSAWaitForMultipleEvents() is OK!\n");

        // If the event triggered was zero then a connection attempt was made
        // on our listening socket.
        if ((Index - WSA_WAIT_EVENT_0) == 0)
        {
            WSAResetEvent(EventArray[0]);
            continue;
        }

        SI = SocketArray[Index - WSA_WAIT_EVENT_0];
        WSAResetEvent(EventArray[Index - WSA_WAIT_EVENT_0]);

        WSAGetOverlappedResult(SI->Socket, &(SI->Overlapped), &BytesTransferred, FALSE, &Flags);
        if( BytesTransferred == 0)
        {
            printf("Closing socket %d\n", SI->Socket);

            closesocket(SI->Socket);
            printf("closesocket() is OK!\n");
```

Step5

Step6,7

```
printf("closesocket() is OK!\n");

            GlobalFree(SI);
            WSACloseEvent(EventArray[Index - WSA_WAIT_EVENT_0]);
            // Cleanup SocketArray and EventArray by removing the socket event handle
            // and socket information structure if they are not at the end of the arrays
            EnterCriticalSection(&CriticalSection);

            if ((Index - WSA_WAIT_EVENT_0) + 1 != EventTotal)
                for (i = Index - WSA_WAIT_EVENT_0; i < EventTotal; i++)
                {
                    EventArray[i] = EventArray[i + 1];
                    SocketArray[i] = SocketArray[i + 1];
                }

            EventTotal--;
            LeaveCriticalSection(&CriticalSection);
            continue;
        }
        // Check to see if the BytesRECV field equals zero. If this is so, then
        // this means a WSARecv call just completed so update the BytesRECV field
        // with the BytesTransferred value from the completed WSARecv() call.
        if (SI->BytesRECV == 0)
        {
            SI->BytesRECV = BytesTransferred;
            SI->BytesSEND = 0;
        }
        else
        {
            SI->BytesSEND += BytesTransferred;
        }
```

```
if (SI->BytesSEND < SI->BytesRECV)
        {
            // Post another WSASend() request.
            // Since WSASend() is not guaranteed to send all of the bytes requested,
            // continue posting WSASend() calls until all received bytes are sent
            ZeroMemory(&(SI->Overlapped), sizeof(WSAOVERLAPPED));
            SI->Overlapped.hEvent = EventArray[Index - WSA_WAIT_EVENT_0];

            SI->DataBuf.buf = SI->Buffer + SI->BytesSEND;
            SI->DataBuf.len = SI->BytesRECV - SI->BytesSEND;

            WSASend(SI->Socket, &(SI->DataBuf), 1, &SendBytes, 0, &(SI->Overlapped), NULL);
            printf("WSASend() is OK!\n");
        }
        else
        {
            SI->BytesRECV = 0;
            // Now that there are no more bytes to send post another WSARecv() request
            Flags = 0;
            ZeroMemory(&(SI->Overlapped), sizeof(WSAOVERLAPPED));
            SI->Overlapped.hEvent = EventArray[Index - WSA_WAIT_EVENT_0];

            SI->DataBuf.len = DATA_BUFSIZE;
            SI->DataBuf.buf = SI->Buffer;

            WSARecv(SI->Socket, &(SI->DataBuf), 1, &RecvBytes, &Flags, &(SI->Overlapped), NULL);
            printf("WSARecv() is OK!\n");
        }
    }
}
```

Step8

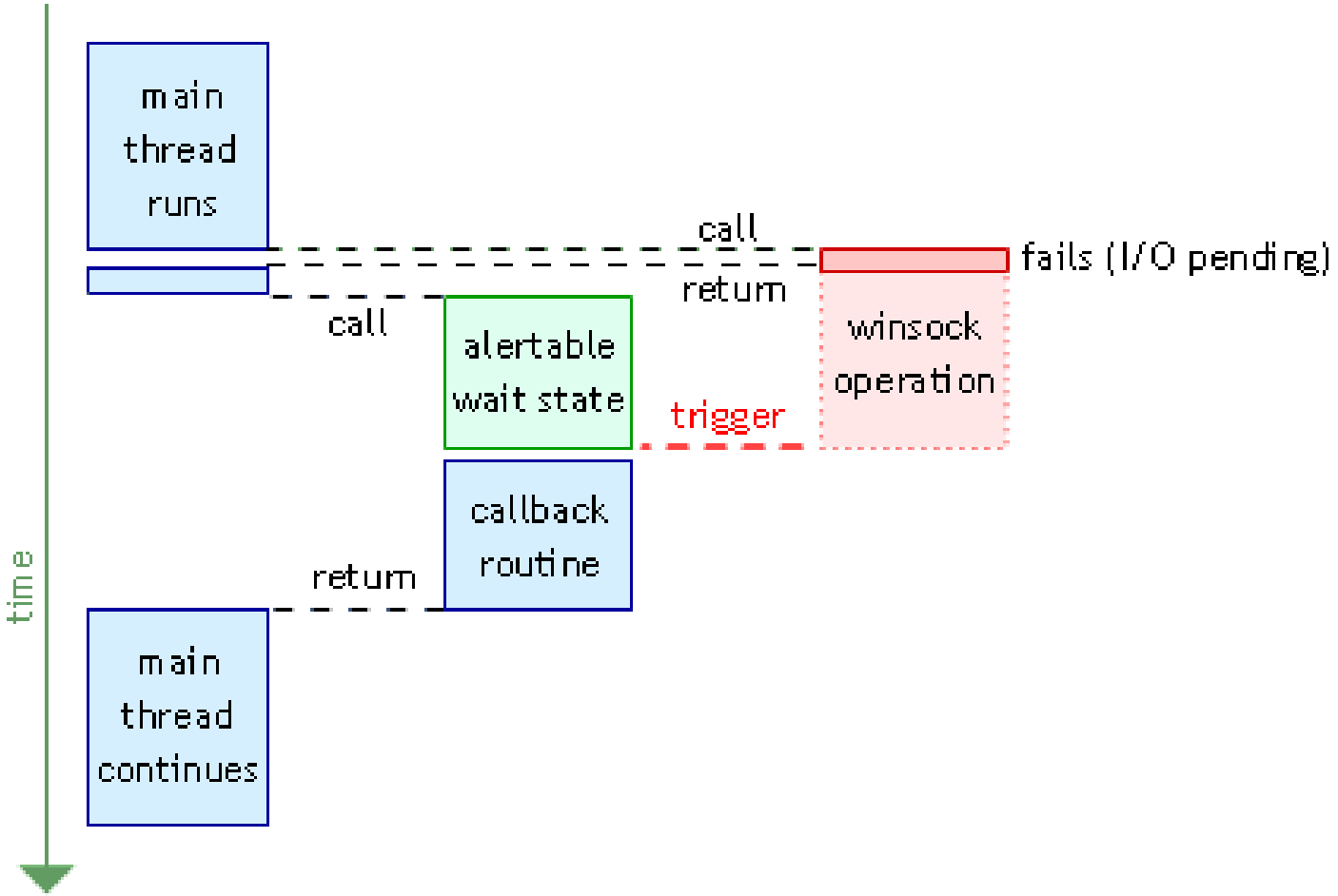
Question: What's this code?

```
if (listen(ListenSocket, 5))
    {
        printf("listen() failed with error %d\n", WSAGetLastError());
        return 1;
    }
    else
        printf("listen() is OK maa...\n");

    // Setup the listening socket for connections
    if ((AcceptSocket = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED)) ==
INVALID_SOCKET)
    {
        printf("Failed to get a socket %d\n", WSAGetLastError());
        return 1;
    }
    else
        printf("WSASocket() looks OK!\n");

    if ((EventArray[0] = WSACreateEvent()) == WSA_INVALID_EVENT)
    {
        printf("WSACreateEvent() failed with error %d\n", WSAGetLastError());
```

Completion Routines

- ✓ **Completion routines** are the other method your application can use to **manage completed overlapped I/O requests**.
- ✓ Completion routines are simply **functions** that the system invokes when an overlapped I/O request completes.
- ✓ Their primary role is to service a completed I/O request using the caller's thread. In addition, applications can continue overlapped I/O processing through the completion routine.
- ✓ To use completion routines for overlapped I/O requests, your application must specify a completion routine, along with a **WSAOVERLAPPED** structure.

```
void CALLBACK CompletionROUTINE(
    DWORD dwError,
    DWORD cbTransferred,
    LPWSAOVERLAPPED lpOverlapped,
    DWORD dwFlags
    );
```


main thread runs
call
fails (I/O pending)
return
call
alertable wait state
winsock operation
trigger
callback routine
return
main thread continues
time

1. **Create a socket** and begin **listening** for a connection on a specified port.
2. **Accept** an inbound connection.
3. Create a **WSAOVERLAPPED** structure for the accepted socket.
4. Post an asynchronous **WSARecv** request on the socket by ① specifying the WSAOVERLAPPED structure as a parameter and ② **supplying a completion routine**.

5. Call **WSAWaitForMultipleEvents**() with the fAlertable parameter set to TRUE and wait for an overlapped request to complete.

   When an overlapped request completes, the **completion routine automatically executes** and WSAWaitForMultipleEvents() returns WSA_IO_COMPLETION. Inside the completion routine, then post another overlapped WSARecv request with a completion routine.

6. Verify that **WSAWaitForMultipleEvents**() returns **WSA_IO_COMPLETION**.

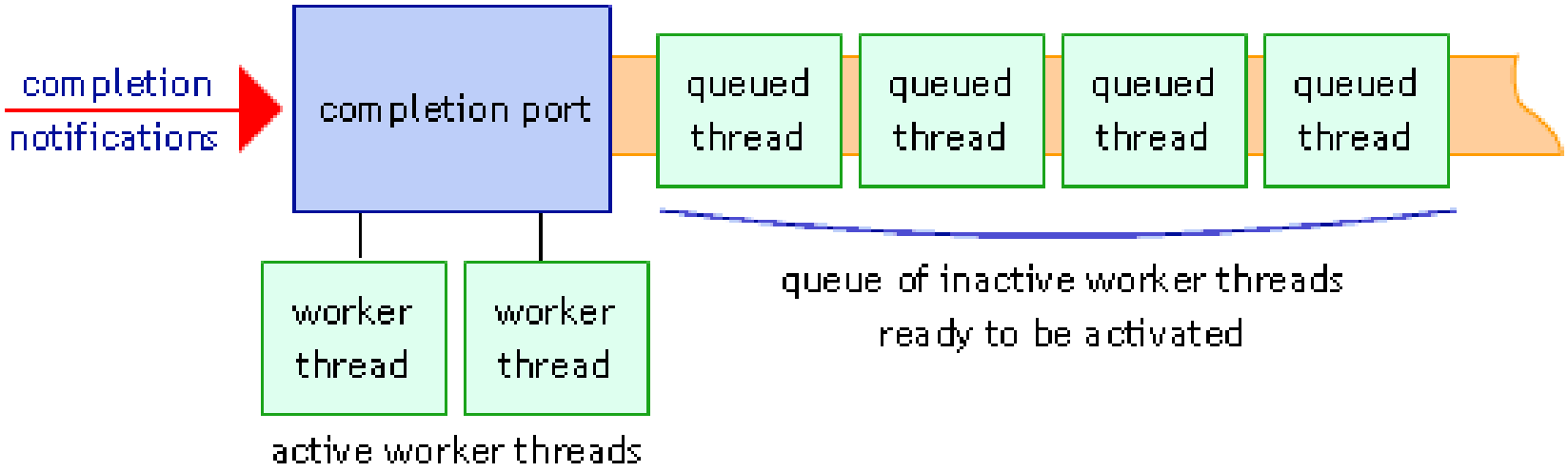
7. Repeat steps 5 and 6.

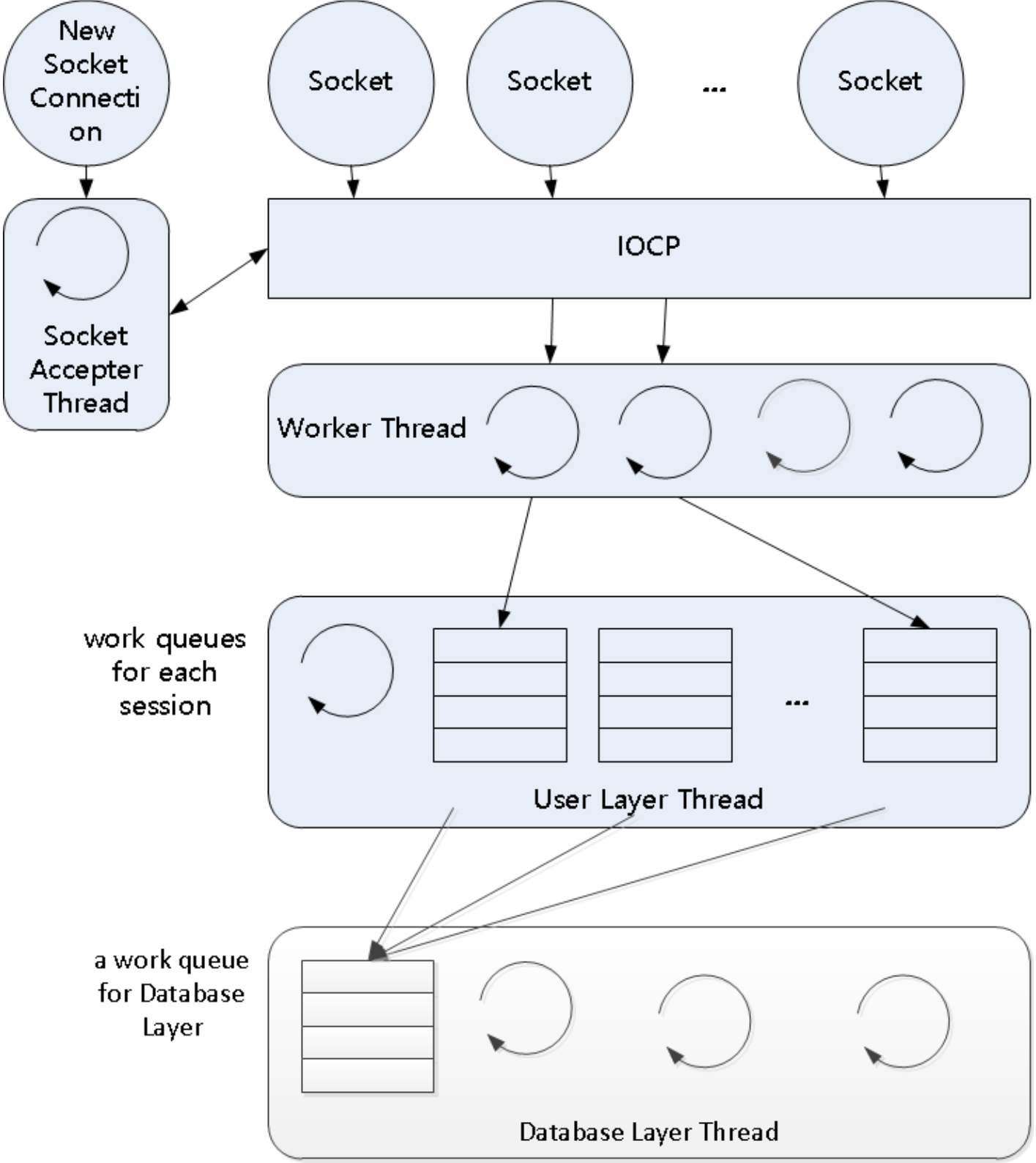
IOCP Model

FINALLY! The Completion Port Model

- ✓ The **completion port model** offers the best system performance possible when an application has to manage many sockets at once.
- ✓ It's available Windows NT and above.
- ✓ To begin using this model, you are required to create an **I/O completion port object** that will be used to manage multiple I/O requests for any number of socket handles.

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


completion
notifications
completion port
queued thread
queued thread
queued thread
queued thread
worker thread
worker thread
active worker threads
queue of inactive worker threads
ready to be activated


New Socket Connecti on
Socket
Socket
...
Socket
IOCP
Socket Accepter Thread
Worker Thread
work queues for each session
...
User Layer Thread
a work queue for Database Layer
Database Layer Thread

Worker Threads and Completion Ports

- ✓ After a **completion port** is successfully **created**, you can begin to **associate socket handles** with the object.
- ✓ Before associating sockets, though, you have to create **one or more worker threads** to service the completion port when socket I/O requests are posted to the completion port object.
- ✓ Calling the **CreateIoCompletionPort**() function on an existing completion port and supplying the first three parameters with socket information.

1. **Create a completion port**.
    1. The fourth parameter is left as 0, specifying that only one worker thread per processor will be allowed.
2. Determine how many **processors** exist on the system.
3. **Create worker threads** to service completed I/O requests on the completion port using processor information in step 2.
4. Prepare a **listening socket** to listen for connections on port 5150.

5. Accept **inbound connections** using the **accept** function.
6. Create a data structure to represent **per-handle data** and **save the accepted socket handle** in the structure.
7. Associate the new socket handle returned from accept with the completion port by calling **CreateIoCompletionPort**().
8. **Start processing I/O** on the accepted connection. Essentially, you want to post one or more asynchronous WSARecv() or WSASend() requests on the new socket using the overlapped I/O mechanism. When these I/O requests complete, **a worker thread services the I/O requests** and continues processing future I/O requests.
9. Repeat steps 5-8 until server terminates.

```
HANDLE CompletionPort;
WSADATA wsd;
SYSTEM_INFO SystemInfo;
SOCKADDR_IN InternetAddr;
SOCKET Listen;
int i;

typedef struct _PER_HANDLE_DATA
{
    SOCKET        Socket;
    SOCKADDR_STORAGE  ClientAddr;
    // Other information useful to be associated with the handle
} PER_HANDLE_DATA, *LPPER_HANDLE_DATA;

// Load Winsock
StartWinsock(MAKEWORD(2, 2), &wsd);

// Step 1:
// Create an I/O completion port
CompletionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);

// Step 2:
// Determine how many processors are on the system
GetSystemInfo(&SystemInfo);
```

```
// Step 3:
// Create worker threads based on the number of
// processors available on the system. For this
// simple case, we create one worker thread for each processor.
for (i = 0; i < SystemInfo.dwNumberOfProcessors; i++)
{
    HANDLE ThreadHandle;

    // Create a server worker thread, and pass the
    // completion port to the thread. NOTE: the
    // ServerWorkerThread procedure is not defined in this listing.
    ThreadHandle = CreateThread(NULL, 0, ServerWorkerThread, CompletionPort, 0, NULL;
    // Close the thread handle
    CloseHandle(ThreadHandle);
}

// Step 4:
// Create a listening socket
Listen = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);

InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(5150);
bind(Listen, (PSOCKADDR)&InternetAddr, sizeof(InternetAddr));

// Prepare socket for listening
listen(Listen, 5);
```

```
while (TRUE)
{
    PER_HANDLE_DATA *PerHandleData = NULL;
    SOCKADDR_IN saRemote;
    SOCKET Accept;
    int RemoteLen;

    // Step 5:
    // Accept connections and assign to the completion port
    RemoteLen = sizeof(saRemote);
    Accept = WSAAccept(Listen, (SOCKADDR *)&saRemote, &RemoteLen);

    // Step 6:
    // Create per-handle data information structure to associate with the socket
    PerHandleData = (LPPER_HANDLE_DATA)GlobalAlloc(GPTR, sizeof(PER_HANDLE_DATA));

    printf("Socket number %d connected₩n", Accept);
    PerHandleData->Socket = Accept;
    memcpy(&PerHandleData->ClientAddr, &saRemote, RemoteLen);

    // Step 7:
    // Associate the accepted socket with the completion port
    CreateIoCompletionPort((HANDLE)Accept, CompletionPort, (DWORD)PerHandleData, 0);

    // Step 8:
    //  Start processing I/O on the accepted socket.
    //  Post one or more WSASend() or WSARecv() calls on the socket using overlapped I/O.
    WSARecv(...);
}
```

```
DWORD WINAPI ServerWorkerThread(LPVOID lpParam)
{
    // The requirements for the worker thread will be discussed later.
}
```

Completion Ports & Overlapped I/O

- ✓ After associating a socket handle with a completion port, you can begin processing I/O requests by posting overlapped send and receive requests on the socket handle.
- ✓ Winsock API calls such as **WSASend**() and **WSARecv**() return immediately when called.
- ✓ It is up to your application to retrieve the results of the calls at a later time through an **OVERLAPPED** structure.
- ✓ In the completion port model, this is accomplished by having **one or more worker threads** wait on the completion port using the **GetQueuedCompletionStatus**() function

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

- **CompletionPort**: the completion port to wait on.
- **lpNumberOfBytesTransferred**: receives the number of bytes transferred after a completed I/O operation, such as WSASend() or WSARecv().
- **lpCompletionKey**: returns per-handle data for the socket that was originally passed into the CreateIoCompletionPort() function.
  - recommend **saving the socket handle** in this key.
- **lpOverlapped**: receives the WSAOVERLAPPED structure of the completed I/O operation
- **dwMilliseconds**: the number of milliseconds that the caller is willing to wait for a completion packet to appear on the completion port.

Per-handle Data and Per-I/O Operation Data

- ✓ The **lpCompletionKey** parameter contains what we call **per-handle data** because the data is related to a socket handle when a socket is first associated with the completion port.
- ✓ The **lpOverlapped** parameter contains an OVERLAPPED structure followed by what we call **per-I/O operation data**, which is anything that your worker thread will need to know when processing a completion packet.

```
typedef struct
{
    OVERLAPPED Overlapped;
    char       Buffer[DATA_BUFSIZE];
    int    BufferLen;
    int        OperationType;
} PER_IO_DATA;
```

```
PER_IO_OPERATION_DATA PerIoData;
WSABUF wbuf;
DWORD Bytes, Flags;

// Initialize wbuf ...
WSARecv(socket, &wbuf, 1, &Bytes, &Flags, &(PerIoData.Overlapped), NULL);

//Later in the worker thread, GetQueuedCompletionStatus() returns with an
//overlapped structure and completion key. To retrieve the per-I/O data the macro
//CONTAINING_RECORD should be used. For example,

PER_IO_DATA  *PerIoData=NULL;
OVERLAPPED   *lpOverlapped=NULL;

ret = GetQueuedCompletionStatus(
        CompPortHandle,
        &Transferred,
        (PULONG_PTR)&CompletionKey,
        &lpOverlapped,          INFINITE);

// Check for successful return
PerIoData = CONTAINING_RECORD(lpOverlapped, PER_IO_DATA, Overlapped);
```

Practice

✓ chapter05 → iocpServer project

I/O Model Consideration

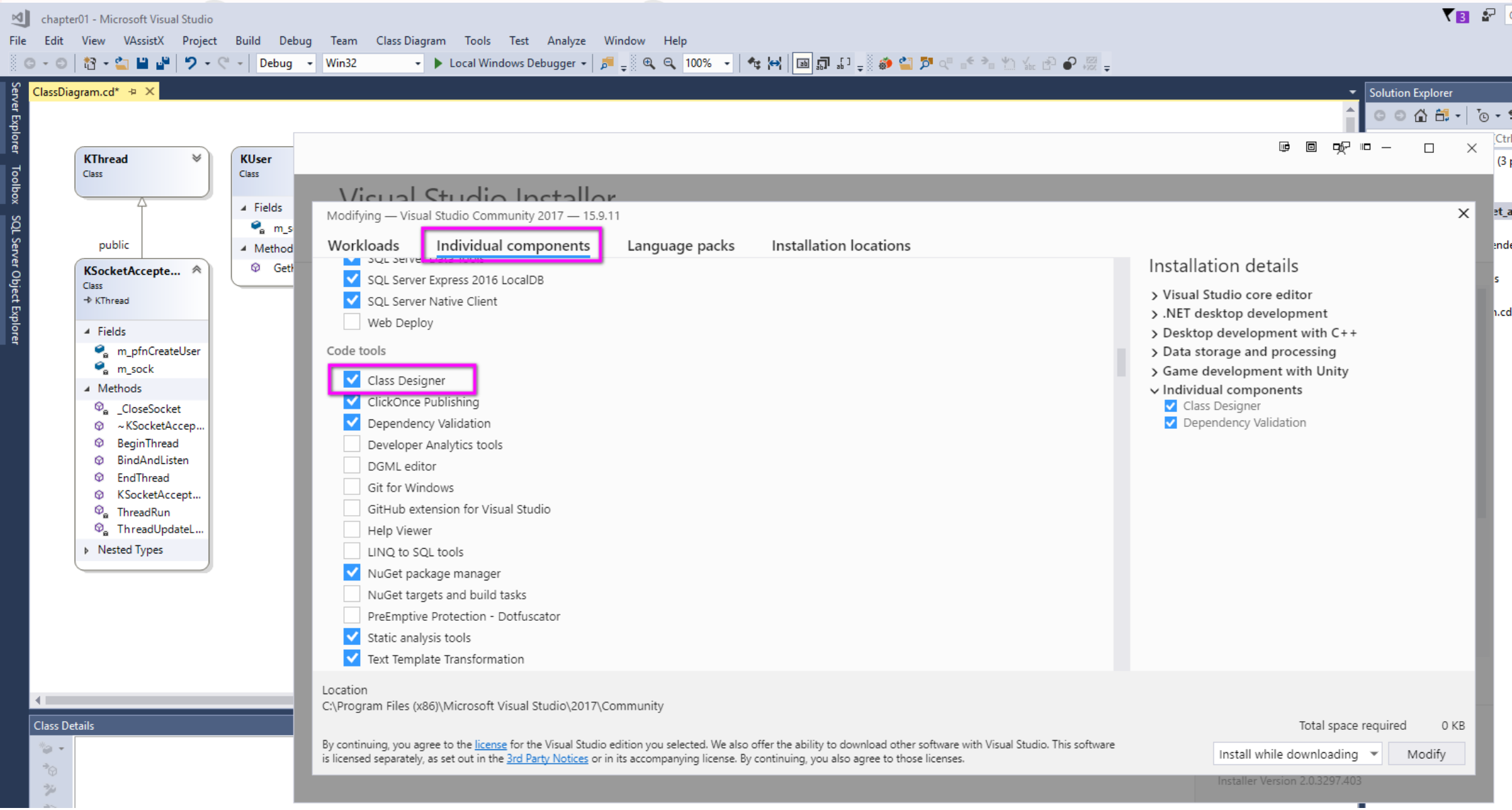
- ✓ When you are developing a **client** application that manages one or more sockets, we recommend using overlapped I/O or WSAEventSelect() over the other I/O models for performance reasons.
- ✓ When you are developing a **server** that processes several sockets at a given time, we recommend using overlapped I/O over the other I/O models for performance reasons.

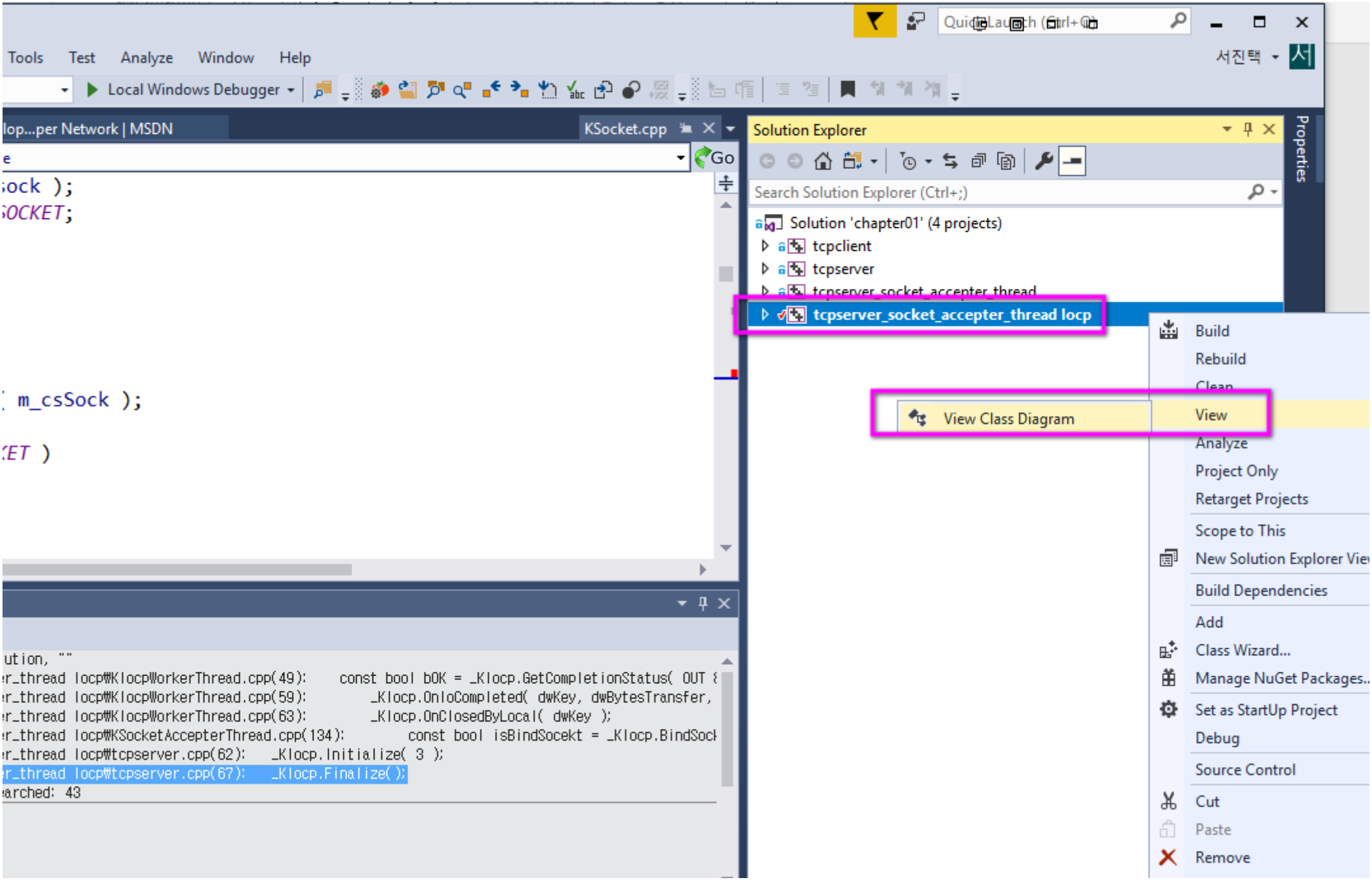
IOCP Server with Accepter Thread

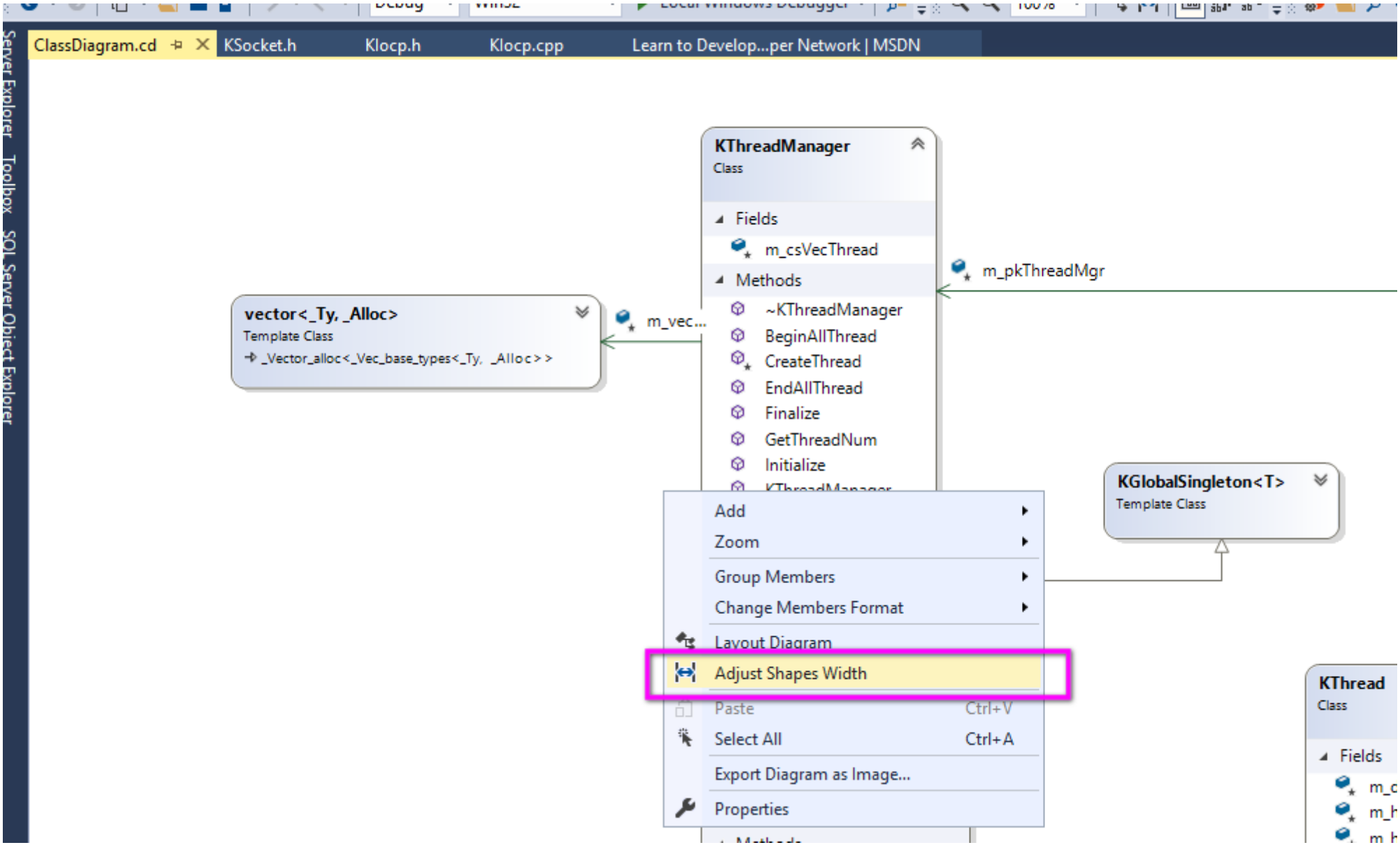
Chap1

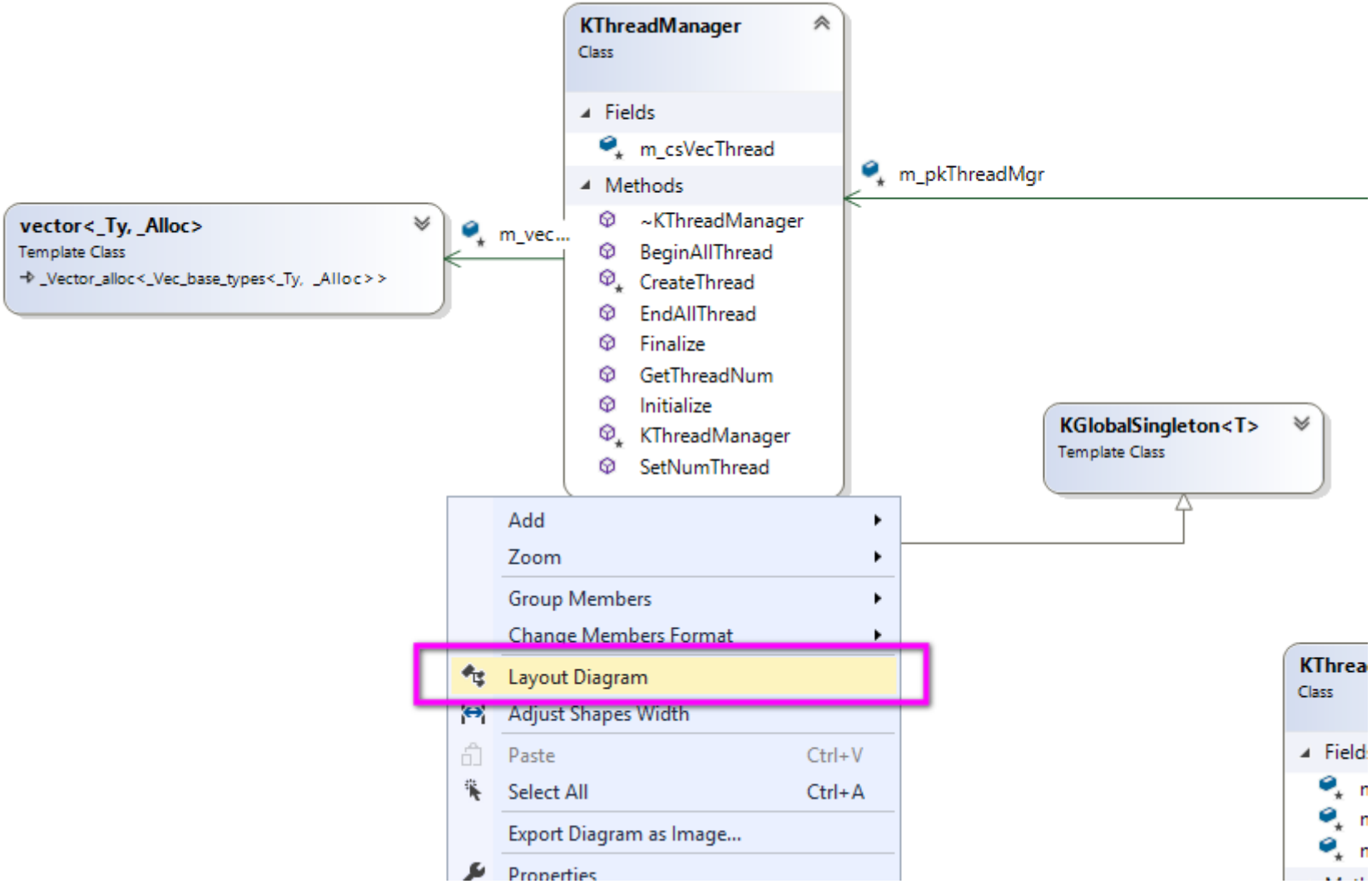
"tcpserver_socket_accepter_thread Iocp" project

Working with Class Designer in Visual Studio 2017

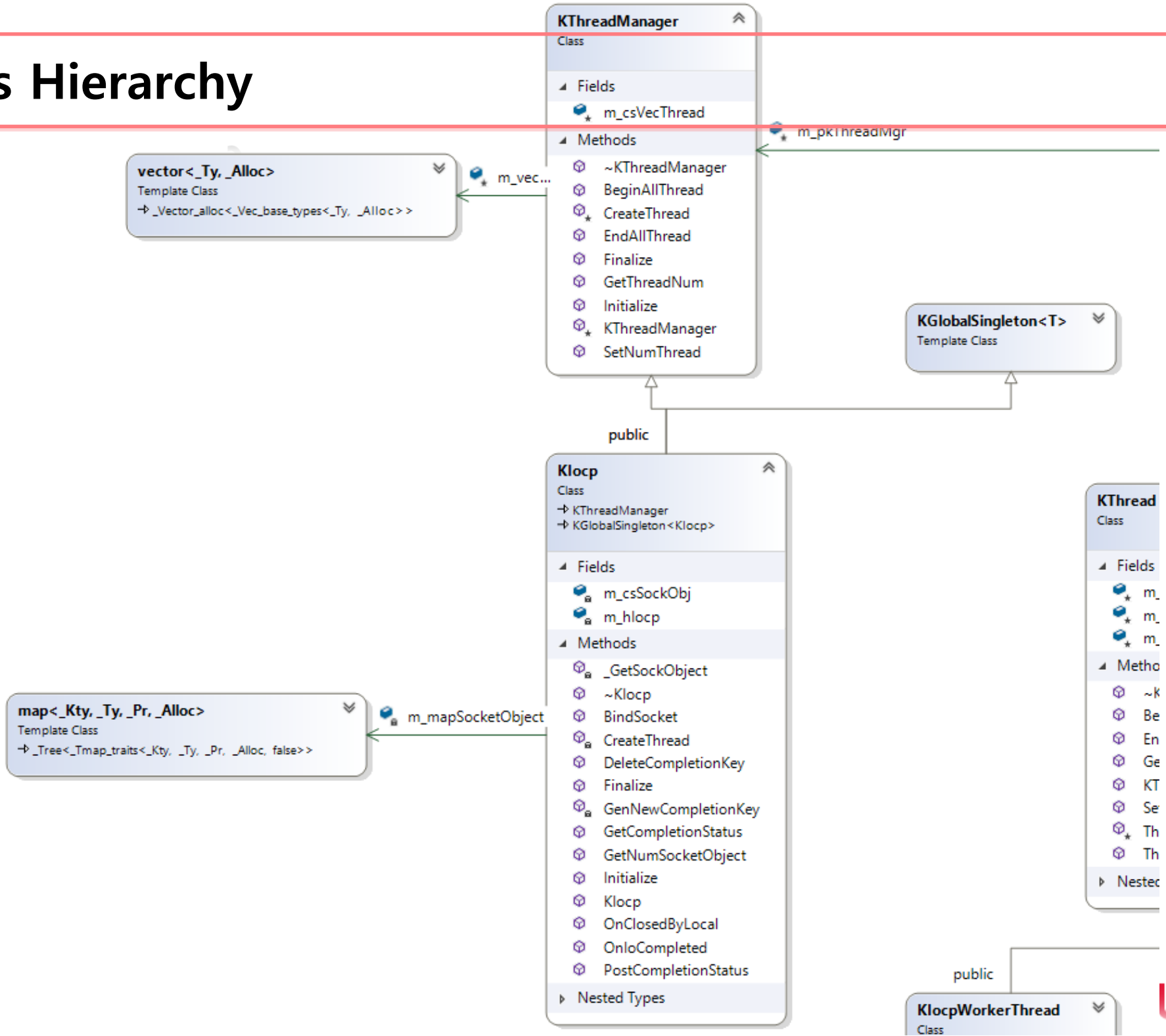


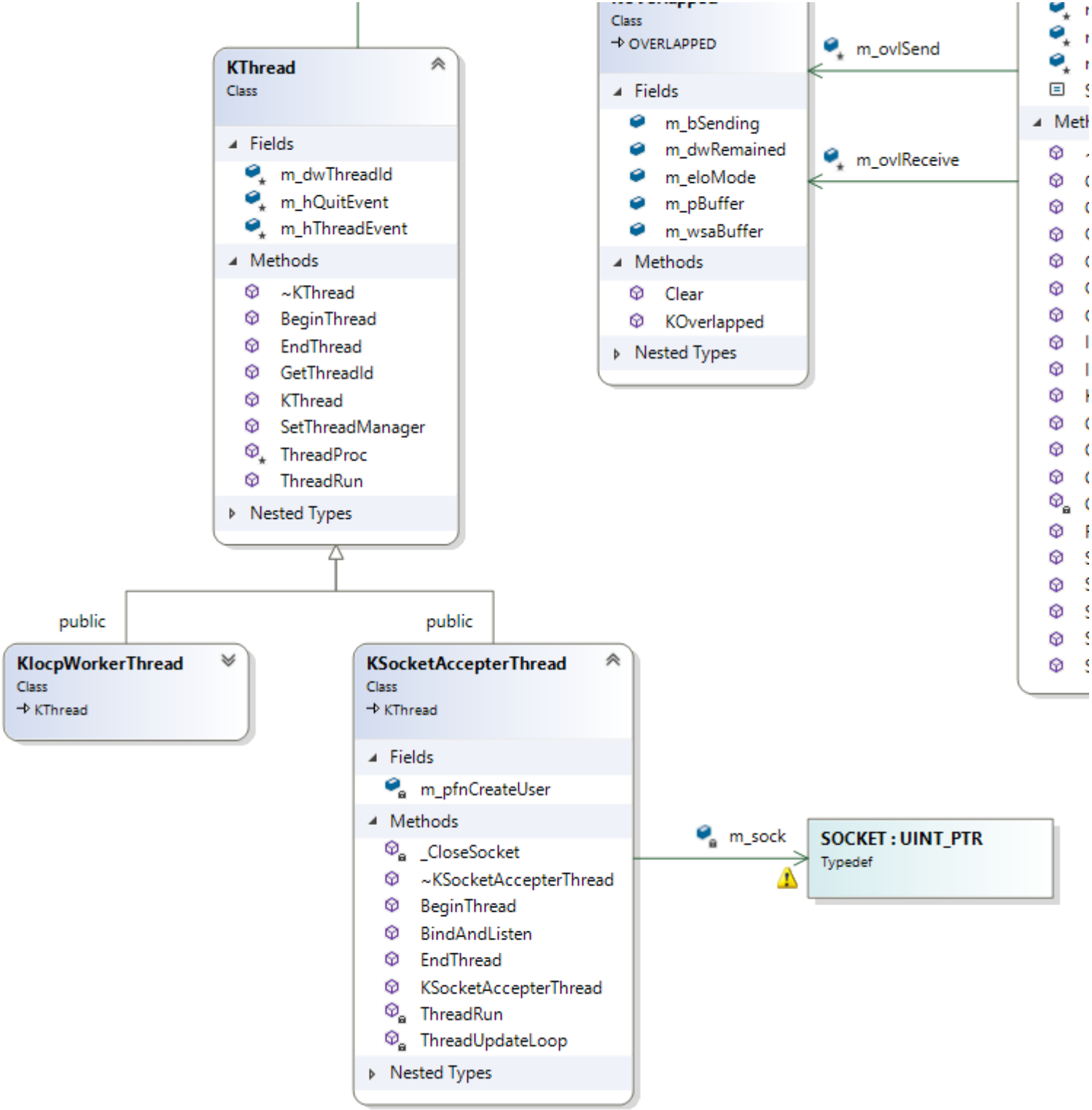

Tools
Test
Analyze
Window
Help
Local Windows Debugger
KSocket.cpp
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'chapter01' (4 projects)
tcpclient
tcpserver
tcpserver_socket_accepter_thread
tcpserver_socket_accepter_thread locp
Properties
View Class Diagram
Build
Rebuild
Clean
View
Analyze
Project Only
Retarget Projects
Scope to This
Build Dependencies
Add
Class Wizard...
Manage NuGet Packages...
Set as StartUp Project
Debug
Source Control
Cut
Paste
Remove

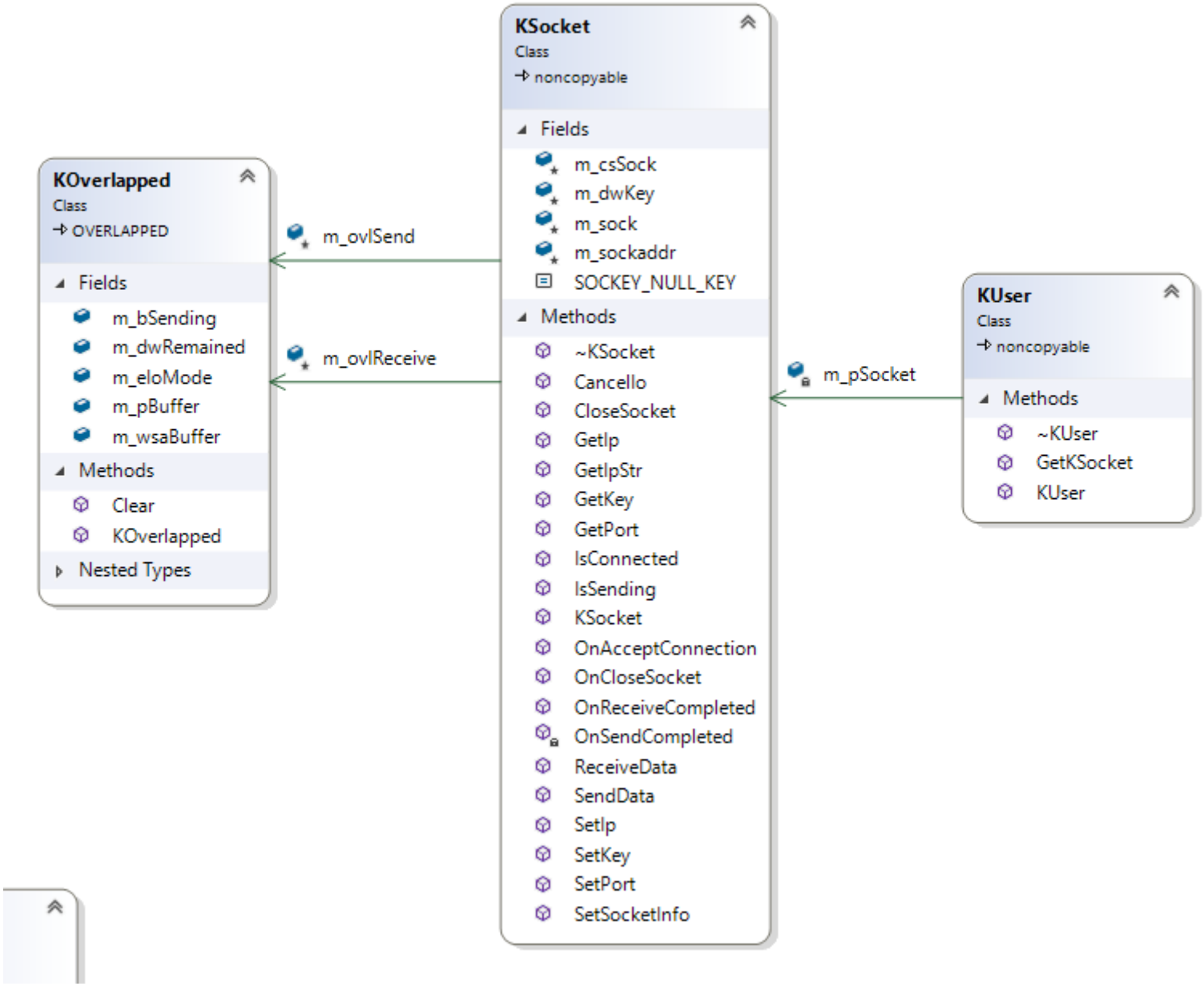

ClassDiagram.cd
KSocket.h
Klocp.h
Klocp.cpp
Learn to Develop...per Network | MSDN
Server Explorer
Toolbox
SQL Server Object Explorer
KThreadManager
Class
Fields
m_csVecThread
Methods
~KThreadManager
BeginAllThread
CreateThread
EndAllThread
Finalize
GetThreadNum
Initialize
m_pkThreadMgr
vector<_Ty, _Alloc>
Template Class
_Vector_alloc<_Vec_base_types<_Ty, _Alloc>>
m_vec...
KGlobalSingleton<T>
Template Class
Add
Zoom
Group Members
Change Members Format
Layout Diagram
Adjust Shapes Width
Paste Ctrl+V
Select All Ctrl+A
Export Diagram as Image...
Properties
KThread
Class
Fields


KThreadManager
Class
Fields
m_csVecThread
Methods
~KThreadManager
BeginAllThread
CreateThread
EndAllThread
Finalize
GetThreadNum
Initialize
KThreadManager
SetNumThread
m_pkThreadMgr
vector<_Ty, _Alloc>
Template Class
_Vector_alloc<_Vec_base_types<_Ty, _Alloc>>
m_vec...
KGlobalSingleton<T>
Template Class
Add
Zoom
Group Members
Change Members Format
Layout Diagram
Adjust Shapes Width
Paste Ctrl+V
Select All Ctrl+A
Export Diagram as Image...
Properties
KThrea
Class
Field

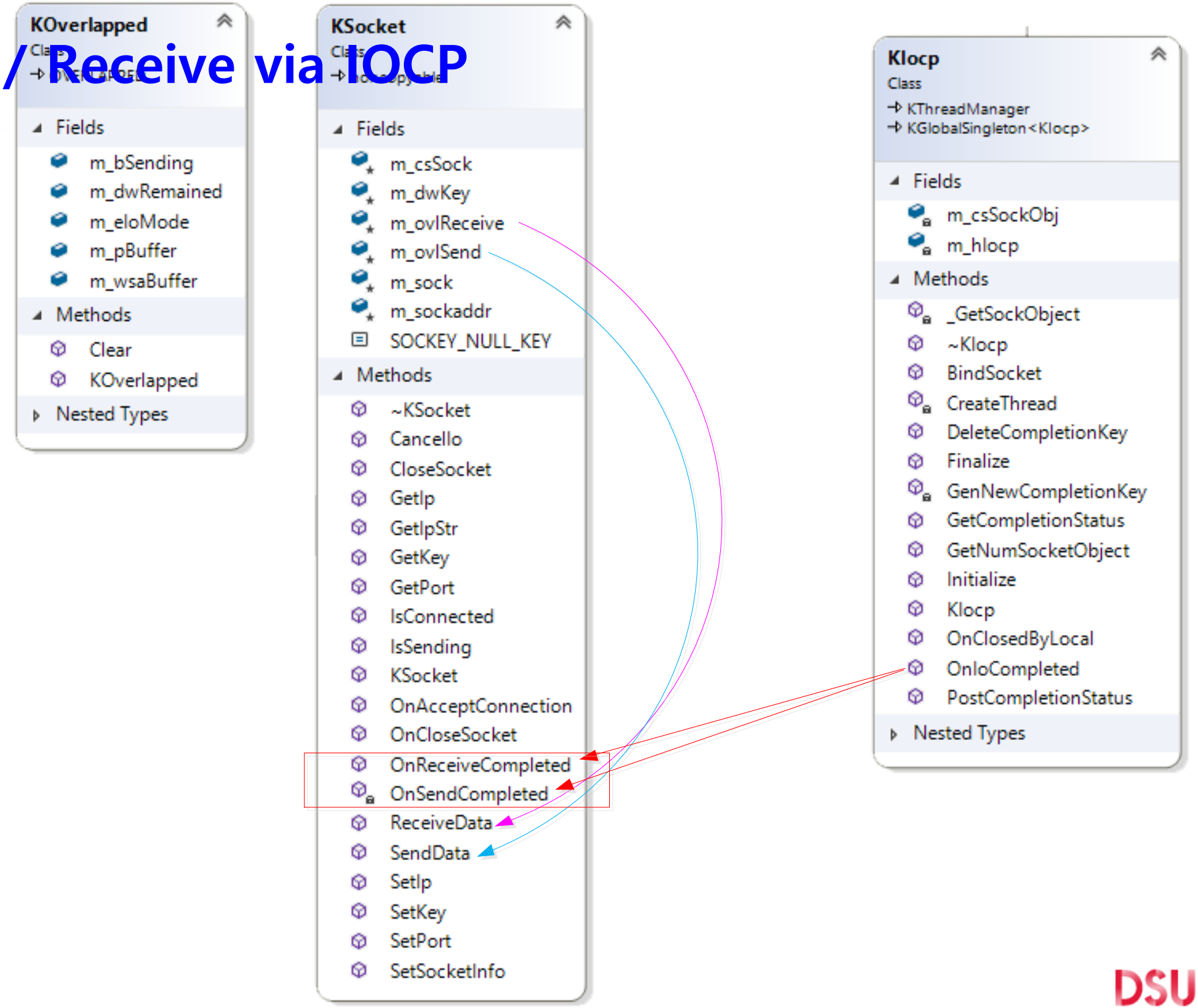
Class Hierarchy




KThread
Class
Fields
m_dwThreadId
m_hQuitEvent
m_hThreadEvent
Methods
~KThread
BeginThread
EndThread
GetThreadId
KThread
SetThreadManager
ThreadProc
ThreadRun
Nested Types
Class
OVERLAPPED
Fields
m_bSending
m_dwRemained
m_eloMode
m_pBuffer
m_wsaBuffer
Methods
Clear
KOverlapped
Nested Types
m_ovlSend
m_ovlReceive
public
public
KIocpWorkerThread
Class
KThread
KSocketAccepterThread
Class
KThread
Fields
m_pfnCreateUser
Methods
_CloseSocket
~KSocketAccepterThread
BeginThread
BindAndListen
EndThread
KSocketAccepterThread
ThreadRun
ThreadUpdateLoop
Nested Types
m_sock
SOCKET : UINT_PTR
Typedef


KSocket
Class
noncopyable
Fields
m_csSock
m_dwKey
m_sock
m_sockaddr
SOCKEY_NULL_KEY
Methods
~KSocket
Cancello
CloseSocket
GetIp
GetIpStr
GetKey
GetPort
IsConnected
IsSending
KSocket
OnAcceptConnection
OnCloseSocket
OnReceiveCompleted
OnSendCompleted
ReceiveData
SendData
SetIp
SetKey
SetPort
SetSocketInfo
KOverlapped
Class
OVERLAPPED
Fields
m_bSending
m_dwRemained
m_eloMode
m_pBuffer
m_wsaBuffer
Methods
Clear
KOverlapped
Nested Types
m_ovlSend
m_ovlReceive
KUser
Class
noncopyable
Methods
~KUser
GetKSocket
KUser
m_pSocket

Send / Receive via IOCP



KSocket::ReceiveData(), SendData()

```
bool KSocket::ReceiveData()
{
    KCriticalSectionLock lock( m_csSock );

    if( m_sock == INVALID_SOCKET )
        return false;

    DWORD dwRead = 0;
    DWORD dwFlag = 0;

    m_ovlReceive.Clear();
    m_ovlReceive.m_wsaBuffer.buf = &m_ovlReceive.m_pBuffer[m_ovlReceive.m_dwRemained];
    m_ovlReceive.m_wsaBuffer.len = MAX_PACKET_SIZE - m_ovlReceive.m_dwRemained;

    int ret = ::WSARecv( m_sock      // socket
        , &m_ovlReceive.m_wsaBuffer // buffer pointer, size
        , 1                         // number of WSABUF structure
        , &dwRead                   // numer of bytes after Io.
        , &dwFlag                   // [in,out] Option Flag
        , &m_ovlReceive             // struct LPWSAOVERLAPPED
        , NULL );                   // callback when Io completed.

    if( SOCKET_ERROR == ret ) {
        switch( ::GetLastError() ) {
        case WSA_IO_PENDING: // the overlapped operation has been successfully initiated
            return true;

        case WSAECONNRESET: // the virtual circuit was reset by the remote side.
```

```
bool KSocket::SendData( const char* szData_, int iSize_ )
{
    KCriticalSectionLock lock( m_csSock );

    // copy data to buffer.
    ::memmove( &m_ovlSend.m_pBuffer[m_ovlSend.m_dwRemained], szData_, iSize_ );
    m_ovlSend.m_dwRemained += iSize_;

    // when last WSASend() is not pending.
    if( m_ovlSend.m_bSending == false ) {
        DWORD dwWrite = 0;
        m_ovlSend.m_wsaBuffer.len = m_ovlSend.m_dwRemained;

        int ret = ::WSASend( m_sock, &m_ovlSend.m_wsaBuffer, 1, &dwWrite
            , 0, &m_ovlSend, NULL );

        bool isSendOk = false;
        if( ret != SOCKET_ERROR ) {
            isSendOk = true;
        }
        else {
            if( ::WSAGetLastError() == ERROR_IO_PENDING )
                isSendOk = true;
        }

        if( isSendOk == true ) {
            m_ovlSend.m_bSending = true;
            return true;
        }
```

KIocp::OnIoCompleted()

```
void KIocp::OnIoCompleted( DWORD dwKey_, DWORD dwBytesTransferred_, IN OVERLAPPED* pOverlapped_ )
{
    if( pOverlapped_ == NULL ) {
        BEGIN_LOG( cerr, L"IOCP Error" );
        return;
    }

    KOverlapped* pkOverlapped = static_cast<KOverlapped*>( pOverlapped_ );
    KSocket* pkSocket = _GetSockObject( dwKey_ );

…
    switch( pkOverlapped->m_eIoMode ) {
    case KOverlapped::IO_RECEIVE:
        if( &pkSocket->m_ovlReceive != pkOverlapped ) {
            BEGIN_LOG( cerr, "diffrent CompletionKey. (recv) key : " << dwKey_ );
        }
        VIRTUAL pkSocket->OnReceiveCompleted( dwBytesTransferred_ );
        break;
    case KOverlapped::IO_SEND:
        if( &pkSocket->m_ovlSend != pkOverlapped ) {
            BEGIN_LOG( cerr, "diffrent CompletionKey. (send) key : " << dwKey_ );
        }
        pkSocket->OnSendCompleted( dwBytesTransferred_ );
        break;
    }
}
```

KSocket::OnReceiveCompleted()

```
void KSocket::OnReceiveCompleted( DWORD dwTransferred )
{
    BEGIN_LOG( clog, "OnReceiveCompleted" )
        << END_LOG;

    if( dwTransferred == 0 ) {
        BEGIN_LOG( cout, L"closed socket: " )
            << LOG_NAMEVALUE( dwTransferred )
            << END_LOG;
        VIRTUAL OnCloseSocket();
        return;
    }

    KSocket::ReceiveData();
}
```

KSocket::OnSendCompleted()

```
void KSocket::OnSendCompleted( DWORD dwTransfered_ )
{
    KCriticalSectionLock lock( m_csSock );

    m_ovlSend.m_bSending = false;
    if( dwTransfered_ >= m_ovlSend.m_dwRemained ) {// sent all requested data
        m_ovlSend.m_dwRemained = 0;
        return;
    }

    int ret = SOCKET_ERROR;
    // there is remained data, send again
    m_ovlSend.m_dwRemained -= m_ovlSend.InternalHigh;
    memmove( &m_ovlSend.m_pBuffer[0], &m_ovlSend.m_pBuffer[dwTransfered_], m_ovlSend.m_dwRemained );

    DWORD dwWrite = 0;
    m_ovlSend.m_wsaBuffer.len = m_ovlSend.m_dwRemained;

    ret = ::WSASend( m_sock, &m_ovlSend.m_wsaBuffer, 1, &dwWrite, 0, &m_ovlSend, NULL );

    // Io is pending
    if( ret == SOCKET_ERROR && ::WSAGetLastError() == ERROR_IO_PENDING ) {
        m_ovlSend.m_bSending = true;
        return;
    }
    // all data was sent after WSASend().
    if( ret != SOCKET_ERROR ) {
        m_ovlSend.m_bSending = true;
        return;
    }
```

References

- ✓ http://www.madwizard.org/programming/tutorials/netcpp/5

MY **BRIGHT** FUTURE

DSU Dongseo University 동서대학교

DSU Dongseo Univ 동서대학교