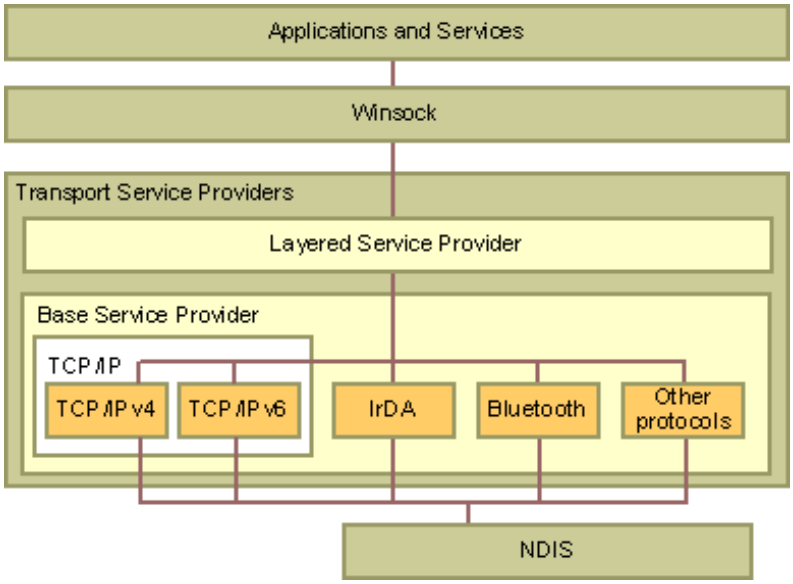
Network Programming(TCP Base) 기초

1 윈속(Winsock)

윈속(Winsock: Windows Socket)은 버클리 대학에서 개발한 유닉스 기반의 소켓(BSD: Berkeley Software Distribution Socket)을 윈도우 환경에 맞게 구성한 API입니다. 따라서 프로그램을 구동하기 위한 환경 설정을 제외하고 나머지 통신에 대한 기본적인 개념과 구조는 거의 같아서 코드의 호환성이 높습니다.

Winsock는 Windows 95 버전부터 API에 포함되어 현재 2.x 버전까지 만들어져 있습니다. 윈속은 윈도우 환경에 맞게 구성되어 있으므로 BSD 소켓과 다르게 몇 가지가 더 첨가 되었는데 먼저 윈속은 DLL로 제공이 되고, 윈속 DLL을 사용하기 위한 API가 존재합니다.

또한 윈도우 시스템은 멀티 테스킹 환경에서 프로세스의 이벤트 관리를 메시지 기반으로 처리하므로 네트워크에서 발생하는 이벤트들을 처리하는 확장 API 함수를 제공합니다.
윈속은 TCP/IP에 대해서 32비트 주소체계 뿐만 아니라 128비트(16바이트) IPv6(윈도우 XP 이상)도 지원이 되고 있으며 이외에 적외선 통신(IrDA: 윈도우 98 이상), Bluetooth(윈도우 XP SP2 이상), IPX/SPX 등 기타 프로토콜을 같은 수준(Level)의 API로 제공하고 있습니다.



<윈속2.2 구조: http://msdn.microsoft.com/en-us/library/ms885821.aspx>

1.1 윈속 초기화

우리는 윈속을 사용하기 위해서 WS2_32.DLL를 로드 해야 하고 초기화하는 방법을 알아야 합니다. 먼저 윈속 라이브러리 ws2_32.lib를 링크하고 winsock2.h 파일을 포함합니다.
winsock2.h 헤더 파일을 포함할 때 주의해야 하는데 이 헤더파일은 윈속에 대한 타입과 설정 등이 독립적으로 구성되어 있어서 windows.h 헤더 파일보다 앞에 두어서 재설정 등의 에러(error) 또는 충돌 경고(warning) 등을 방지합니다.

```
#pragma comment(lib, "ws2_32.lib")      //윈속 라이브러리 링크
...
#include <winsock2.h>                   // winsock2 헤더파일 포함
...
#include <windows.h>
```

WSAStartup 함수는 WS2_32.DLL을 사용할 수 있도록 로드하고 초기화하는 함수로서 첫 번째 인수에 윈속의 가장 높을 버전을 지정합니다.

```
int WSAStartup( WORD wVersionRequested, WSADATA* lpWSAData)
```

버전 설정은 MAKEWORD와 같은 매크로를 사용해서 "MAKEWORD(2, 2)" 와 같이 상위바이트는 major 버전을, 그리고 하위 바이트는 minor 버전을 2byte WORD로 구성합니다.
두 번째 인수 lpWSAData에 윈속 시스템 정보를 반환합니다. 따라서 WSADATA 구조체 변수는 {0}으로 초기화한 다음 사용하는 것이 좋습니다.
WSAStartup() 함수가 성공하면 숫자 '0'을 반환하고 이외에 실패할 경우 에러 코드를 반환합니다.

에러 코드	내용
WSASYSNOTREADY	네트워크 서브 시스템이 준비되지 않음
WSAVERNOTSUPPORTED	지원이 안 되는 버전
WSAEINPROGRESS	윈속 1.11 블로킹 처리가 진행 중
WSAEPROCLIM	윈속이 처리 가능한 한계에 도달함
WSAEFAULT	lpWSAData 인수가 유효하지 않은 포인터

< WSAStartup 함수 반환 값>

사용하려는 네트워크 프로토콜이 만약 컴퓨터에 설치되어 있는지 확인할 필요도 있습니다. 이 때 WSAEnumProtocols 함수를 사용하면 사용할 수 있는 트랜스포트 계층의 프로토콜에 대한 정보를 얻을 수 있습니다.

```
int WSAEnumProtocols(LPINT lpiProtocols
                     , LPWSAPROTOCOL_INFO lpProtocolBuffer
                     , ILPDWORD lpdwBufferLength)
```

함수의 첫 번째 인수 lpiProtocols은 사용 가능한 프로토콜의 배열입니다. 만약 사용자가 TCP, UDP를 사용할 수 있는지 묻고 싶다면 int형 배열을 만들고 이 배열에 IPPROTO_TCP, IPPROTO_UDP와 같은 값을 할당하고 전달합니다.
두 번째 인수는 프로토콜로 지원된 서비스들을 얻기 위한 WSAPROTOCOL_INFO 구조체 배열입니다.
세 번째 인수는 두 번째 인수 lpProtocolBuffer 버퍼에 대한 바이트의 개수를 반환합니다. 만약 첫 번째 인수와 두 번째 인수를 NULL 설정하면 사용할 수 있는 모든 프로토콜들의 정보를 이 변수에서 반환 받게 됩니다.
WSAEnumProtocols() 함수는 처리가 성공하면 사용할 수 있는 프로토콜의 개수를 반환합니다. 만약 실패 하면 SOCKET_ERROR가 반환되고, WSAGetLastError 함수를 호출하여 에러 코드를 확인합니다.

프로그림을 시작할 때 TCP, UDP를 사용할 수 있는지 WSAEnumProtocols() 함수로 확인 할 수 있습니다. 처음 버퍼의 크기를 정할 수 없으므로 WSAEnumProtocols() 함수의 첫 번째와, 두 번째 인수는 NULL로 설정하고 세 번째 인수만 프로토콜 반환 버퍼의 크기를 받을 주소를 전달합니다. 물론 NULL 인수를 첫 번째, 두 번째 인수에 전달했기 때문에 SOCKET_ERROR를 반환할 것입니다.

```
// 사용할 수 있는 프로토콜의 크기 얻기
// 첫 번째, 두 번째 인수가 NULL로 설정되어 있어 결과는 Error가 되어야 함.
hr = WSAEnumProtocols(NULL, NULL, &dProtoBuf);
int nCnt = dProtoBuf/sizeof(WSAPROTOCOL_INFO);
if(SOCKET_ERROR != hr || 0 == nCnt)
        return -1;
...
proto[0] = IPPROTO_TCP;
proto[1] = IPPROTO_UDP;

// 프로토콜을 사용할 수 있는지 확인
hr = WSAEnumProtocols(proto, protoInf, &dProtoBuf);
...
if(SOCKET_ERROR == hr)
        return -1;
```

네트워크는 파일 입출력과 다르게 물리적으로 거리가 있는 데이터를 주고 받기 때문에 네트워크에

서 발생하는 수많은 문제들을 해결해야 합니다. 따라서 작업을 요청하고 나서 반드시 어떤 문제점이 있는지 확인해야 합니다.

```
int WSAGetLastError()
```

WSAGetLastError() 함수는 네트워크에서 가장 최근에 일어난 소켓의 에러 코드를 얻어내는 함수입니다. 우리는 이 함수를 가지고 소켓에서 발생한 에러의 종류를 파악하고 대처합니다.
주의할 것은 소켓이 비동기(Asynchronies) 입출력을 하고 있다면 소켓은 처리 결과를 SOCKET_ERROR로 종종 반환할 수도 있습니다.
예를 들어 WSAAsyncSelect 모델로 클라이언트의 소켓이 만들어지고 서버 커넥션(connection)을 요청할 때 결과는 메시지 처리 함수에서 처리하기 때문에 바로 SOCKET_ERROR를 반환 합니다. 이 때 반드시 WSAGetLastError()를 사용해야 합니다.

```
hr = connect(g_scHost, (SOCKADDR*)&g_sdHost, sizeof(g_sdHost));
if(SOCKET_ERROR == hr)
{
        hr = WSAGetLastError();
        if(WSAEWOULDBLOCK != hr)
                return 0;
}
```

사실 모든 네트워크 에러를 확인 하는 코드를 작성하는 것이 무척 중요합니다.
WSAGetLastError() 함수로 에러 코드를 얻었으면 이 에러 코드를 FormatMessage() 함수를 사용해서 문자열로 출력할 수 있습니다.

```
void LogGetLastError()
{
        int hr = WSAGetLastError();
        char* lpMsgBuf;
        FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM
                        | FORMAT_MESSAGE_IGNORE_INSERTS
                        , NULL, hr, 0, (LPSTR)&lpMsgBuf, 0, NULL );
        printf( "%sWn", lpMsgBuf);
        LocalFree( lpMsgBuf );
}
```

1.2 소켓

윈속 DLL을 사용할 수 있다면 네트워크 통신의 주체인 소켓 기술자(socket descriptor. 이하 소켓)를 생성합니다. 소켓은 BSD socket() 함수 또는 WSASocket() 함수로 생성합니다. 일반적으로 socket() 함수를 자주 사용하며 이 함수는 중첩 입출력(overlapped I/O)가 가능한 소켓을 생성하는데 만약 소켓을 만들 수 없다면 INVALID_SOCKET을 반환합니다. WSASocket() 함수는 socket() 함수보다 윈도우 환경에 맞게 좀더 좀 더 다양한 지원이 되는 함수인데 여기서는 자세한 설명을 생략하겠습니다.

```
SOCKET socket(int af, int type, int protocol),
SOCKET WSASocket (int af, int type, int protocol
                LPWSAPROTOCOL_INFO lpProtocolInfo, GROUP g, DWORD dwFlags)
```

```
// TCP 소켓(socket descriptor) 생성 방법
SOCKET scHost = 0;
if(INVALID_SOCKET == (scHost = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)))
        return -1;

// 같은 종류의 소켓입니다.
scHost = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, 0, WSA_FLAG_OVERLAPPED);

...
// UDP 소켓(socket descriptor) 생성 방법
if(INVALID_SOCKET == (scHost = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)))
        return -1;
```

윈속은 네트워크 통신 뿐만 아니라 NetBIOS, Bluetooth, 적외선 등에서도 사용할 수 있습니다. 이들은 AF(Address Family, 또는 PF: Protocol Family)로 구분을 하고 socket() 함수의 첫 번째 인수에 AF를 설정합니다. TCP, UDP를 사용하려면 AF에 'AF_INET' 상수를 설정합니다.
두 번째 인수 type은 AF에 대한 소켓의 통신 타입을 설정하는 것입니다. 통신 타입은 크게 전화처럼 두 통신 주체가 미리 연결된 상태에서 통신을 처리하는 연결 지향형(connection-oriented)과 방송이나 무전과 같이 연결을 신경 쓰지 않고 일방적으로 통신하는 비연결 지향형(message-oriented)으로 구분합니다. 우리의 관심은 TCP, UDP에 있으므로 TCP에서는 type를 SOCK_STREAM으로 UDP에서는 SOCK_DGRAM으로 설정합니다.
세 번째 인수 protocol은 소켓의 프로토콜입니다. 현재의 윈속은 WOSA(Windows Open Services Architecture)에 맞게 제작되어 응용프로그램 제작자들을 위한 API 와 Transport stack 을 제작하는 제작사들을 위한 SPI(Service Provider Interface)을 제공합니다. 이 세 번째 인수의 값을 숫

자 '0'으로 설정하면 DLL을 호출하는 Caller는 서비스 공급자(Service Provider, 또는 Device Service Provider)가 사용하는 프로토콜을 선택하게 됩니다. 명시적으로 소켓 타입이 TCP이면 'IPPROTO_TCP', UDP는 'IPPROTO_UDP'를 사용합니다.

socket() 함수는 소켓 생성을 실패하게 되면 INVALID_SOCKET 상수를 반환합니다. 이 때 WSAGetLastError() 함수로 에러 상태를 확인 하고 이들 내용에 따라 적절한 코드를 추가 하도록 합니다.

에러 코드	내용
WSANOTINITIALISED	WSAStartup 함수의 성공적인 호출이 없이 사용.
WSAENETDOWN	네트워크 서브시스템 또는 service provider 에러
WSAEAFNOSUPPORT	지원 되지 않는 어드레스 패밀리 지정. 예를 들어 컴퓨터에 하드웨어 또는 드라이버가 설치되지 않은 경우
WSAEINPROGRESS	블로킹 윈속 1.1 이 진행 중, 또는 service provider 가 callback 함수를 처리 중
WSAEMFILE	사용할 수 있는 소켓이 없음
WSAENOBUFS	사용할 수 있는 버퍼가 없어 소켓을 생성할 수 없음
WSAEPROTONOSUPPORT	지원 안 되는 프로토콜
WSAEPROTOTYPE	소켓에 맞지 않거나 잘못된 타입 설정
WSAESOCKTNOSUPPORT	어드레스 패밀리에서 지원 안 되는 소켓 타입 설정
WSAEINVAL	- 이미 바인딩 된 소켓 다음 조건에서 적어도 한 개 이상의 문제가 있을 때 에러 발생 - 인수 'GROUP g'가 유효하지 않음 - WSAPROTOCOL_INFO 변수 내용이 유효하지 않거나 이미 사용되어 복제된 소켓 처리에서 사용되어 완전하지 않은 포인터가 됨 - AF, Type, Protocol 개별 내용은 유효하나 조합했을 때 지원이 안됨
WSAEFAULT	WSAPROTOCOL_INFO 변수의 주소 처리 영역이 유효하지 않음
WSAINVALIDPROVIDER	2.2 과 다른 Service provider 버전
WSAINVALIDPROCTABLE	Service provider가 유효하지 않거나 WSPStartup() 함수로 완전하지 않은 프로시저 테이블을 리턴 했음.

<소켓에 대한 에러 코드>

소켓은 시스템의 자원으로 더 이상 소켓을 사용하지 않으면 자원을 반환(혹은 해제)해야 합니다.

```
int closesocket(SOCKET s)
```

closesocket() 함수는 파일의 close() 함수와 비슷하게 시스템 자원인 소켓 디스크립터를 시스템에 반환(해제)합니다.

```
// 소켓(socket descriptor)
SOCKET  scHost=0;
…
// TCP 소켓(socket descriptor) 생성
if(INVALID_SOCKET == (scHost = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)))
…
// 소켓 해제
closesocket(scHost);
```

지금까지 윈속과 소켓을 알아보았습니다. 윈속의 초기화/해제, 그리고 TCP/UDP 환경의 유효성, 소켓의 생성/해제를 종합해서 코드 다음과 같이 작성할 수 있습니다. 전체 코드는 nw01_socket.zip 에 있으며 만약 이 프로그램을 실행했을 때 "The winsock environment is OK" 문장이 출력이 되면 사용하고 있는 PC 환경이 TCP/UDP 기반 응용 프로그램을 실행하기에 충분함을 의미합니다.

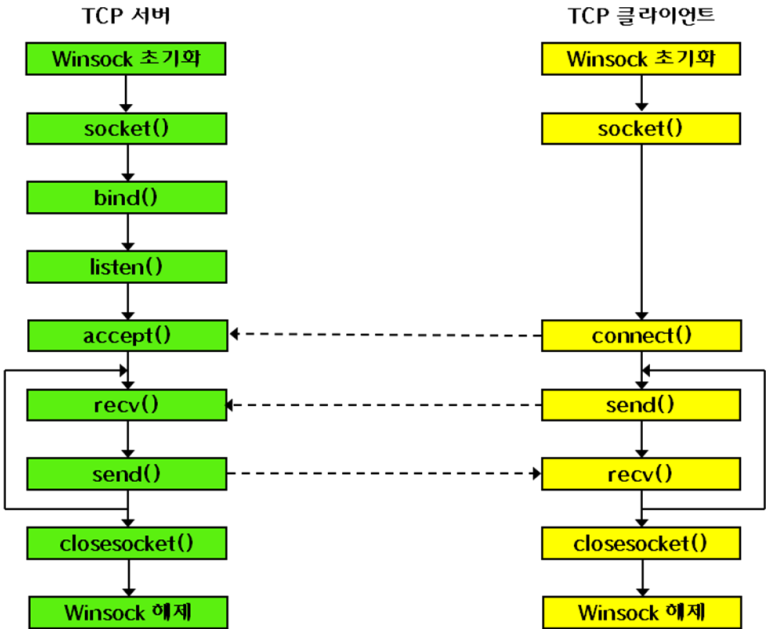
```
#pragma comment(lib, "ws2_32.lib")
#include <winsock2.h>
#include <windows.h>
#include <stdio.h>

int main()
…
WSADATA wsData={0};

// Load WinSock DLL
hr = WSAStartup(MAKEWORD(2, 2), &wsData);
if(0 != hr)
        return -1;

DWORD dProtoBuf=0;
```

```
// 첫 번째, 두 번째 인수가 NULL로 설정 되어 있어 단순히 크기만 얻고
// 함수 호출 자체는 Error가 되어야 함.
hr = WSAEnumProtocols(NULL, NULL, &dProtoBuf);
int nCnt = dProtoBuf/sizeof(WSAPROTOCOL_INFO);
if(SOCKET_ERROR != hr || 0 == nCnt)
        return -1;

// TCP, UDP를 사용할 수 있는지 확인
int*              proto    = new int[nCnt];
WSAPROTOCOL_INFO* protoInf = new WSAPROTOCOL_INFO[nCnt];
memset(proto   , 0, nCnt * sizeof(int) );
memset(protoInf, 0, nCnt * sizeof(WSAPROTOCOL_INFO) );
proto[0] = IPPROTO_TCP;
proto[1] = IPPROTO_UDP;

hr = WSAEnumProtocols(proto, protoInf, &dProtoBuf);
delete [] proto;
delete [] protoInf;
if(SOCKET_ERROR == hr)
        return -1;

// TCP socket descriptor 생성
SOCKET scHost = 0;
scHost = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if(INVALID_SOCKET == scHost)
        return -1;

// release socket descriptor
closesocket(scHost);

// Unload WinSock DLL
WSACleanup();

printf("The winsock environment is OK.Wn");
...
```

<nw01_socket.zip>

1.3 TCP Server 프로그래밍

간단히 기본적인 TCP 기반 서버와 클라이언트의 동작을 보면 먼저 서버가 운영이 되면서 listen 소켓으로 클라이언트의 접속을 기다립니다. 클라이언트는 소켓을 생성하고 이 소켓으로 서버와 접속합니다. 서버는 접속을 허락하고 클라이언트와 통신할 새로운 소켓을 생성합니다. 생성된 소켓과 클라이언트와 데이터를 송신/수신을 하고 접속을 끊은 후 소켓을 종료합니다. 클라이언트는 소켓을 종료하면 다시 소켓을 만들고 프로그램을 진행해야 합니다. 하지만 서버의 경우 클라이언트와 송수신할 소켓만 종료했기 때문에 다른 클라이언트의 접속을 기다릴 수 있습니다. 이러한 과정을 관련 함수들과 나열해서 다음과 같이 그림으로 표현할 수 있습니다.



<서버와 클라이언트 동작과 관련 함수>

서버의 동작을 프로그램 하기 위해 순서를 다시 정리한다면 다음과 같습니다.

1) 윈속 라이브러리 초기화
2) 클라이언트의 접속(Accept)을 담당하는 Listen 소켓 생성
3) 소켓에 주소와 포트를 연결하는 Binding
4) Listen 상태
5) 클라이언트의 접속 Accept → 클라이언트와 데이터 송/수신을 위한 소켓 생성
6) 클라이언트와 데이터 송수신
7) 소켓 Close
8) 윈속 해제

socket()함수를 사용해서 접속에 대한 listen 소켓을 만들었다면 이 소켓에 아이피 주소(이하 아이피. IP Address: Internet Protocol Address), 포트(Port)와 소켓을 결합(binding) 합니다. 아이피 주소는 논리적인 주소로 네트워크에서 통신을 하기 위해서 사용하는 구별 가능한 고유한 번호입니다. 마치 주민번호와 같다고 할 수 있습니다. 차이는 주민번호는 한 번 받으면 바꾸기가 어렵지만 아이피는 쉽게 바꿀 수 있습니다. 쉽게 바꿀 수 있다 하더라도 유효한 아이피만 네트워크 통신이 가능하고, 라우터는 이 아이피를 관리해서 인터넷에 연결하도록 도와줍니다.

```
int bind(SOCKET s, const struct sockaddr* name, int namelen)
```

bind() 함수는 아이피, 포트를 소켓에 연결하며 실패하면 SOCKET_ERROR를 반환합니다. 에러 WSAGetLastError() 함수로 에러 코드를 확인하면 다음과 같습니다.

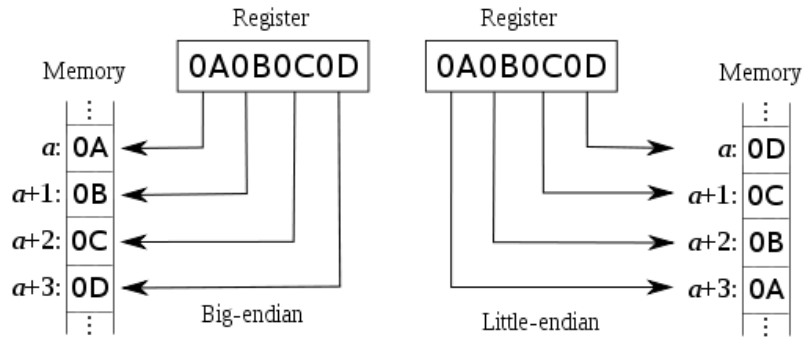
에러 코드	내용
WSAEACCES	옵션 SO_BROADCAST을 활성화 되지 않은 상태에서 접근이 허가되지 않은 방식으로 데이터그램(Datagram) 소켓 접근
WSAEADDRINUSE	소켓-주소(프로토콜/네트워크 주소/포트)는 하나만 사용할 수 있음 다른 소켓이 사용하는 주소를 새로운 소켓이 바인딩 할 때, 소켓이 closesocket() 함수를 호출 했어도 소켓이 완전히 해제되지 않은 상태에서 bind() 함수를 호출함. 소켓 옵션에서 재사용 SO_REUSEADDR 옵션을 적용
WSAEADDRNOTAVAIL	유효하지 않은 주소
WSAEFAULT	주소 체계와 주소가 잘못 또는 일치하지 않거나 namelen의 값이 작음.
기타	WSANOTINITIALISED, WSAENETDOWN, WSAENOBUFS, WSAEINPROGRESS, WSAEINVAL -> 소켓 생성 참조

<bind()함수 에러 코드>

아이피 주소와 포트는 sockaddr 구조체 변수에 저장을 합니다. 현재 아이피 주소는 32비트(4byte) 주소체계와 128비트(16byte) 주소체계 두 종류가 있습니다. 이들을 각각 IP version 4 주소(IPv4), IP version 16 주소(IPv6) 부르며 16byte 주소체계는 현재 4byte 주소 체계가 가지는 한계를 넘기 위해서 지구상의 모든 기기에 전부 번호를 부여할 수 있을 정도로 거의 무한대의 번호를 지정할 수 있습니다.
4byte 아이피 주소와 포트를 저장하기 위해서 sockaddr 대신 sockaddr_in 구조체를 주로 사용합니다. 이 둘의 크기는 같지만 sockaddr_in 구조체는 포트, 아이피를 저장할 수 있는 필드가 존재해서 사용하기가 무척 편리합니다.

```
struct sockaddr_in {
   short sin_family;            // 주소 패밀리(Address Family)
   unsigned short sin_port;     // 포트 번호(빅-엔디언)
   struct in_addr sin_addr;     // 아이피 번호(빅-엔디언)
   char sin_zero[8];            // 사용되지 않는 필드. 0으로 설정
};
```

IPv6 버전 sockaddr_in6은 구조체의 필드 타입만 다르고 구조는 IPv4와 같습니다. TCP/IP를 사용하기 위해서 sin_famiy는 AF_INET(또는 PF_INET) 상수를 사용합니다.
참고로 네트워크에 대한 정보를 채우기 위해서 우리는 바이트 순서(Byte-order), 즉, 엔디언(Endian)를 알고 있어야 합니다. 엔디언은 크게 빅-엔디언(big-endian), 리틀-엔디언 두 가지와 이 둘을 혼용한 미들-엔디언(Middle-endian)으로 구분합니다. 빅 엔디언은 사람이 숫자를 쓰는 방법처럼 바이트 단위로 값을 저장할 때 바이트 열에서 인덱스의 순서가 큰 쪽에서 작은 쪽으로 저장되는 방법입니다. 리틀-엔디언(little-endian)은 이와 반대로 인덱스가 작은 쪽에서 큰 쪽으로 저장되는 것을 의미합니다.



<빅-엔디언, 리틀-엔디언의 데이터 저장 순서>

예를 들어 0xABCDEF12 값을 4바이트 배열에 저장한다고 한다면 빅 엔디언은 0xAB, 0xCD, 0xEF, 0x12로 저장되지만 리틀-엔디언은 0x12, 0xEF, 0xCD, 0xAB로 저장합니다.
인텔 계열 프로세서는 리틀-엔디언을 사용하고 있으며 PPC(Power PC) 계열, ARM 등의 RISC 기반 칩셋 중에서 빅 엔디언을 사용합니다.
참고로 인텔의 주장을 인용한다면 주로 값이 작은 정수 값들의 사용빈도가 높기 때문에 이들의 접근을 빠르게 하기 위해 앞쪽에 배열하는 것이 성능에 좋다고 하지만 이 둘의 성능 차이는 거의 없습니다.

네트워크에서 "168.122.245.1"과 같은 주소를 해석할 때 빅-엔디언이 유리한 점이 있는데 네트워크 통신의 프로토콜(Protocol)들이 빅-엔디언 방식을 사용하기 때문에 우리는 빅-엔디언으로 변경

해주는 함수들을 사용해서 데이터 정보를 채워야 합니다.
sockaddr_in 필드의 sin_port 필드는 포트를 저장하는 16비트 필드로 빅-엔디언 16비트 short형으로 변경해주는 htons() 함수를 사용해서 포트를 저장합니다.

sin_addr는 32비트 아이피 주소를 저장하는 필드 입니다. 만약 아이피 주소를 "163.32.78.56" 같은 문자열로 가지고 있다면 inet_addr() 함수를 사용해서 32비트 값을 얻어 냅니다. 참고로 inet_addr() 함수와 반대로 32비트에서 문자열을 반환하는 함수 inet_ntoa()도 기억하기 바랍니다.

서버는 하나 혹은 다수의 아이피를 사용하는 경우도 있습니다. 이 때 모든 아이피에 대해서 같은 포트로 접속을 허가한다면 INADDR_ANY를 사용하면 편리합니다. 이 상수 값도 빅-엔디언으로 바꿔야 되며 long형에 대한 htonl() 함수를 사용해서 htonl (INADDR_ANY) 로 지정합니다.

```
SOCKET scListen=0;      SOCKADDR_IN sdListen={0};
...
// 소켓에 주소, 포트를 바인딩
memset(&sdListen, 0, sizeof(SOCKADDR_IN));
sdListen.sin_family      = AF_INET;
sdListen.sin_addr.s_addr = htonl(INADDR_ANY);   // 또는 = inet_addr("IP ADDRESS");
sdListen.sin_port        = htons( atoi("20000") );
if(SOCKET_ERROR  == bind(scListen, (SOCKADDR*)&sdListen, sizeof(SOCKADDR_IN))
        return -1;
```

sockaddr_in 필드의 sin_port 필드는 포트를 저장하는 16비트 필드로 0~65535까지 사용할 수 있으며 빅-엔디언 16비트 short형으로 변경해주는 htons() 함수를 사용해서 포트를 저장합니다.

포트를 지정할 때 주의할 점이 있는데 TCP/IP 기반 위에 만들어진 프로토콜들은 전부 포트번호가 있고 이들은 인터넷 할당 번호 관리기관(IANA: Internet Assigned Numbers Authority)에서 권고하는 번호를 사용하는 것이 좋습니다.

일반적으로 포트는 "잘 알려진"(well-known: 0~1023), "등록된"(registered: 1024~49151), 그리고 "동적"(dynamic: 49152~65535) 포트로 구분합니다. "잘 알려진" 포트는 21-FTP, 23-TELNET, 25-SMTP, 80-HTTP 등과 같이 주로 TCP/IP 기반 위의 프로토콜이 사용하는 포트입니다.

응용프로그램에서는 "잘 알려진" 포트는 무조건 피하고 "등록 된" 포트 중에서 사용하지 않는 포트나 동적 포트를 사용하는 것이 좋습니다.

서버가 클라이언트의 접속을 담당하는 소켓을 만들고 아이피, 포트를 할당했다면 접속을 기다리는

상태로 만들어야 합니다.

```
int listen(SOCKET s, int backlog)
```

listen 함수는 외부로부터 들어오는 접속을 감지하기 위해 특정 소켓(listen socket)이 사용될 것임을 시스템에게 알려주는 함수로 첫 번째 인수는 소켓, 두 번째 인수는 대기하는 접속을 위한 큐의 최대 길이를 명시하며 일반적으로 상수 'SOMAXCONN'로 명시해서 기본 서비스 제공자 (service provider)가 최대 큐의 길이를 설정하도록 합니다. 함수의 호출이 실패하면 SOCKET_ERROR를 반환합니다.

에러 코드	내용
WSAEADDRINUSE	소켓의 로컬 주소가 이미 사용 중이고, SO_REUSEADDR 옵션으로 설정되어 있지 않음
WSAEISCONN	소켓이 이미 접속된(connected) 상태
WSAEMFILE	유효한 소켓이 없음
WSAENOTSOCK	소켓이 아님
WSAEOPNOTSUPP	listen 상태를 지원하는 소켓이 아님
기타	WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, WSAEINVAL, WSAENOBUFS -> 소켓 생성 참조

<listen() 함수 에러 코드>

```
if( SOCKET_ERROR == listen(scLstn, SOMAXCONN))
        return -1;
```

클라이언트의 접속을 담당하는 소켓을 'listen' 상태로 만들고 나서 클라이언트가 접속할 때까지 기다리고 접속하면 통신할 수 있는 소켓을 만들어야 합니다.

```
SOCKET accept(SOCKET s, struct sockaddr FAR* addr, int FAR* addrlen)
```

accept() 함수는 첫 번째 인수에 listen socket을 받아서 클라이언트가 접속하면 접속한 클라이언트와 데이터를 송신/수신할 수 있는 소켓을 반환합니다. 함수의 실패는 유효하지 않은 소켓 상수 INVALID_SOCKET 를 반환하고 이 때 WSAGetLastError() 함수를 사용에서 정확한 에러를 확인하고 대처합니다.

에러 코드	내용
WSAEFAULT	addrlen 인수가 작거나 유효하지 않은 addr 인수
WSAEINTR	blocking 윈속 1.1 호출이 WSACancelBlockingCall() 함수로 취소됨
WSAEOPNOTSUPP	연결 지향형 (SOCK_STREAM)이 아닌 소켓을 참조하고 있음
WSAEWOULDBLOCK	소켓이 nonblocking으로 표시되고, 접속이 허용된 어떤 연결(connection)도 없음.
WSAEMFILE, WSAENOTSOCK	listen() 함수 참조
기타	WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, WSAEINVAL, WSAENOBUFS -> 소켓 생성 참조

<accept() 함수 에러 코드>

accept() 함수의 두 번째 인수에 클라이언트의 아이피와 포트 등을 SOCKADDR_IN 구조체 변수로 반환 받을 수 있습니다. 세 번째 인수는 입/출력 인수로 두 번째 인수의 크기를 입력하고 다시 반환을 받습니다.

```
SOCKET          scCln=0;
SOCKADDR_IN     sdCln={0};
int iSize=sizeof(sdCln);
if(INVALID_SOCKET == (scCln=accept(scLstn, (SOCKADDR*)&sdCln, &iSize)))
{
        hr = WSAGetLastError();
        // 에러 처리
```

때로는 2~3 번째 인수(addr, addrlen)를 NULL로 설정할 수도 있습니다. 이 경우 클라이언트의 정보를 표시하기 위해서 소켓에서 getpeername() 함수를 사용해서 소켓 구조체를 얻기도 합니다.

```
void GetIp(char* s, SOCKET scH)
{
        int size = sizeof(SOCKADDR_IN);
        SOCKADDR_IN sdH ={0};
        getpeername(scH, (SOCKADDR *)&sdH, &size);
        strcpy(s, inet_ntoa(sdH.sin_addr) );
}
```

서버가 클라이언트와 통신할 수 있는 소켓을 accept() 함수로 얻었습니다. 클라이언트에게 데이터를 전송하기 위해서 send() 함수를 사용합니다.

```
int send(SOCKET s, const char FAR* buf, int len, int flags)
```

send() 함수에 접속한 소켓, 보낼 데이터가 저장된 버퍼, 데이터의 길이, 그리고 동작에 대한 flag를 설정하면 데이터가 전송 되고 성공하면 0을 반환하고 실패하면 SOCKET_ERROR를 반환 합니다.

에러 코드	내용
WSAEACCES	요청한 주소가 broadcast 주소인데 적당한 flag(SO_BROADCAST)가 설정 안 되어 있음
WSAEINTR	소켓이 닫혀있음
WSAEFAULT	buf(buffer) 인수가 완전(유효)하지 않음
WSAENETRESET	작업이 진행되는 동안 keep-alive 활동이 실패를 찾아 연결을 끊김
WSAENOTCONN	연결되지 않은 소켓
WSAENOTSOCK	소켓이 아님
WSAEOPNOTSUPP	지원되지 않은 작업. MSG_OOB가 지정되더라도 소켓이 SOCK_STREAM 와 같은 타입이 아님
WSAESHUTDOWN	소켓이 정지. SD_SEND 또는 SD_BOTH로 소켓이 shutdown()함수로 설정됨
WSAEWOULDBLOCK	소켓이 nonblocking로 표시되고, 요청한(수신) 작업이 블로킹 상태
WSAEMSGSIZE	소켓이 메시지 지향형(ex: UDP)이고 메시지의 크기가 전송계층의 하부에서 제공되는 최대보다 큼
WSAEHOSTUNREACH	데이터 전송이 시간 내에 원격 호스트 까지 도달하지 못함
WSAECONNABORTED	Time-out 또는 다른 실패의 원인으로 가상 연결 회선이 끊김
WSAECONNRESET	가상 회로망이 원격지에서 "hard"나 "abortive" 종료를 수행해서 리셋됨
WSAETIMEDOUT	다른 쪽 시스템이 알림 없이 종료되거나 네트워크 장애로 접속이 끊어짐(drop)
기타	WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, WSAENOBUFS, WSAEINVAL, WSAEPROTONOSUPPORT -> 소켓 생성 참조

<send() 함수 에러 코드>

nonblocking 소켓을 사용하게 되면 에러가 아니더라도 대기나 다른 이유로 실패(SOCKET_ERROR)를 자주 반환 받는데 이 때 WSAGetLasterError() 함수를 사용해서 에러의 원인을 파악하고 대처합니다. flag는 동작을 정하는 값들이 사용되고 특별한 경우가 아니면 0으로 설정합니다.

```
char sbuf[128]={0};
sprintf(sbuf, "Connected Message. socket: %d", (int)scCln);
if(SOCKET_ERROR == (send(scCln, sbuf, strlen(sbuf), 0)))
{
        hr = WSAGetLastError();
        // 에러 처리
```

데이터 송신이 있으면 반대로 수신도 있습니다. send() 함수와 대응되는 recv() 함수는 데이터 수신을 위한 함수 입니다.

```
int recv(SOCKET s, char FAR* buf, int len, int flags)
```

recv() 함수는 소켓, 버퍼 주소, 버퍼의 최대 크기 등을 인수로 전달하면 소켓의 수신 버퍼에 쌓여 있는 데이터를 반환하거나 블로킹 소켓의 경우 반환될 때까지, 즉, 데이터 수신이 있을 때까지 기다리는 함수 입니다.
만약 에러가 없다면 수신된 바이트 수를 반환합니다. 만약 서로가 모든 전송을 끝내고 상대편이 closesocket() 함수 등을 사용해서 소켓을 해제한 우아한 종료(gracefully closing)를 요청하면 숫자 '0'을 반환합니다. 우아한 종료를 위해서 송/수신이 작업 중이면 shutdown() 함수를 closesocket() 함수 호출 전에 미리 실행 합니다.

이외에 실패하면 SOCKET_ERROR를 반환하며 WSAGetLastError() 로 에러를 확인 합니다. 특히, nonblocking 소켓에서 WSAEWOULDBLOCK이 자주 발생하므로 주의해야 합니다.

에러 코드	내용
WSAEFAULT	buf(buffer) 인수가 완전(유효)하지 않음
WSAESHUTDOWN	소켓이 정지됨. SD_RECEIVE 또는 SD_BOTH로 소켓이 shutdown()함수로 설정됨
WSAEWOULDBLOCK	소켓이 nonblocking로 표시되고, 요청한(수신) 작업이 블로킹 상태
send() 와 동일	WSAENOTCONN, WSAEINTR, WSAENETRESET, WSAENOTSOCK, WSAEMSGSIZE WSAEHOSTUNREACH, WSAECONNABORTED, WSAETIMEDOUT, WSAECONNRESET

기타	WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, WSAENOBUFS, WSAEINVAL, WSAEPROTONOSUPPORT -> 소켓 생성 참조

<recv() 함수 에러 코드>

지금까지 서버의 프로그램 흐름과 관련 함수들을 순차적으로 알아보았습니다. 정리한다면 사실, 네트워크에 데이터를 송/수신하는 것은 파일에 입/출력 하는 것과 구현 대상만 다를 뿐 본질적으로 동일 내용입니다. 이러한 이유로 유닉스/리눅스 시스템은 파일(또는 시스템) 입출력과 동일하게 네트워크 소켓 생성만 socket() 함수를 사용하고 write(send), read(recv), close(closesocket) 등 파일 입/출력 함수들을 그대로 사용합니다.
재미있는 사실은 유닉스/리눅스 계열은 open/close/read/write/ioctl 다섯 개의 함수로 시스템을 통제할 수 있다고 합니다.

```
// 간단한 1-1 통신 서버
char    sPt[32]="20000";
…
int main()
…
SOCKET          scLstn=0;
SOCKADDR_IN     sdLstn={0};
SOCKET          scCln=0;
SOCKADDR_IN     sdCln={0};
…
// Load WinSock DLL
…
char sBufSnd[1024] = "Welcome to network programming!!!";

// TCP Listen 소켓 생성
if(INVALID_SOCKET == (scLstn = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)))
        return -1;

// Listen 소켓에 주소, 포트 할당
sdLstn.sin_family      = AF_INET;
sdLstn.sin_addr.s_addr = htonl(INADDR_ANY);
sdLstn.sin_port        = htons( atoi(sPt) );
if(SOCKET_ERROR  == bind(scLstn, (SOCKADDR*)&sdLstn, sizeof(SOCKADDR_IN)))
        return -1;
```

```
// Listen 상태
if( SOCKET_ERROR == listen(scLstn, SOMAXCONN))
        return -1;

// nonblocking 리슨 소켓를 accept 함수에 적용하면 연결 수락까지 기다림
int iSize=sizeof(sdCln);
if(INVALID_SOCKET == (scCln=accept(scLstn, (SOCKADDR*)&sdCln, &iSize)))
        return -1;
…
while(1)
{
        ++iCnt;
        sprintf(sBufSnd, "Network message %d", iCnt);
        send(scCln, sBufSnd, strlen(sBufSnd), 0);       // 데이터 송신
}

// 연결종료
closesocket(scCln);     // client socket
closesocket(scLstn);    // listen socket
…
```

<TCP 서버: nw02_tcp_basic.zip>

1.5 TCP Client 프로그래밍

서버와 통신하는 클라이언트는 bind, listen 상태 없이 소켓이 생성되고 나서 바로 서버와 접속(connnect)을 시도합니다. 접속이 성공하면 서버와 마찬가지로 송수신합니다. 전체 클라이언트의 동작은 다음과 같이 요약할 수 있습니다.

1) 윈속 라이브러리 초기화
2) 원격 서버 접속용 소켓 생성
3) 접속(Connection)
4) 서버와 데이터 송/수신
5) 소켓 해제
6) 윈속 해제

클라이언트는 Connection에서 서버의 접속 허락을 받는데 blocking 소켓의 경우, 타임아웃 될 때

까지 접속을 기다립니다. 여기서 blocking의 의미는 하나의 작업(또는 처리)가 끝날 때까지 다음 작업은 기다리는 것으로 비선점형 멀티-테스킹과 유사합니다.

```
int connect(SOCKET s, const struct sockaddr FAR* name, int namelen)
```

connect 함수는 지정된 소켓을 지정된 원격지로 접속을 요청하는 함수로 이루는 함수로 서버가 연결을 허락하고 접속이 성공하면 0을 반환 합니다. 실패하면 SOCKET_ERROR를 반환하고 에러 코드는 WSAGetLastError() 함수로 얻을 수 있습니다.

에러 코드	내용
WSAEACCES	옵션 SO_BROADCAST을 활성화 되지 않은 상태에서 접근이 허가되지 않은 방식으로 데이터그램(Datagram) 소켓 접근
WSAEADDRINUSE	재사용 SO_REUSEADDR 옵션을 적용하지 않고 이미 사용된 주소(프로토콜/네트를 사용함
WSAEADDRNOTAVAIL	유효하지 않은 주소
WSAEFAULT	주소 체계와 주소가 잘못 또는 일치하지 않거나 namelen의 값이 작음.
WSAENOTSOCK	소켓이 아님
기타	WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, WSAENOBUFS, WSAEINVAL -> 소켓 생성 참조

만약 소켓이 연결 지향형 nonblocking 소켓으로 만들어지고 connect() 함수를 실행한다면 바로 접속이 이루어지지 않는 경우(SOCKET_ERROR)가 많습니다. 이 때 WSAGetLastError()로 에러 코드를 확인 해서 WSAEWOULDBLOCK 이면 실패가 아닌 접속을 기다리고 있는 것으로 처리하고 접속에 대한 통지 메시지 처리를 구현합니다.
만약 통지를 select() 함수로 사용하는 경우, 성공하면 쓰기 소켓 집합(Write File Descriptions Set)에 통지가 오고, 실패의 경우 예외 집합(Exception FDs Set)에 메시지가 저장됩니다.
만약 소켓이 WSAAsyncSelect() 함수나 WSAEventSelect() 함수를 사용했다면, 성공과 실패 상관없이 FD_CONNECT로 통지됩니다. 실패를 처리하기 위해서 WSAAsyncSelect는 WSAGETSELECTERROR와 같은 매크로를 통해서 에러 코드를 얻고, WSAEventSelect는 WSAEnumNetworkEvents에서 처리한 WSANETWORKEVENTS 구조체 변수 iErrorCode[FD_CONNECT_BIT]에 저장된 값으로 에러 코드를 얻습니다.

connect() 함수는 주소, 포트를 소켓에 binding 하고 원격 호스트에 접속을 요청하기 때문에 인수의 구조와 반환 값이 listen 소켓의 binding 할 때 사용한 bind() 함수와 같습니다. 따라서 bind와 마찬가지로 sockaddr 보다 sockaddr_in 구조체 변수를 사용하고 구조체 변수의 값은 전부 big-

endian을 사용해야 합니다.

```
SOCKET          scHost=0;
SOCKADDR_IN     sdHost={0};
…
sdHost.sin_family      = AF_INET;
sdHost.sin_addr.s_addr = inet_addr("127.0.0.1");
sdHost.sin_port        = htons( 20000 );
if(SOCKET_ERROR == connect(scHost, (SOCKADDR*)&sdHost, sizeof(SOCKADDR_IN)))
        hr = WSAGetLastError();
```

클라이언트의 send(), recv() 함수는 서버와 동일 하므로 설명은 생략하겠습니다. 다음은 이전의 서버에 대응하는 간단한 blocking 소켓을 사용한 클라이언트 코드 입니다.

```
// 간단한 1-1 통신 클라이언트
…
#define MAX_BUF 8192
char    sPt[32]="20000";
char    sIp[64]="127.0.0.1";

int main()
…
SOCKET          scHost=0;
SOCKADDR_IN     sdHost={0};
…
// Load WinSock DLL
…
// 서버 접속을 위한TCP 소켓 생성
if(INVALID_SOCKET == (scHost=socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)))
        return -1;

// 서버 연결 요청
sdHost.sin_family      = AF_INET;
sdHost.sin_addr.s_addr = inet_addr(sIp);
sdHost.sin_port        = htons( atoi(sPt) );
if(SOCKET_ERROR == connect(scHost, (SOCKADDR*)&sdHost, sizeof(SOCKADDR_IN)))
        return -1;
```

```
while(1)
{
	char sBufRcv[MAX_BUF+4]={0};
	int iRcv=0;

	// 데이터 수신
	iRcv = recv(scHost, sBufRcv, MAX_BUF, 0);

	if(SOCKET_ERROR == iRcv)
	{
		printf("Receive Socket Error₩n");
		hr = WSAGetLastError();
		LogGetLastError(hr);
		break;
	}
	else if(0 == iRcv)
	{
		printf("Gracefully closed₩n");
		break;
	}

	printf("Recv from server : %s₩n", sBufRcv);
}

// 연결종료
closesocket(scHost);

// Unload WinSock DLL
...
```

<nw02_tcp_basic.zip>

1.6 Socket Option

소켓은 다양한 범위에 적용되기 위해서 범용으로 제작 되었습니다. 따라서 적당한 옵션을 주어서 응용 프로그램에 맞는 소켓으로 바꾸어야 합니다. ioctlsocket(), {set|get}sockopt() 함수는 소

켓을 제어하는 함수로 제어가 성공하면 0을 반환하고 실패하면 SOCKET_ERROR를 반환합니다.

```
int ioctlsocket(SOCKET s, long cmd, u_long FAR* argp)
```

ioctlsocket(i/o control socket) 함수는 소켓의 입출력 모드를 제어하는 함수로 특정한 상태로 지정한 소켓을 제어하는데 사용됩니다. 이 함수는 소켓과 연관된 명령어(Command) 대해서 마지막 인수를 가지고 소켓의 상태를 설정하거나 반대로 값을 얻어올 수 있습니다.

Command	내용
FIONBIO	argp 인수의 값이 0이면 소켓을 동기(blocking) 모드로, 0이 아니면 비동기(nonblocking) 모드로 변경. 참고로 모든 소켓은 동기 모드로 시작함
FIONREAD	네트워크의 입력 버퍼에서 한 번에 소켓으로부터 읽어올 수 있는 전체 데이터 크기를 얻기 위해 사용
SIOCATMARK	소켓으로부터 out-of-band 데이터가 모두 읽혀졌는지를 판단하기 위해 사용

이들 command 중에서 우리는 FIONBIO를 기억해야 합니다. 설명에도 나와 있듯이 이 명령어 인수는 동기(blocking), 비동기(nonblocking) 모드 소켓을 설정합니다. 소켓을 생성하면 모든 소켓은 동기 모드(blocking mode)로 동작 합니다.
동기 모드는 scanf() 함수 또는 GetMessage() 함수처럼 요청한 동기 처리(blocking process) 끝날 때까지 동기 모드를 호출한 프로세스를 대기 상태로 만듭니다.

동기 모드는 확실히 동기화라는 강점이 있지만 네트워크 통신은 PC와 다르게 원격으로 처리되기 때문에 처리에 대한 많은 응답 시간이 소요될 수 있습니다. 따라서 작업을 요청하고 완료에 대한 통지를 나중에 통보 받도록 비동기 통지 방식의 프로그램으로 작성하는 것이 보통입니다.

비동기 소켓을 사용할 때 주의할 것은 에러 처리 입니다. 비동기 소켓을 사용하면 요청한 작업의 대부분이 원격으로 처리되기 때문에 실패(SOCKET_ERROR)를 반환합니다. 때문에 반드시 WSAGetLastError() 함수로 에러 코드를 확인하고 WSAEWOULDBLOCK이면 작업이 처리 중으로 여기고 나중에 소켓 함수를 다시 호출합니다.
좀 더 자세한 비동기 통지에 대한 예는 WSAAsyncSelect, WSAEventSelect 에서 다시 알아보도록 하겠습니다.
다음은 비동기 통지를 받기 위해서 FIONBIO 명령어를 사용, 소켓 입출력 모드를 비동기 소켓 모드로 변경하는 예입니다.

```
// 비동기 소켓 모드
int v = 1;
int size = sizeof(v);
```

```
ioctlsocket(scHost, FIONBIO, (u_long*)&v);
```

```
int getsockopt(SOCKET s, int level, int optname
                        , char FAR* optval, int FAR* optlen)
int setsockopt(SOCKET s, int level, int optname
                        , const char FAR* optval, int optlen)
```

getsockopt() 함수는 설정된 소켓의 옵션 값을 가져오고, setsockopt() 함수는 소켓 옵션을 설정하는 함수입니다.
입력 값 level 은 소켓 옵션 레벨의 정의이며, SOL_SOCKET과 IPPROTO_TCP 중 하나가 될 수 있습니다.
optname은 함수가 처리할 소켓 옵션입니다.
optval 는 요청된 옵션에 대한 입력 또는 반환 되는 값이 저장될 버퍼 주소입니다.
마지막 인수 optlen은 입력 값 버퍼의 크기로 사용되거나 반대로 얻어올 값(버퍼)의 크기를 의미합니다.

옵션	Data Type	내용
+SO_ACCEPTCONN	BOOL	listen 상태
SO_BROADCAST	BOOL	브로드캐스트(broadcast) 가능 소켓
+SO_BSP_STATE	CSADDR_INFO	로컬,원격 주소/포트 , 소켓 타입, 프로토콜 정보 반환
SO_CONDITIONALW_ACCEPT	BOOL	Listening 소켓에서 연결 허용 여부 결정
+SO_CONNECTW_TIME	DWORD	연결지향형 소켓에서 연결 되고 나서 경과된 시간
SO_DEBUG	BOOL	디버깅 가능
SO_DONTLINGER	BOOL	SO_LINGER 옵션 무시
SO_DONTROUTE	BOOL	라우팅 테이블 무시
+SO_ERROR	int	에러 상태 반환하고 에러 상태 지움
SO_EXCLUSIVEWADDRUSE	BOOL	같은 주소, 포트로 바인딩 하는 다른 소켓(프로세스)을 막음. 반드시 bind() 함수 실행 전에 호출. Ex)고의로 같은 주소/포트를 사용하는 프로그램에도 적용됨
+SO_GROUP_ID	GROUP	예약됨
SO_GROUPW_PRIORITY	int	예약됨

SO_KEEPALIVE	BOOL	Keep-alive 활성화. ATM 소켓 제외
SO_LINGER	struct LINGER	현재의 linger 옵션
+SO_MAX_MSGW_SIZE	unsigned int	SOCK_DGRAM와 같은 메시지 지향형 소켓의 메시지 최대 크기.
SO_OOBINLINE	BOOL	OOB Data 를 일반 데이터 스트림으로 수신
+SO_PORTW_SCALABILITY	BOOL	로컬 컴퓨터에서 다른 로컬 주소-포트 쌍에 대해 여러 번 와일드 카드 포트를 할당하는 포트 할당이 최대까지 허용된 소켓에 대한 로컬 포트의 확장성 활성화
+SO_PROTOCOLW_INFO	WSAPROTOCOLW_INFO	바인딩 된 프로토콜 정보
SO_RCVBUF	int	수신 버퍼크기
-SO_RCVTIMEO	DWORD	동기 모드 소켓의 수신 상태 타임 아웃
SO_REUSEADDR	BOOL	사용된 주소의 재사용 옵션. ATM 소켓 제외
SO_SNDBUF	int	전송 버퍼 크기
-SO_SNDTIMEO	DWORD	동기 소켓의 전송 상태 타임 아웃
+SO_TYPE	int	소켓의 타입 (ex: SOCK_STREAM, SOCK_DGRAM)
-SO_UPDATEW_ACCEPT_CONTEXT	int	listening 소켓의 문맥(context)으로 accepting 소켓 갱신
PVD_CONFIG	Service Provider Dependent	SP(Service Provider)로부터 온 소켓과 관련된 불투명한 자료구조 객체. 이 객체는 현재의 SP 구성 정보를 저장한 객체. 이 자료구조의 정확한 형식은 포맷은 SP에 따름

<level = SOL_SOCKET, +: get, -: set only>

옵션	Data Type	내용
+SO_ACCEPTCONN	BOOL	listen 상태
SO_BROADCAST	BOOL	브로드캐스트(broadcast) 가능 소켓
+SO_BSP_STATE	CSADDR_INFO	로컬,원격 주소/포트 , 소켓 타입, 프로토콜 정보 반환
SO_CONDITIONALW_ACCEPT	BOOL	Listening 소켓에서 연결 허용 여부 결정
+SO_CONNECTW_TIME	DWORD	연결지향형 소켓에서 연결 되고 나서 경과된 시간

SO_DEBUG	BOOL	디버깅 가능
SO_DONTLINGER	BOOL	SO_LINGER 옵션 무시
SO_DONTROUTE	BOOL	라우팅 테이블 무시
+SO_ERROR	int	에러 상태 반환하고 에러 상태 지움
SO_EXCLUSIVEW ADDRUSE	BOOL	같은 주소, 포트로 바인딩 하는 다른 소켓(프로세스)을 막음. 반드시 bind() 함수 실행 전에 호출. Ex)고의로 같은 주소/포트를 사용하는 프로그램에도 적용됨
TCP_NODELAY	BOOL	Nagle 알고리즘

<level = IPPROTO_TCP>

이들 옵션 중에서 TCP_NODELAY은 Nagle 알고리즘의 활성 여부 입니다. Nagle 알고리즘은 전송 버퍼에 MSS(Maximum Segment Size)로 정의된 크기만큼 쌓이면 상대편에 보내고 MSS보다 작으면 이전에 보낸 데이터에 대한 ACK를 기다립니다. ACK가 도착하면 쌓여 있는 데이터가 MSS보다 작더라도 전송합니다.
Nagle 알고리즘을 사용하면 전송과 응답(ACK) 횟수가 줄이기 네트워크의 트래픽(Traffic)를 감소시킬 수 있습니다. 하지만 게임은 트래픽이 발생하더라도 전송할 일이 있으면 무조건 보내야 하기 때문에 Nagle 알고리즘 적용을 off(TRUE) 해야 합니다. 소켓은 기본적으로 Nagle이 활성화 되어 있습니다.

```
// Nagle off
int v = 1;
int size = sizeof(v);
setsockopt(scHost, IPPROTO_TCP, TCP_NODELAY, (char*)&v, size);
```

소켓을 해제하더라도 서버와 클라이언트의 소켓 종료는 완전히 종료할 때까지 시간이 소요(Time-Wait) 됩니다. 완전히 종료되기 전에 <주소 체계/주소/포트>를 새로운 소켓에 바인딩 하면 WSAEADDRINUSE가 발생합니다. 주소를 바인딩 하기 전에 SO_REUSEADDR를 사용하면 사용 중인 주소를 바인딩 할 수 있습니다.

```
// SO_REUSEADDR: 반드시bind 전에
v = 1;
setsockopt(scHost, SOL_SOCKET, SO_REUSEADDR, (char*)&v, size);
```

접속을 다시 하려면 보통 소켓을 해제하고 다시 주소를 바인딩 하는데 소켓을 해제하기 전에 shutdown() 함수를 호출해서 송/수신 데이터를 완전히 차단합니다. SD_RECEIVE(수신 차단),

SD_SEND(송신 차단), SD_BOTH(송/수신 차단) 3가지 방법 중에서 SD_BOTH 를 사용합니다.

```
// Shutdown both send and receive operations.
shutdown(scHost, SD_BOTH);
closesocket(scHost);
```

소켓의 종료는 종료 요청(FIN)을 보내고 ACK를 받습니다. 상대방도 FIN을 보내고 ACK를 수신합니다. 이러한 과정을 거치지 않고 곧바로 종료할 필요도 있습니다. SO_DONTLINGER를 사용하면 gracefully closing이 되고 함수는 바로 리턴 합니다.

```
// SO_DONTLINGER: do not linger on close waiting for unsent data to be sent
v = 1;
setsockopt(scHost, SOL_SOCKET, SO_DONTLINGER, (char*)&v, size);
```

때로는 소켓을 강제 종료할 필요도 있고, 강제 종료할 때 시간을 설정해서 종료하거나 즉시 종류할 수도 있습니다. LINGER 구조체와 SO_LINGER를 사용해서 closesocket() 함수가 실행하는 방식을 정할 수 있습니다. LINGER 구조체의 l_onoff 가 0 이면 closesocket() 함수에서 바로 빠져 나오고 0이 아니면 l_linger로 설정한 시간 동안 기다립니다. 다음은 기다리지 않고 곧 바로 종료되는 옵션 설정 입니다.

```
// Linger 설정
LINGER l={1,0};
size = sizeof(LINGER);
setsockopt(scHost, SOL_SOCKET, SO_LINGER, (char*)&l, size);
```

recv() 함수는 소켓에 할당된 수신 버퍼의 데이터를 가져오는 함수 입니다. IP datagram은 16비트로 저장 되므로 TCP의 세그멘트는 헤더를 포함해서 최대 65535바이트를 한 번에 송/수신 할 수 있습니다. 그런데 이더넷(Ethernet)를 LAN위에 TCP/IP를 사용하므로 이더넷의 MTU(Maximum Transport Unit) 1500byte에 영향을 받습니다. 따라서 한번에 보낼 수 있는 데이터 크기는 IP 헤더(20byte)와 TCP 헤더(20~44byte)를 제외하면 대략 1430byte정도만 사용할 수 있습니다. 수신 버퍼를 꺼내 가지 않으면 넘쳐날 수도 있습니다. 이를 방지하기 위해 수신할 수 있는 최대 크기에 4~6배의 공간을 확보합니다.
보통 수신 버퍼는 8k 정도로 설정되어 있으며 필요하다면 SO_RCVBUF를 사용해서 버퍼의 크기를 조정 할 수 있습니다. 윈속은 특별히 이 값을 0으로 설정하면 소켓의 수신 버퍼를 사용하지 않고 사용자가 지정한 버퍼에 수신된 데이터가 쌓이게 되어 수신 버퍼에서 프로그램이 설정한 버퍼로 복사되는 buffering을 줄일 수 있습니다.

```
// SO_RCVBUF: check the receive buffer size for recv
```

```
v = 0;
size = sizeof(v);
hr = setsockopt(scHost, SOL_SOCKET, SO_RCVBUF, (char*)&v, size);
hr = getsockopt(scHost, SOL_SOCKET, SO_RCVBUF, (char*)&v, &size);
```

수신 버퍼의 옵션을 설정하는 것과 동일 하게 송신 버퍼도 SO_SNDBUF을 가지고 송신 버퍼의 크기를 설정할 수 있으며, 0으로 설정하면 프로그램이 지정한 버퍼를 송신버퍼로 사용합니다.
송/수신에서 직접 버퍼를 사용하는 경우에 주의할 것은 송/수신이 완료되지 않은 상태에서 send() 또는 recv() 함수에 설정된 버퍼의 내용을 변경하면 안됩니다.

```
// SO_SNDBUF: check the send buffer size for send
v = 0;
hr = getsockopt(scHost, SOL_SOCKET, SO_SNDBUF, (char*)&v, &size);
...
// directly perform i/o on the buffer (no buffering and no copy)
size = sizeof(v);
v = 0;
hr = setsockopt(scHost, SOL_SOCKET, SO_SNDBUF, (char*)&v, size);
```

<nw03_sockopt.zip>

지금까지 TCP와 관련된 몇 개의 주요한 소켓 옵션을 살펴보았습니다. UDP로 프로그램을 작성한다면 마찬가지로 적당한 옵션을 선택해서 네트워크 통신의 환경을 최적으로 만들어주는 것이 필요합니다.

1.7 비동기 모드 소켓과 Relay 서버

동기 모드(blocking mode) 소켓은 하나의 처리가 완료될 때까지 프로세스가 대기하기 때문에 여러 소켓을 처리하지 못하고 1:1 통신 정도만 제대로 진행 할 수 있었습니다.
만약 비동기 모드(nonblocking mode)소켓을 사용한다면 서버-클라이언트 통신을 완전하지 않지만 적당한 1:n 통신으로 구성할 수 있습니다.

여기서 비동기 모드 소켓으로 설정할 때 전술이 필요합니다. 서버는 accept/send/recv 를 실행합니다. 따라서 accept를 담당하는 listen 소켓, 그리고 accept를 함수를 통해서 생성된 소켓을 비동기 모드로 변경합니다. listen 은 bind() 함수 실행 후에, 그리고 접속한 클라이언트는 accpt() 함수가 성공하면 해당 클라이언트 소켓을 비동기 모드로 설정합니다.

```
// 서버
bind(scLstn, (SOCKADDR*)&sdLstn, sizeof(SOCKADDR_IN));
…
// listen socket을 비동기 소켓 설정
u_long nonBlocking =1;
hr = ioctlsocket(scLstn, FIONBIO, &nonBlocking);
…

while(1)
{
…
        scCln=accept(scLstn, (SOCKADDR*)&sdCln, &iSize);
        if(INVALID_SOCKET == scCln)
        {
                hr = WSAGetLastError();
                if( WSAEWOULDBLOCK != hr)
                        goto END;

                continue;
        }

        // 클라이언트와 통신하는 소켓을 비동기 소켓으로 설정
        u_long nonBlocking =1;
        hr = ioctlsocket(scCln, FIONBIO, &nonBlocking);
        break;
}
…
```

<nw04_nonblocking.zip>

클라이언트는 connect/send/recv 를 실행합니다. connection 전에 비동기로 설정하면 접속 통지(Connection Notify)를 얻을 수 없기 때문에 connection() 함수가 성공하면 비동기 모드로 설정합니다. 물론 recv() 함수 호출에서 SOCKET_ERROR를 받으면 WSAEWOULDBLOCK인지 꼭 확인해야 합니다.

```
// 클라이언트
if(SOCKET_ERROR == connect(scHost, (SOCKADDR*)&sdHost, sizeof(SOCKADDR_IN)))
        return -1;
```

```
// 비동기 소켓 설정
u_long nonBlocking =1;
ioctlsocket(scHost, FIONBIO, &nonBlocking);
…
while(1)
…
        // 데이터 수신
        iRcv=recv(scHost, sBufRcv, MAX_BUF, 0);
        if(SOCKET_ERROR == iRcv)
        {
                hr = WSAGetLastError();
                if( WSAEWOULDBLOCK == hr)
                        continue;

                printf("Receive Socket ErrorWn");
                break;
        }
…
```

<nw04_nonblocking.zip>

간단히 비동기 모드의 서버-클라이언트 1:1 통신을 구현해 보았습니다. 앞서 비동기 모드로 소켓을 만들면 1:n 통신이 가능하다고 했습니다. 클라이언트는 변경할 필요는 서버에서 accept/send/recv 처리를 하나의 while Loop에서 진행하도록 코드를 작성하도록 합니다.
accpt() 함수가 실패했을 때 에러 코드가 WSAEWOULDBLOCK이 아니면 listen 소켓의 문제이므로 while loop를 빠져 나오도록 하고 유효한 소켓이면 클라이언트 리스트에 추가합니다.
수신 처리 과정은 접속한 클라이언트 소켓만큼 순회하면서 수신된 데이터가 있는지 recv() 함수로 확인합니다. 테스트를 위해서 수신된 데이터가 있으면 접속한 클라이언트 모두에서 송신합니다.
또한 recv() 함수 호출에서 접속을 종료 했거나 유효하지 않은 소켓이 있으면 종료하고 소켓을 0으로 설정합니다.
수신 처리가 끝나고 나서 소켓이 0으로 설정된 데이터는 소켓 리스트에서 제거합니다. 프로그램의 편의를 위해서 STL vector 컨테이너를 사용했습니다.

```
while(1)
…
        // accept process
        scCln=accept(scLstn, (SOCKADDR*)&sdCln, &iSize);
```

```
if(INVALID_SOCKET == scCln)
{
        hr = WSAGetLastError();
        if( WSAEWOULDBLOCK != hr)
                break;
}
else
{
        // 새로운 소켓을 소켓 리스트에 추가
        m_vCln.push_back(scCln);
        …
}
…
// receive process
for( ; _F != m_vCln.end(); ++_F)
…
        int iRcv = recv(scH, sRcv, MAX_BUF, 0);
        if(SOCKET_ERROR == iRcv)
        {
                hr = WSAGetLastError();
                if(WSAEWOULDBLOCK != hr)
                {
                        // Err:close
                        …
                        (*_F) = 0;
                }
                continue;
        }
        else if(0 == iRcv)
        {
                // close
                …
                (*_F) = 0;
                continue;
        }

        …
```

```
                // send the received message to all client
                EchoMsg(sSnd, iLen);
        }
        // 사용할 수 없는 소켓 제거
        DeleteNotUseHost();
...
```

<nw05_relay.zip>

2 스레드(Thread), 동기화(Synchronization)

2.1 스레드

이전 예제 nw05_relay.zip는 하나의 프로세스에서 accept(), send(), recv()를 실행하고 있습니다. 만약 이 코드를 가지고 채팅 서버나 룸(Room) 서버 등 처리할 일이 많은 서버를 구성한다고 했을 때 하나의 프로세스에서 거의 독립된 프로세스들을 같이 처리하는 것은 코드의 확장성이 떨어지고 설령 구현했다 하더라도 복잡해질 것이라는 것은 분명합니다.
또한 이들은 비동기 소켓이기 때문에 소켓에서 네트워크 이벤트(accept/connect/send/recv/close)가 발생하지 않아도 계속 실행하고 있어서 CPU에 처리할 일이 없이 단지 점유만 하고 있어서 컴퓨터의 성능을 떨어뜨릴 확률이 높습니다. 따라서 먼저 이들은 다중 처리가 가능하도록 독립적으로 동작하는 프로세스가 되도록 해야 하고 다음으로 CPU의 점유율을 낮추기 위해서 네트워크 이벤트가 발생했을 때만 실행 할 수 있는 프로세스로 구성 되어야 합니다.
이와 같은 문제점들은 Multi-Tasking(다중 작업)를 사용해서 해결할 수 있습니다. 특히, 멀티-테스킹이 지원되는 모든 운영체제들은 멀티-프로세싱(Multi-Processing) 또는 멀티-스레딩(Multi-Threading) API를 제공합니다. 이들 API를 사용해서 accept(), send(), recv() 등 통신에 관련 된 처리를 독립적으로 동작하는 프로세스로 만들 수가 있습니다.

마이크로 소프트 윈도우 계열은 프로세스를 생성하는 CreateProcess() 함수와 스레드를 생성하는 CreateThread() 함수가 제공됩니다.
프로세스란 우리가 프로그램이라고 부르는 것으로 CPU를 점유하고 있는 실행 중인 프로그램입니다. 프로세스는 프로그램을 구동하기 위해서 필요한 자원(기억 공간, 파일/하드웨어/소프트웨어 Handle 등)과 CPU의 작업에 대한 실행 문맥(execution context)을 처리하는 스레드(thread)로 구성 되어 있습니다. 간단히 프로세스는 자원+스레드라 생각하면 됩니다.
하나의 프로세스는 최소한 하나의 스레드를 가지며 main()/WinMain() 함수로 실행되는 스레드를 주 스레드(main thread)라 하며 주 스레드 이외에 다수의 자식 스레드를 둘 수가 있습니다.
이렇게 다수의 스레드를 가진 프로그램은 우리가 원하는 다중 처리에 알맞은 형태라 할 수 있습니다. 물론 멀티-스레딩 대신 멀티-프로세싱으로 프로세스가 프로세스를 생성해서 해결할 수도 있지만 이 방법은 네트워크 통신에 대해서 스레드를 사용하는 것보다 시간적, 공간적으로 효율이 떨어집니다.
이유는 간단한데 먼저 프로세스는 자원+스레드이기 때문에 스레드를 생성하는 것보다 자원의 소모가 큽니다. 다음으로 정해진 CPU를 여러 프로세스가 사용하기 위해서 사용 중인 프로세스의 명령어와 상태에 대한 문맥을 PCB(Process Control Block)에 보관하고 다른 프로세스의 문맥을 적재하는 문맥 교환(Context Switching) 작업의 시간이 스레드보다 프로세스가 더 소모합니다. 이러한 이유로 우리가 원하는 네트워크 프로그램에 대해서 멀티-스레딩을 선택하게 된 것이지만 완전한 독립 프로세스가 필요한 환경이라면 멀티-프로세싱을 선택하는 것도 당연합니다.

스레드를 사용할 때 주의할 것은 스레드는 스택만 독립적으로 사용할 수 있고 나머지(전역 변수, static 변수 등) 기억공간은 공유합니다.(기억 공간을 공유하는 내용은 동기화 문제를 만들 수 있으며 다음 장에서 더 좀 살펴보겠습니다.)
또한 스레드는 커널 객체(Kernel Object)로 시스템에서 관리 됩니다. 운영체제는 4개의 서브 시스템 즉, 프로세서 관리자, 주 기억 장치 관리자, 파일 관리자, 장치 관리자로 시스템을 관리하며 커널 객체는 이들 관리자에서 생성이 되고 해제(또는 시스템에 반납)하는 객체입니다.

대표적인 커널 객체가 파일 객체 입니다. 파일 객체는 윈도우 시스템에서 CreateFile() 함수로 객체가 생성이 되고 CloseHandle() 함수로 자원을 반납합니다. 참고로 윈도의 Naming rule에 의해 특별한 커널 객체들의 생성은 "Create" 접두어로 시작하고 첫 번째 인수에 보안 속성을 지정합니다. 특히 뮤텍스, 세마포어, 이벤트와 같은 동기화 객체들은 마지막 인수에 이름을 지정할 수 있어서 프로세스 사이에서 동일한 객체를 사용할 수도 있습니다.

```
BOOL CloseHandle(HANDLE hObject)
```

대부분의 커널 객체들은 CloseHandle()로 자원을 반납합니다. 참고로 CloseHandle()을 사용할 수 있는 커널 객체는 Access token, Communications device, Console input, Console screen buffer, Event, File, File mapping, Job, Mailslot, Memory resource notification, Mutex, Named pipe, Pipe, Process, Semaphore, Socket, Thread, Transaction, Waitable timer 등이 있습니다.

프로세스 생성도 공부해야겠지만 우리의 주 관심은 스레드에 있기 때문에 스레드에 관련된 함수들을 먼저 살펴 보겠습니다. 먼저 스레드 생성 함수입니다.

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpAttribs
        , SIZE_T dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress
        , LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId)
```

CreateThread() 함수는 이름 그대로 스레드를 생성하는 함수 입니다. 첫 번째 인수는 보안 속성을 전달하며 보통 NULL로 설정합니다. 두 번째 인수는 스레드의 스택 크기입니다. 0으로 설정하면 Default 크기(1MB)로 설정됩니다. 세 번째 인수는 스레드 함수 주소(함수 포인터)를 지정합니다. 스레드 함수는 <반환/std call/void* 인수>을 지켜야 하고 가장 일반적인 형태는 (DWORD)(__stdcall*)(void* pParam) 형식입니다.
네 번째 인수는 스레드 함수 인수인 pParam으로 전달되는 인수 입니다. 다섯 번째 인수는 스레드 시작 상태 입니다. 0으로 설정하면 스레드는 signal 상태로 곧바로 실행이 되며 CREATE_SUSPENDED로 설정하면 실행은 되지 않고 일시 중지 상태가 됩니다.

```
HANDLE  m_hThread = NULL;
```

```
DWORD   WINAPI ThreadProc(void* pParam)
{
        while(1)
        {
                // thread proc
        }
        return 0;
}

HANDLE hThread = CreateThread(NULL, 0, ThreadProc, NULL, 0, NULL);
CloseHandle(hThread);
```

스레드 함수가 스스로 종료해서 리턴 하기 전에 강제로 종료 시킬 수도 있습니다. 이 때 종료 코드를 GetExitCodeThread() 함수로 얻어야 합니다.

```
BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode)
```

만약 스레드가 계속 실행 중이라면 ExitCode에 STILL_ACTIVE 값을 얻을 수 있습니다.

GetExitCodeThread() 함수로 스레드의 종료 코드를 얻고 exit(0)과 유사하게 스레드 내부에서 종료하거나 아니면 외부에서 강제로 종료 할 수 있습니다.

```
void ExitThread(DWORD dwExitCode)
BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode)

DWORD   dExit=0;
hr = GetExitCodeThread(hThread, &dExit);
if(0 != hr && STILL_ACTIVE == dExit)
        TerminateThread(hThread, dExit);
```

ExitThread() 함수는 스레드 내부에서 종료하는 함수이고 TerminateThread()는 외부에서 강제로 스레드를 종료시키는 함수 입니다. 주의할 것은 스레드를 종료시킨다고 해서 곧바로 자원해 해제되지 않을 수도 있습니다. 스레드를 종료하는 가장 안전한 방법은 스레드 함수가 스스로 return 하도록 하는 것입니다.

스레드는 일시 중지시킬 수도 있습니다. SuspendThread() 함수는 스레드를 일시적으로 중지시키고, ResumeThread() 함수는 일시 중지된 스레드를 깨워 활성화 합니다. 주의할 것은 SuspendThread()

함수는 호출한 숫자만큼 스레드의 'suspend count' 를 증가시키기 때문에 스레드가 재기하기 위해서는 SuspendThread() 함수를 호출한 숫자만큼 ResumeThread() 함수를 호출해야 실행 됩니다.

```
DWORD SuspendThread(HANDLE hThread)
DWORD ResumeThread(HANDLE hThread)
```

SuspendThread() 함수 대신 Sleep() 함수를 사용하면 설정한 시간 동안 스레드를 일시 정지시킬 수도 있습니다. 설정한 시간이 0이면 같은 우선 순위를 가진 다른 스레드에게 실행 시간을 양보하지만 스레드가 없으면 즉시 계속 실행이 됩니다.

```
DWORD Sleep(DWORD milliseconds)
```

스레드 함수 내에서 현재의 스레드 핸들이 필요할 수도 있습니다. GetCurrentThread() 함수는 스레드 핸들을 반환 합니다. 이 함수로 얻은 핸들은 의사 핸들(Pseudo handle)이라 하며 CloseHandle 함수로 해제할 필요는 없습니다.

```
HANDLE GetCurrentThread()
```

스레드의 아이디도 필요할 때가 있습니다. GetCurrentThreadId() 함수는 스레드 함수 내부에서, GetThreadId() 함수는 스레드 함수 외부에서 스레드의 아이디를 얻는 함수 입니다.

```
DWORD GetCurrentThreadId()
DWORD GetThreadId(HANDLE Thread)
```

운영 체제는 스케줄링(scheduling)을 사용해서 각각의 프로세스들이 CPU를 사용하도록 합니다. 이들 우선 순위는 설정이 가능하며 {Get|Set}ThreadPriority() 함수로 처리합니다.

```
int GetThreadPriority(HANDLE hThread)
BOOL SetThreadPriority(HANDLE hThread, int nPriority)
```

게임 프로그램에서 자주 사용되는 스레드 생성함수는 _beginthreadex(), CreateThread() 함수 두 가지 입니다. _beginthreadex() 함수는 내부적으로 CreateThread() 함수를 사용하는데 C run-time 함수들을 안전하게 멀티스레드(Multithread) 환경에서 사용하기 위해 만들어진 함수 입니다. 만약 스레드 함수 내에서 C 함수들을 사용하고 있다면 _beginthreadex() 함수를 사용해야 합니다.
다음은 _beginthreadex() 함수와 CreateThread() 함수를 사용하는 예제 입니다.

```
DWORD WaitForSingleObject(HANDLE h, DWORD wait_timeout_milliseconds)
```

WaitForSingleObject() 함수는 객체가 신호 상태(signal)가 되거나 주어진 wait timeout(1/1000초 단위) 동안 프로세스를 대기 상태로 만드는 함수 입니다. 스레드, 뮤텍스, 세마포어 들과 자동 리셋 모드(auto reset mode) 이벤트들은 이 함수를 거치면 자동으로 비신호(non-signal) 상태가 됩니다.

```
uintptr_t _beginthreadex(void* security
                , unsigned stack_size, unsigned (__stdcall *start_address)(void *)
                , void* arglist, unsigned initflag, unsigned *thrdaddr)
void _endthreadex(unsigned retval)
```

CreateThread()에 대응하는 _beginthreadex() 함수와 ExitThread()에 대응하는 _endthreadex() 함수는 멀티-프로세싱 환경으로 고려되지 않고 작업이 된 라이브러리를 이용할 때 사용됩니다. 대표적으로 C-라이브러리 들 중 일부는 전역 변수를 사용합니다. 스레드는 전역 변수를 공유하기 때문에 원하지 않은 결과를 만들 수가 있고, _beginthreadex(), _endthreadex()는 이 문제를 해결하기 위해서 고안된 스레드 함수 입니다. _beginthreadex() 함수를 디버그 모드로 추적해 보면 안전하게 사용하기 위한 블록 설정과 CreateThread() 함수를 사용하고 있음을 볼 수 있습니다.

우리는 이 함수를 주로 사용할 것이며 이 함수들을 사용하기 위해서 <process.h> 파일을 포함 시키고 프로젝트의 속성 중에서 런-타임 라이브러리를 멀티-스레드 라이브러리로 설정합니다. 오래된 Visual Studio 6.0을 사용하는 경우에 프로젝트는 기본으로 Single Process로 설정되어 있기 때문에 꼭 확인 합니다.

이전의 스레드 함수를 가지고 _beginthreadex() 함수로 스레드를 만들 경우에 함수 형식에 대한 캐스팅이 필요합니다.

```
DWORD  dThread = 0;
HANDLE hThread = (HANDLE)_beginthreadex(NULL, 0
                        , (unsigned (__stdcall*)(void*))ThreadProc
                        , NULL, 0, (unsigned*)&dThread);
```

스레드를 연습해 보기 위해 주 스레드, 자식 스레드에서 카운터를 출력하는 예제를 만들어봅시다.

```
#include <windows.h>
#include <process.h>
#include <stdio.h>
```

```
// 테스트 스레드
DWORD WINAPI ThreadProc(void* pParam)
{
	INT nCnt=0;
	while(++nCnt<20)
	{
		Sleep(200);
		printf("Thread Proc: %d₩n", nCnt);
	}

	_endthreadex(0);
	return 0;
}

int main()
{
	// 스레드 생성
	HANDLE  hThread = NULL;
	DWORD   dThread = 0;
	hThread = (HANDLE)_beginthreadex(NULL, 0
			, (unsigned (__stdcall*)(void*))ThreadProc
			, NULL, 0, (unsigned*)&dThread);

	INT nCnt=0;
	while(++nCnt<20)
	{
		Sleep(200);
		printf("Main Proc: %d₩n", nCnt);
	}

	// 스레드 종료를 무한히 기다림
	WaitForSingleObject(hThread, INFINITE);

	// 스레드 자원 반납
	::CloseHandle(hThread);
	return 0;
}
```

<nw11_thread.zip>

2.2 임계 구역(Critical Section)

동기화라는 것은 프로세스가 컴퓨터의 자원을 순서대로 접근(Access)하거나 처리(Process)하는 것입니다. 데이터를 공유하는 멀티-스레드 프로그램은 동기화 환경을 성정하지 않고 프로그램이 진행이 된다면 경쟁 상태(Racing Condition), 교착 상태(Dead Lock)의 문제가 발생합니다.
경쟁 상태는 공유된 자원을 동시에 접근을 시도 했을 발생하는 문제이고 교착 상태는 자원을 점유하고 있으면서 다른 자원의 점유를 요청할 때 무한 대기 상태로 빠지는 현상입니다.
경쟁 상태의 예를 들면 A 프로세스는 값을 올리고 B 프로세스는 값을 내리는 역할을 한다고 합시다. A 프로세스가 공유되는 자원의 값을 올리고 일을 처리하는 순간 B 프로세스가 값을 내리게 되면 결과는 완전히 벗어나게 됩니다.
교착 상태의 예는 A 프로세스가 C를 점유하고 있으면서 B 프로세스가 점유하고 있는 D를 요청한다고 합시다. 만약 B 프로세스도 A가 점유하고 있는 C를 요청한다면 서로가 데이터 점유를 해제하지 않는 상황이라면 무한 대기에 빠지게 됩니다. 동기화에 대한 문제를 해결하기 위해서 윈도우 API는 임계 구역(Critical Section), 뮤텍스(Mutex), 세마포어(Semaphore), 그리고 이벤트(Event) 4개의 객체를 제공합니다.

임계 구역은 유저 모드를 기반으로 하는 동기화 방법으로 오직 스레드 사이에서만 사용할 수 있습니다. 임계 구역을 설정하는 것은 은행에서 번호표를 받고 기다리는 것과 동일합니다. 앞에 있는 고객의 일 처리가 끝날 때 까지 뒤에 있는 사람들은 기다리듯이 임계 구역에 스레드가 진입 하고 임계 구역을 빠져 나오기 전까지 다른 스레드는 대기 상태가 됩니다.

```
CRITICAL_SECTION m_CS={0};              // 임계 구역 객체 생성
…
InitializeCriticalSection(&m_CS);       // 임계 구역 초기화
…
EnterCriticalSection(&m_CS);            // 임계 구역 진입: 다른 프로세스는 기다림

// 명령문 실행

LeaveCriticalSection(&m_CS);             // 임계 구역을 빠져 나감
…
DeleteCriticalSection(&m_CS);           // 임계 구역 해제
```

<임계 구역 함수들과 사용 예>

경쟁 상태의 문제점을 보이고 이것을 임계 구역으로 해결하는 예제를 만들어 봅시다. 다음은 주-스레드와 2개의 자식 스레드를 사용해서 공유 되는 전역 변수의 값을 증가 시키는 예제입니다.

```
…
CRITICAL_SECTION m_CS={0};                    // critical section

DWORD WINAPI WorkThread(void*)
…
        while(5000>nLimit++)
        {
                EnterCriticalSection(&m_CS);    // Lock: 다른 프로세스는 기다림
                ++m_Total;                      // 명령문 실행: global value를 변경
                sprintf(m_sMsg, "th_%3d         %3d₩n",(int)dId, m_Total);
                fprintf(m_fp, m_sMsg);
                LeaveCriticalSection(&m_CS);    // UnLock
        }
        return 0;
…
int main()
{
        InitializeCriticalSection(&m_CS);       // create critical section
        …
        for(n=0; n<2; ++n)                      // test Thread 2개 생성
                hThread[n] = (HANDLE)_beginthreadex(…);
        …
        while(5000>nLimit++)
        {
                EnterCriticalSection(&m_CS);    // Lock: 다른 프로세스는 기다림
                ++m_Total;                      // 명령문 실행: global value를 변경
                sprintf(m_sMsg, "MainProc       %3d₩n", m_Total);
                fprintf(m_fp, m_sMsg);
                LeaveCriticalSection(&m_CS);    // UnLock
        }
…
        DeleteCriticalSection(&m_CS);           // release critical section
…
```

<nw12_cmse_lock.zip>

임계 구역을 사용하면 순차적으로 값이 증가하겠지만 임계 구역 사용을 전부 주석 처리해서 임계 구역을 사용하지 않는다면 실행 할 때마다 다르겠지만 log.txt에 기록된 내용을 보면 프로세스가 교체되는 시점에 다음과 같이 값이 증가되지 않고 이전 값을 출력하고 있음을 볼 수 있습니다.

```
th_n564         2460
tainProc        2460
…
th_nPr4         2476
thinProc        2476
…
MainProc        2479
th_nPr4c        2479
MainProc        2481
th_nP64         2481
…
```

<log.txt>

2.3 뮤텍스(Mutex), 세마포어(Semaphore)

임계 구역은 사용하는 주 이유는 유저 모드 동기화이기 때문에 커널 객체를 사용하는 커널 모드 동기화보다 빠릅니다. 하지만 장점 대신 단점도 존재하는데 커널 모드 동기화 객체들은 Time-out을 설정할 수 있지만 임계 구역은 시간을 설정할 수 없어서 무한 대기 상태가 될 수도 있습니다. 또한 윈도우의 커널 모드 동기화 객체들은 이름을 가질 수 있어서 생성된 커널 모드 동기화 객체들을 찾아서 사용할 수 있습니다. 이런 이유들로 인해 일부 특정한 동기화에서 커널 모드 동기화를 사용하기도 합니다.

뮤텍스는 간단히 임계 구역을 대신해서 사용할 수 있는 커널 모드 동기화 객체로 임계 구역과 동일하게 스레드 사이에서 사용할 수 있고 더 나아가 프로세스 사이에서도 사용될 수 있습니다.

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpAttribs, BOOL bInitialOwner, LPCTSTR lpName)
```

CreateMutex()는 뮤텍스 객체를 생성하는 함수로 첫 번째 인수는 보안 속성을 지정하고 보통 NULL로 설정합니다. 두 번째 인수는 뮤텍스를 생성할 때 소유 여부 입니다. TRUE이면 이 함수를 호출한 스레드가 뮤텍스를 소유하고 non-signal 상태로 만들어서 다른 프로세스가 소유하지 못하도록 합니다. 보통 FALSE로 설정하며 signal 상태가 됩니다. 세 번째 인수는 뮤텍스의 이름입니다. 만

약 뮤텍스가 이름을 가지면 이름으로 뮤텍스를 공유 할 수 있습니다. 만약 뮤텍스의 이름이 NULL로 설정되어 있으면 뮤텍스는 프로세스 사이에서 공유가 안되고 오직 동일한 프로세스 내에서만 사용됩니다.

```
HANDLE OpenMutex(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName)
```

OpenMutex()는 이름으로 생성된 뮤텍스를 찾는 함수로 첫 번째 인수는 SYNCHRONIZE를 사용해서 같은 이름의 뮤텍스를 찾습니다.

```
BOOL ReleaseMutex(HANDLE hMutex)
```

ReleaseMutex()는 non-signal 뮤텍스를 signal 뮤텍스로 만듭니다. 대기 시간을 INFINITE로 설정하면 [WaitForSingleObject, ReleaseMutex] 블록은 마치 Lock/Unlock과 같으며 임계 구역의 [EnterCriticalSection, LeaveCriticalSection] 과 동일합니다. 이전 임계 구역 예제를 뮤텍스로 바꾸면 다음과 같습니다.

```
…
HANDLE  m_hMx   = NULL;                 // mutex handle

DWORD WINAPI WorkThread(void*)
…
        while(5000>nLimit++)
        {
                WaitForSingleObject(m_hMx, INFINITE);   // Lock
                ++m_Total;              // 명령문 실행: global value를 변경
                …
                ReleaseMutex(m_hMx);                    // UnLock
        }
        return 0;
}

int main()
{
        // 동일한 mutex가 있는지 확인
        m_hMx = OpenMutex(SYNCHRONIZE, FALSE, "My thread mutex");
        if(NULL == m_hMx)
                m_hMx = CreateMutex(NULL, FALSE, "My thread mutex");
        …
```

```
        // test Thread 2개 생성
        for(n=0; n<2; ++n)
                hThread[n] = (HANDLE)_beginthreadex(…);
        …
        while(5000>nLimit++)
        {

                WaitForSingleObject(m_hMx, INFINITE); // Lock
                ++m_Total;
                …
                ReleaseMutex(m_hMx);            // UnLock
        }

        CloseHandle(m_hMx);                     // release mutex
…
```

<nw12_cmse_lock.zip>

세마포어 객체는 계수기가 있는 동기화 객체 입니다. 계수기는 [0, 최대 계수 값] 사이의 값으로 0이면 non-signal, [1, 최대값]사이에서는 signal 상태가 됩니다.

```
HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES lpAttribs
                        , LONG lInitialCount, LONG lMaximumCount, LPCTSTR lpName)
HANDLE OpenSemaphore(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName)
BOOL ReleaseSemaphore( HANDLE hSemaphore, LONG lReleaseCount, LPLONG lpPreviousCount)
```

동기화에 대한 커널 객체들과 관련된 함수의 형태는 거의 비슷합니다. 뮤텍스와 마찬가지로 이름 있는 세마포어를 만들 수 있으며 이름으로 이미 생성된 세마포어를 OpenSemaphore() 함수로 찾을 수도 있습니다. 세마포어를 생성하는 CreateSemaphore() 함수의 2 번째 인수는 시작 인덱스를 3 번째 인수는 최대 계수 값을 설정합니다. 시작 값은 최대값으로 설정하는 것이 보통이며 WaitForSingleObject() 함수를 만나면 계수기가 감소하고 0이 되면 non-signal 상태가 됩니다. ReleaseSemaphore()는 2 번째 인수만큼 계수기 값을 증가 시킵니다. 세마포어의 최대 값, 시작 값, 그리고 ReleaseSemaphore의 증가 값을 1로 설정하면 Mutex와 동일하게 동작 합니다.

```
…
HANDLE  m_hSp   = NULL;                         // Semaphore handle

DWORD WINAPI WorkThread(void*)
…
```

```
        WaitForSingleObject(m_hSp, INFINITE);   // Lock
        …
        ReleaseSemaphore(m_hSp, 1, NULL);       // UnLock
…

int main()
{
        // 동일한 semaphonre가 있는지 확인
        m_hSp = OpenSemaphore(SYNCHRONIZE, FALSE, "My thread semaphore");
        if(NULL == m_hSp)
                m_hSp = CreateSemaphore(NULL, 1, 1, "My thread semaphore");
        …
        while(…)
        {
                WaitForSingleObject(m_hSp, INFINITE);   // Lock
                …
                ReleaseSemaphore(m_hSp, 1, NULL);       // UnLock
        }
        CloseHandle(m_hSp);                             // release semaphore
…
```

<nw12_cmse_lock.zip>

2.4 이벤트(Event) 객체

여러 동기화 객체 중에서 네트워크 게임 프로그래머는 다른 동기화 객체 보다 Event를 가장 잘 알아야 합니다. 이벤트 객체는 다음의 WSAEventSelect I/O 모델과 Overlapped I/O 모델에서 두루 사용되는데 수동적인 소켓을 이벤트 객체와 바인딩 해서 소켓에서 발생하는 사건들을 이벤트 객체로 알아낼 수 있으며 Mutex, Semaphore 등을 Event 객체로 대처할 수도 있기 때문입니다.

```
HANDLE  m_hEv   = NULL;                         // Event handle
DWORD WINAPI WorkThread(void*)
…
        WaitForSingleObject(m_hEv, INFINITE);   // Lock
        …
        SetEvent(m_hEv);                        // UnLock
…
```

```
int main()
{
	// 동일한 event가 있는지 확인
	m_hEv = OpenEvent(SYNCHRONIZE, FALSE, "My thread event");
	if(NULL == m_hEv)
		m_hEv = CreateEvent(NULL, FALSE, TRUE, "My thread event");
	…
	while(…)
	{
		WaitForSingleObject(m_hEv, INFINITE);   // Lock
		…
		SetEvent(m_hEv);                        // UnLock
	}
	CloseHandle(m_hSp);                         // release event
…
```

<nw12_cmse_lock.zip>

이벤트 객체는 시스템의 이벤트를 전달할 수 있는 동기화 객체입니다. 특히, 소켓과 같이 프로세스가 주기적으로 이벤트 내용을 확인하는 수동적인 객체들에 대해서, 능동적으로 프로세스에게 메시지를 전달할 수 있도록 한 것이 이벤트 객체로 다른 동기화 객체가 할 수 없는 유연성이 높은 객체 입니다.

이벤트 객체의 종류는 크게 자동 리셋 모드(auto reset mode) 이벤트와 수동 리셋 모드(manual reset mode) 두 종류가 있습니다. 셋(set)은 컴퓨터 하드웨어의 상태가 신호 상태(signal)이고, 리셋(reset)은 비신호(non-signal) 상태를 나타냅니다.
자동 리셋 객체는 대기 상태가 해제 될 때 자동으로 비신호 상태로 전환 되는 객체입니다. 신호 대기 함수(wait function)인 WaitForSingleObject() 함수나 WaitForMultipleObjects()와 같은 함수들은 지정된 객체들이 신호상태가 될 때까지 기다리다가 신호상태가 되면 반환을 합니다.
이와 같은 함수들에 의해 함수의 호출 이후 지정된 객체의 신호 상태 그대로 유지되는 객체들을 수동 리셋 모드 객체라 하고 자동으로 비신호 상태로 전환 되는 객체를 자동 리셋 모드 객체라 하며 자동 리셋 모드 객체는 스레드, 뮤텍스, 세마포어 등이 있습니다.

수동 리셋 모드 객체가 필요한 경우는 여러 개의 동기화 객체를 하나의 대기 함수에서 사용할 때입니다. 대표적으로 WaitForMultipleObjects() 함수는 동기화 객체의 리스트를 사용하고 리스트 중에서 신호 상태의 객체 인덱스를 반환 합니다.
그런데 이 함수는 동시에 신호 상태가 된 객체들에 대해서는 가장 낮은 인덱스만 반환을 해서 인

덱스가 높은 쪽으로 신호 상태의 객체들을 검사해야 합니다. 이 검사에서도 WaitForMultipleObjects() 또는 WaitForSingleObject() 함수를 사용해야만 하기 때문에 객체가 자동 리셋이면 첫 번째 호출에서 비신호 상태로 전환되기 때문에 두 번째로 인덱스를 검사하는 과정에서 통과해 버리게 됩니다.
여기서는 간단한 예제만 보이고 실제로 네트워크에서 사용하는 방법은 WSAEventSelect 에서 다시 보겠습니다.

```
HANDLE CreateEvent(LPSECURITY_ATTRIBUTES lpAttribs, BOOL bManualReset
                        , BOOL bInitialState, LPTSTR lpName)
HANDLE OpenEvent(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName)
```

이벤트 객체는 CreateEvent() 함수로 생성됩니다. 두 번째 인수를 TRUE로 설정하면 수동 리셋 이벤트를, FALSE로 설정하면 자동 리셋 모드 이벤트로 생성됩니다. 세 번째 인수는 이벤트의 시작 상태로 FALSE는 비신호(non-signal) TRUE는 신호(signal) 상태가 됩니다.

```
BOOL SetEvent(HANDLE hEvent)
BOOL ResetEvent(HANDLE hEvent)
```

이벤트는 강제로 신호/비신호 상태로 만들 수 있습니다. SetEvent() 함수는 신호 상태로, ResetEvent() 함수는 비신호 상태로 이벤트 객체를 설정합니다. 특히, 수동 리셋 모드 이벤트는 wait function을 빠져 나와도 신호 상태가 되기 때문에 필요에 따라 ResetEvent() 함수를 사용할 수도 있습니다.

네트워크에서 소켓과 이벤트가 바인딩(binding) 되면 소켓 입/출력의 통지를 이벤트 객체를 통해서 확인할 수 있습니다. 이벤트는 통지 받기 전에는 비신호 상태로 있다가 입/출력이 발생하면 신호 상태로 전환되는 것이 가장 일반적인 형태 입니다.
따라서 응용프로그램의 프로세스는 이벤트가 신호 상태가 될 때까지 무한히 기다리다가 신호 상태가 되면 필요한 작업을 처리하는 형태가 됩니다. 이 것을 이벤트 객체 함수들로 다음과 같이 정리할 수 있습니다.

```
HANDLE  m_hEv = NULL;   // Event Handle
…
// 응용 프로그램 작업 스레드
DWORD WINAPI WorkThread(void*)
…
while(…)
{
```

```
        …
        // 시스템에 의해 signal될 때 까지 기다림.
        WaitForSingleObject(m_hEv, INFINITE);
        // 곧바로 다른 스레드는 대기하도록 non-signal로 만듦.
        ResetEvent(m_hEv);
        // 작업 처리
        …
}
…

int main()
{
        // 이벤트 동작 흉내를 위해 maual-reset, non-signal 이벤트 생성
        // WSACreateEvent로 생성한 이벤트와 동일
        m_hEv = CreateEvent(NULL, TRUE, FALSE, NULL);

        // 응용 프로그램 작업 스레드 생성
        HANDLE hThread = (HANDLE)_beginthreadex(…, WorkThread,…);

        // 시스템을 흉내내는 프로세싱
        while(…)
        {
                …
                // 일정한 시간 간격으로 이벤트 객체를 signal로 설정
                SetEvent(m_hEv);
        }
…
```

<nw13_event_single.zip>

네트워크 프로그램에서 하나의 이벤트보다 다수의 이벤트를 사용할 경우가 많이 있습니다. 시간 대기 함수 WaitForSingleObject() 보다 WaitForMultipleObjects() 함수가 좀 더 자주 사용되는데 WaitForMultipleObjects() 함수로 WaitForSingleObject()함수를 대처할 수 있기 때문입니다.

```
DWORD WaitForMultipleObjects(DWORD nCount, const HANDLE* lpHandles
                                , BOOL bWaitAll, DWORD wait_timeout_dwMilliseconds)
DWORD WaitForSingleObject(const HANDLE h, DWORD wait_timeout_dwMilliseconds)
```

WaitForMultipleObjects (이하 WFMO), WaitForSingleObject (이하 WFSO) 함수는 대표적인 시간 대기 함수(wait function)으로 WFMO는 WFSO와 비슷하게 객체가 신호 상태(signal)가 되거나 주어진 wait timeout(1/1000초 단위) 동안 프로세스를 대기 상태로 만드는 함수 입니다.

첫 번째 인수는 감시할 객체들의 개수이며 이 값은 최대 MAXIMUM_WAIT_OBJECTS 를 넘지 못합니다. 두 번째 인수는 객체의 배열 리스트이고 이 배열의 인덱스도 MAXIMUM_WAIT_OBJECTS-1까지 입니다.

세 번째 인수는 WFMO 함수에서 가장 주의해야 할 부분으로 대기시간을 무한대(INFINITE)로 설정하고 이 세 번째 인수를 TRUE로 설정하면 배열에 저장된 객체들 모두가 신호 상태가 되어야 WFMO함수는 리턴 합니다.

세 번째 인수가 FALSE이면 배열에 저장된 객체가 하나라도 신호 상태가 되면 함수는 반환을 하고 반환 값은 'WAIT_OBJECT_0 + 배열 인덱스'가 됩니다. 따라서 반환은 [WAIT_OBJECT_0, (WAIT_OBJECT_0 + nCount - 1)] => WAIT_OBJECT_0 + [0, nCount) 범위의 값이 됩니다.

만약 2개 이상의 객체가 신호 상태가 되면 신호 상태가 된 가장 낮은 가장 낮은 인덱스 값을 반환합니다. 따라서 가장 낮은 인덱스부터 WFMO 함수로 타임 아웃을 0으로 설정하고 신호 상태인지 확인해야 합니다. 이러한 이유로 최소한 WFMO 함수를 두 번 호출하기 때문에 이벤트가 자동 리셋이면 첫 번째에서 호출에서 비신호 상태가 되어서 두 번째 WFMO() 함수 호출 때 검사가 안되므로 객체는 수동-리셋 모드 객체가 되어야 프로그램이 쉽게 작성 됩니다.

```
HANDLE  m_vEvnt[MAX_EVENT] = {0};       // Event Handle List
…
DWORD WINAPI WorkThread(void*)
…
while(1)
{
        // 신호 상태 객체를 찾음
        // [WAIT_OBJECT_0, WAIT_OBJECT_0 + nCnt - 1] 사이를 반환 받음
        nIdx  = WaitForMultipleObjects(nCnt, m_vEvnt, FALSE, INFINITE);
        nIdx -= WAIT_OBJECT_0;          // 인덱스를 재조정. [0, nCnt)
        if(!(0 <= nIdx && nIdx <nCnt))  // 프로세스 또는 이벤트 객체들 에러
                break;

        for(n = nIdx; n<nCnt; ++n)      // find all signaled event
        {
                // 이벤트 리스트의 인덱스를 찾기위해 WFMO 함수를 한 번 더
```

```
                // 호출하기 때문에 이벤트는 수동 리셋으로 만들어야 함.
                hr = WaitForMultipleObjects(1, &m_vEvnt[n], TRUE, 0);
                if(WAIT_OBJECT_0 != hr) // Error
                        continue;

                // 수동 리셋 모드이므로 비 신호 상태로 변경함
                ResetEvent(m_vEvnt[n]);
        }
}
…

int main()
…
// 수동-리셋 이벤트 객체 생성
for(i=0; i<MAX_EVENT; ++i)
        m_vEvnt[i] = CreateEvent(NULL, TRUE, FALSE, NULL);

// 작업 스레드 생성
HANDLE hThread = (HANDLE)_beginthreadex(…);
…
// 시스템 흉내내기. random으로 이벤트를 signal
while(nLimit<10)
{
        …
        n = rand()%15;
        …
        set<int >::iterator _F = evlst.begin();
        for( ; _F != evlst.end(); ++_F)
        {
                SetEvent(m_vEvnt[(*_F)]);
        }
…
```

<nw14_event_multi.zip>

2.5 Multi-Threading Network

서버-클라이언트 통신에 대한 프로그램을 작성한다면 서버는 주-스레드 이외에 Accept, Send, Receive를 처리할 3종류의 스레드가 필요합니다. 클라이언트는 서버의 Accept 대신 Connection 용 스레드로 3개의 스레드가 필요할 것 같지만 실제로 자주 연결을 끊고 재 접속하는 경우가 아니라면 Connection은 한 번만 사용되고 윈도우의 메시지나 이벤트로 처리할 수 있어서 총 2개의 스레드를 가지고 네트워크를 구성합니다.

스레드를 사용해서 단순한 1:1 통신을 완성한다면 이를 확장해서 1:n 서버-클라이언트 프로그램을 충분이 만들 수 있습니다. 또한 스레드를 사용하므로 반드시 교착상태(Dead Lock)를 고려해야 합니다.
안정성은 커널 모드 동기화 객체가 가장 좋지만 좀 더 빠른 유저 모드 동기화 객체 임계 구역(critical section)을 선택해서 간단한 1:1 통신을 구성해 보도록 합시다.

먼저 다음과 같이 동기(blocking) 모드 소켓을 사용하는 3개의 스레드 함수를 선언하고 주-스레드에서 Accept용 스레드를 생성합니다.

```
//서버
…
SOCKET  g_scLstn = 0;           // listen socket
DWORD WINAPI WorkAcc(void*);    // Accept용쓰레드
DWORD WINAPI WorkRcv(void*);    // Receive용쓰레드
DWORD WINAPI WorkSnd(void*);    // Send용쓰레드

int main()
…
InitializeCriticalSection(&m_CS);       // 임계 구역 초기화
…
hr = listen(g_scLstn, SOMAXCONN);
if( SOCKET_ERROR == hr)
        return -1;

//Accept 용 Thread 생성
g_hThAcc = (HANDLE)_beginthreadex(NULL, 0
                , (unsigned (__stdcall*)(void*))WorkAcc, NULL, 0, NULL);

// Main process
while(1)
{
```

```
        …
}
…
```

Accept 스레드는 클라이언트가 접속하면 송신용, 수신용 스레드를 생성합니다. 간단히 클라이언트가 접속했다는 메시지를 보낼 수 있도록 공통으로 사용하는 송신 버퍼에 메시지를 기록합니다.

```
DWORD WINAPI WorkAcc(void *)
…
while(1)
{
        // 클라이언트 접속을 기다린다.
        g_scCln = accept(g_scLstn, …);
        if(INVALID_SOCKET == g_scCln)
                continue;

        // 수신용 Thread 생성
        g_hThRcv = (HANDLE)_beginthreadex(NULL, 0
                , (unsigned (__stdcall*)(void*))WorkRcv, NULL, 0, NULL);

        // 송신용 Thread 생성
        g_hThSnd = (HANDLE)_beginthreadex(NULL, 0
                , (unsigned (__stdcall*)(void*))WorkSnd, NULL, 0, NULL);

        // 송신 버퍼에 접속 메시지를 기록
        EnterCriticalSection(&m_CS);
        memset(g_bufSnd, 0, MAX_BUF);
        sprintf(g_bufSnd, "Connected: %d", (int)g_scCln);
        LeaveCriticalSection(&m_CS);
}
…
```

수신을 담당하는 스레드는 recv() 함수를 중심으로 구성합니다. recv() 함수로 받은 데이터를 그대로 클라이언트에게 보내기 위해서 송신 버퍼에 서버 메시지를 추가해서 기록합니다.

```
DWORD WINAPI WorkRcv(void *)
…
```

```
while(1)
{
...
        int iRcv = 0;
        char bufRcv[MAX_BUF+4]={0};

        iRcv=recv(g_scCln, bufRcv, MAX_BUF, 0);
        if(SOCKET_ERROR == iRcv)
                break;

        else if(0 == iRcv)
                break;

        printf("Recv from Client : %d %s₩n", (int)g_scCln, bufRcv);

        // copy to send buffer
        EnterCriticalSection(&m_CS);
        memset(g_bufSnd, 0, MAX_BUF);
        sprintf(g_bufSnd, "%5d : %s", g_scCln, bufRcv);
        LeaveCriticalSection(&m_CS);
}
...
```

송신 스레드는 송신 버퍼를 검사해서 전송할 데이터가 있으면 send() 함수를 호출해서 전송합니다. 검사 판별은 문자열의 길이로 확인하며 전송이 완료되면 송신 버퍼의 내용을 null 문자('₩0')으로 초기화합니다. 이렇게 하면 송신 버퍼에 기록된 내용이 없으므로 문자열 길이가 0이 되므로 전송이 안됩니다.

```
DWORD WINAPI WorkSnd(void *pParam)
...
while(1)
{
        ...
        iLen = strlen(g_bufSnd);
        if(0 >= iLen)
                continue;
```

```
        iSnd=send(g_scCln, g_bufSnd, iLen, 0);
        if(0 >= iSnd)
        {
                break;
        }

        // clear the send buffer
        EnterCriticalSection(&m_CS);
        memset(g_bufSnd, 0, sizeof( g_bufSnd));
        LeaveCriticalSection(&m_CS);
…
```

<nw21_thread_1to1.zip>

클라이언트는 서버와 비슷하게 동기(blocking) 모드 소켓을 배경으로 접속이 성공하면 송신, 수신 스레드를 생성합니다.

```
// 클라이언트
…
DWORD WINAPI WorkRcv(void*);    // Receive용쓰레드
DWORD WINAPI WorkSnd(void*);    // Send용쓰레드

int main()
…
hr = connect(g_scHost, (SOCKADDR*)&g_sdHost, sizeof(SOCKADDR_IN));
if(SOCKET_ERROR == hr)
        return -1;


// create recv/send thread
// 수신용Thread
g_hThRcv = (HANDLE)_beginthreadex(NULL, 0
        , (unsigned (__stdcall*)(void*))WorkRcv, NULL, 0, NULL);

// 송신용Thread생성
g_hThSnd = (HANDLE)_beginthreadex(NULL, 0
        , (unsigned (__stdcall*)(void*))WorkSnd, NULL, 0, NULL);
…
```

클라이언트 수신 스레드는 서버와 거의 같습니다. recv() 함수의 반환 값에 따라서 에러, 접속 끊김, 그리고 받은 문자열 출력 세가지를 구현합니다.

```
DWORD WINAPI WorkRcv(void *)
...
while(1)
{
        ...
        if(g_scHost)
        {
                int iRcv = 0;
                char bufRcv[MAX_BUF+4]={0};

                iRcv=recv(g_scHost, bufRcv, MAX_BUF, 0);
                if(SOCKET_ERROR == iRcv)
                {
                        hr = WSAGetLastError();
                        break;
                }
                else if(0 == iRcv)
                {
                        break;
                }

                printf("Recv: %sWn", bufRcv);
        }
}
...
```

1:1 통신이므로 송신 스레드는 서버와 동일하게 구성합니다.

```
DWORD WINAPI WorkSnd(void *pParam)
...
while(1)
{
        ...
```

```
        if(0 >= iLen)
                continue;

        iSnd=send(g_scHost, g_bufSnd, iLen, 0);
        if(0 >= iSnd)
        {
                hr = WSAGetLastError();
                break;
        }

        // clear the send buffer
        EnterCriticalSection(&m_CS);
        memset(g_bufSnd, 0, sizeof( g_bufSnd));
        LeaveCriticalSection(&m_CS);
}
...
```

<nw21_thread_1to1.zip>

1:1 통신을 구현해봤습니다. 1:n 통신은 단순 echo 서버 입니다. 이것을 확장해서 클라이언트가 보낸 메시지를 모든 접속한 클라이언트에게 보내는 UDP의 broadcast와 유사한 서버를 구성해 봅시다.
먼저 여러 클라이언트를 관리할 수 있도록 소켓, 소켓 어드레스, 송신 버퍼, 수신 스레드, 송신 스레드를 기본으로 하는 구조체를 구성합니다. 여기서 소켓 어드레스는 소켓에서 getpeername() 함수로 가져올 수도 있습니다.

```
// 호스트 리스트를 저장하기 위한 구조체
struct RemoteHost
{
        SOCKET          scH;            // 소켓
        SOCKADDR_IN     sdH;            // 소켓 어드레스
        int             nBuf;           // 송신 버퍼에 저장된 문자열 길이
        char            sBuf[MAX_BUF];  // Send buffer
        HANDLE          hThRcv;         // Receive용쓰레드핸들
        HANDLE          hThSnd;         // Send용쓰레드핸들

        RemoteHost();
        RemoteHost(SOCKET s, SOCKADDR_IN* d);
```

```
        void Set(SOCKET s, SOCKADDR_IN* d, HANDLE hRcv, HANDLE hSnd);
        void Reset();
};
```

호스트 자료 구조의 Set 함수는 호스트 객체에 자료를 설정해 주는 함수 입니다. 서버는 미리 특정한 숫자만큼 전역 변수로 클라이언트 리스트를 생성해 놓습니다.

```
RemoteHost      g_rmCln[MAX_CLIENT];    // client list
```

만약 새로운 클라이언트가 접속하면 소켓, 소켓 주소, 수신 스레드, 송신 스레드를 바인딩 합니다.

```
void RemoteHost::Set(SOCKET s, SOCKADDR_IN* d, HANDLE hRcv, HANDLE hSnd)
{
        scH     = s;                            // 소켓 저장
        memcpy(&sdH, d, sizeof(SOCKADDR_IN));   // 소켓 주소 복사
        nBuf    = 0;                            //
        memset(sBuf, 0 , MAX_BUF);              // 송신 버퍼 초기화
        hThRcv= hRcv;                           // 수신용 스레드
        hThSnd= hSnd;                           // 송신용 스레드
}
```

호스트 자료 구조의 Reset 함수는 호스트 객체의 내용을 초기화 합니다. 클라이언트가 접속을 끊으면 새로운 클라이언트가 사용할 수 있도록 초기화 합니다.

```
void RemoteHost::Reset()
{
        if(scH)                         // 소켓 해제
        {
                shutdown(scH, SD_BOTH);
                closesocket(scH);
                scH = 0;
        }

        if(nBuf)                        // 버퍼 초기화
        {
                nBuf = 0;
                memset(sBuf, 0 , MAX_BUF);
        }
```

```
        if(hThRcv)
        {
                GetExitCodeThread(hThRcv, &dExit);
                if(dExit)               // STILL_ACTIVE 이면 수신 스레드 강제 종료

                        TerminateThread(hThRcv, dExit);
                ::CloseHandle(hThRcv);
                hThRcv = 0;
        }

        if(hThSnd)
        {
                GetExitCodeThread(hThSnd, &dExit);
                if(dExit)               // STILL_ACTIVE 이면 송신 스레드 강제 종료
                        TerminateThread(hThSnd, dExit);

                ::CloseHandle(hThSnd);
                hThSnd = 0;
        }
}
```

클라이언트 접속을 담당하는 WorkAcc 스레드 함수는 클라이언트가 접속하면 호스트 리스트에서 초기화 되어 있는 클라이언트 인스턴스를 찾아서 송신 스레드와 수신 스레드를 새로 생성하고 찾아낸 클라이언트 인스턴스에 소켓, 소켓 주소, 송신 스레드, 수신 스레드를 바인딩 합니다.
스레드 내부에서 클라이언트의 인스턴스를 사용할 수 있도록 _beginthreadex() 함수에 클라이언트 인스턴스 변수를 전달합니다.

```
DWORD WINAPI WorkAcc(void *pParam)
…
while(1)
{
        SOCKET scH = 0; SOCKADDR_IN sdH ={0};
        scH = accept(g_scLstn, (SOCKADDR*)&sdH, …);
…
        RemoteHost* pCln = FindNotUseHost();    // 초기화 되어 있는 인스턴스 찾기
        …
```

```
        // 데이터 수신용 Thread 생성
        HANDLE hRcv = (HANDLE)_beginthreadex(NULL, 0
                        , (unsigned (__stdcall*)(void*))WorkRcv
                        , (void*)pCln                   // 클라이언트 인스턴스 전달
                        , CREATE_SUSPENDED, NULL);

        // 데이터 송신용 Thread 생성
        HANDLE hSnd = (HANDLE)_beginthreadex(NULL, 0
                        , (unsigned (__stdcall*)(void*))WorkSnd
                        , (void*)pCln                   // 클라이언트 인스턴스 전달
                        , CREATE_SUSPENDED , NULL);

        // 클라이언트 인스턴스에, 소켓, 소켓 구조, 송신 스레드, 수신 스레드 설정
        pCln->Set(scH, &sdH, hRcv, hSnd);
        ResumeThread(hRcv);                             // 수신 스레드 재개
        ResumeThread(hSnd);                             // 송신 스레드 재개
}
…
```

데이터 수신을 담당하는 WorkRcv 스레드 함수는 특별히 새로 추가된 내용은 거의 없습니다. 클라이언트 인스턴스에 포함 된 소켓을 가지고 recv() 함수로 소켓의 수신 버퍼가 채워질 때까지 기다립니다. 만약 수신된 데이터가 있으면 접속한 모든 클라이언트에게 전송합니다.

```
DWORD WINAPI WorkRcv(void *pParam)
…
RemoteHost* pCln = (RemoteHost*)pParam;

while(0<pCln->scH)
{
        int iRcv = 0;
        char sBufRcv[MAX_BUF+4]={0};

        // 수신 데이터 기다림
        iRcv = recv(pCln->scH, sBufRcv, MAX_BUF, 0);
…
        // 송신 데이터 만들기
```

```
        INT     iLen;
        char    sSndBuf[MAX_BUF]={0};

        sprintf(sSndBuf, "%5d : %s", pCln->scH, sBufRcv);
        iLen = strlen(sSndBuf);

        // 수신 데이터 재 전송
        EchoMsg(sSndBuf, iLen);
}
...
```

데이터 송신을 담당하는 WorkSnd 스레드 함수는 클라이언트 인스턴스의 송신 버퍼에 기록된 내용이 있으면 send() 함수를 사용해서 데이터를 전송합니다. 데이터 전송이 일부만 보낼 수도 있으므로 while 루프를 사용해서 전부 보낼 때까지 send 함수를 호출합니다.

```
DWORD WINAPI WorkSnd(void *pParam)
...
RemoteHost* pCln = (RemoteHost*)pParam;

while(0<pCln->scH)
{
...
        if(1 > pCln->nBuf)
                continue;

        int iSnd = 0;
        int iTot = 0;
        while( iTot < pCln->nBuf) // 송신 버퍼의 내용을 전부 보낼 때까지 반복
        {
                iSnd = send(pCln->scH, pCln->sBuf + iTot, pCln->nBuf - iTot, 0);
                if(SOCKET_ERROR == iSnd)
                {
                        ...
                        goto END;
                }
                // 보낸 데이터 계산
                iTot += iSnd;
```

```
        }

        // 송신 버퍼 초기화
        EnterCriticalSection(&m_CS);
        pCln->nBuf = 0;
        memset(pCln->sBuf, 0, sizeof(pCln->sBuf));
        LeaveCriticalSection(&m_CS);
}
```

송신 스레드는 버퍼에 쌓여 있을 때만 전송합니다. 따라서 송신 버퍼에 데이터를 채워줄 함수가 필요합니다. 여기서는 broadcast가 목적이므로 접속한 모든 클라이언트의 송신 버퍼를 채우기 위한 함수를 다음과 같이 작성합니다. 송신 스레드에 의해 버퍼가 사용될 수 있기 때문에 임계 구역을 설정해서 송신 버퍼의 내용을 설정합니다.

```
void EchoMsg(char* s, int len)
...
// 임계 구역 설정
EnterCriticalSection(&m_CS);
RemoteHost* pCln;
for(int i=0; i<MAX_CLIENT; ++i)
{
        pCln = &g_rmCln[i];

        // 소켓이 설정되어 있지 확인
        if(0 >= pCln->scH)
                continue;

        // 데이터 복사
        memset(pCln->sBuf, 0, sizeof(pCln->sBuf));
        strcpy(pCln->sBuf, s);
        pCln->nBuf = len;
}
LeaveCriticalSection(&m_CS);
...
```

<nw22_thread_cs.zip>

3 Asynchronous Notify I/O

지금까지 우리는 BSD 소켓을 가지고 TCP 소켓을 구현해 보았습니다. 연결 지향 소켓인 TCP 소켓을 1:1 통신은 쉽게 구현할 수 있었습니다. 하지만 1:n 통신을 하기 위해 비동기(nonblocking) 소켓을 사용해서 주기적으로 send, recv를 반복하거나 송신, 수신 스레드를 사용했었습니다.
이런 방법이 도입된 이유는 키보드나 마우스 같은 능동적인 객체들은 자신의 signal이 변화하면 능동적으로 시스템에 interrupt를 요청해서 변화된 내용을 전달하지만 소켓과 같은 수동적인 객체들은 시스템이 이들 객체의 신호 상태를 계속 검사해서 변화가 있으면 처리해야 하기 때문입니다. 따라서 주기적으로 polling 방식으로 수동적인 객체들의 신호 상태를 점검하는 별도의 스레드가 있어야 이들을 처리할 수 있습니다.

개별적으로 접속한 모든 클라이언트에게 송수신 스레드를 만들어주는 것은 전체 접속 인원이 아주 작을 때만 필요한 방법입니다. 스레드가 많아지면 문맥 교환(context switching)이 자주 발생되어 효율이 떨어집니다.
우리는 여기서 몇 가지 생각해봐야 할 내용이 있습니다. 먼저 소켓 입출력은 CPU의 연산 보다 빈번한 일이 아니라는 것입니다. 특정한 숫자만큼 여러 소켓을 하나로 묶어서 입출력을 처리한다면 스레드의 숫자를 줄여서 효율이 높아질 것입니다.
두 번째는 소켓을 능동적인 객체로 전환하는 것입니다. 소켓과 같은 수동적인 객체를 이벤트 객체와 바인딩 해서 사용하면 응용 프로그램이 입출력 이벤트를 처리할 수 있도록 이벤트 객체가 인식해줍니다. 따라서 스레드가 불필요하게 항상 활동하는 것이 아니라 소켓 이벤트가 발생할 때만 재개하게 되어 CPU의 효율을 높여 줍니다.
결론적으로 스레드를 줄이고 입출력이 필요할 때만 스레드가 활동하게 된다면 서버는 같은 일을 하면서도 시스템 자원을 적게 사용하기 때문에 클라이언트의 요청을 좀 더 빠르게 처리하고, 고속으로 데이터를 전송할 수 있게 됩니다.

이런 내용들은 사실 소켓과는 무관하고 시스템의 사양(specification)에 달려 있는 문제 입니다. 윈도우 시스템은 위와 같은 내용을 구현할 수 있도록 Select, AsyncSelect, EventSelect, Overlapped I/O 방식을 지원해 주고 있습니다.

윈속 입출력 모델	모바일	PC, 서버
Select	CE 1.0 이상	95 이상, NT 이상
AsyncSelect	지원 안 함	95 이상, NT 이상
EventSelect	CE .NET 4.0 이상	95 이상, NT 3.51 이상
Overlapped	CE .NET 4.0 이상	95 이상, NT 3.51 이상

Completion Port	지원 안 함	2000, XP 이상, NT 3.5 이상

<윈속 I/O 모델과 지원 운영체제>

윈속의 I/O 모델 중에서 Select, AsyncSelect, EventSelect는 비동기 통지(asynchronous notify)가 가능한 모델입니다. 비동기 통지란 비동기 객체를 사용해서 작업 처리를 요청한 후에 작업이 완료될 때까지 대기하지 않고 빠져 나와 이후에 결과를 통보 받는 것입니다.
따라서 비동기 통지를 이용하려면 비동기 소켓을 사용해야 하는데 Select 는 소켓 설정 함수를 호출해서 직접 설정하고 AsyncSelect, EventSelect는 각각 WSAAsyncSelect, WSAEventSelect 함수를 호출하면 자동으로 비동기 소켓으로 전환 됩니다.

3.1 Select I/O 모델

Select I/O 모델은 TCP/IP가 유닉스 시스템에서 최초로 구현이 되었고, 여러 소켓을 하나로 묶어서 입출력의 이벤트를 검사하고 처리했었습니다. 유닉스의 select() 함수 인터페이스를 그대로 사용하기 때문에 select i/o 방식으로 부릅니다.

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

단순히 select() 함수는 WaitForMultipleObjects() 와 비슷하게 신호 상태의 객체를 확인하기 위한 시간 대기 함수(waiting function) 입니다.
윈도우 시스템은 select() 함수 이외에 자신들의 시스템에 최적화된 멋진 시간 함수들이 많이 있어서 select() 함수로 입출력을 확인하는 일이 별로 없을 것이라고 생각할 수도 있지만 스레드가 지원되지 않은 임베디드(Embedded) 환경이나 입출력이 빈번하지 않고 멀티-스레드 환경이 필요 없는 프로그램에서도 의외로 자주 사용됩니다.

첫 번째 인수는 파일 디스크립터(file descriptor)의 숫자로 소켓의 개수이며 BSD 소켓과 사양을 맞추기 위해서 사용됩니다. 윈도우 시스템에서는 0으로 설정합니다. 두 번째 인수부터 각각 읽기 소켓 집합, 쓰기 소켓 집합, 예외처리 집합을 설정합니다.
마지막 인수는 이 함수를 최대로 기다리는 시간을 설정하는 인수이며, NULL일 경우 blocking 동작이 진행 됩니다.
Select 방식을 사용한다는 것은 멀티-스레딩 지원이 없거나 이용하지 않는 다는 것을 의미합니다. 따라서 소켓은 동기 모드 보다 비동기 모드가 적합하고 send는 별도로 처리하고 select 함수는 접속과 수신에만 사용하도록 합니다.
select 함수를 사용할 때에 적당히 기다리는 시간(wait time out)을 설정하는 것이 좋은데 속도보다는 안정성이라면 프로세스 사이클로 정할 수도 있지만 PeekMessage() 함수를 사용하는 것처럼

입출력을 확인하고 다른 작업을 계속하는 경우라면 timeout을 0으로 설정합니다.

```
typedef struct fd_set {
    u_int  fd_count;             // 배열에 저장된 소켓의 숫자
    SOCKET fd_array[FD_SETSIZE]; // 소켓 배열
} fd_set;
```

fd_set 자료 구조는 집합에 포함된 소켓(또는 파일 디스크립터)의 개수를 저장합니다. fd_array는 집합에 저장된 소켓의 배열로 최대 FD_SETSIZE로 크기가 정해져 있으며 FD_SETSIZE는 현재 64로 정해져 있습니다.
fd_set를 사용하기 위해서 입출력을 감시할 소켓들을 fd_array에 0번부터 순차적으로 저장해야 하는데 수동으로 직접 채우는 것보다 윈속 헤더파일에서 제공된 매크로를 사용하는 것이 좀 더 편리합니다.

```
FD_CLR(fd, set), FD_SET(fd, set), FD_ZERO(set), FD_ISSET(fd, set)
```

FD_CLR 매크로는 첫 번째 인수로 지정된 소켓이 집합에서 소켓을 저장하고 있는 배열에 존재하면 이 소켓을 배열에서 지우고 지워진 위치에서부터 저장된 개수만큼 앞쪽으로 소켓을 이동시킵니다. 그리고 집합에 저장된 전체 소켓 개수 값을 1 감소 시킵니다. FD_SECT 매크로는 소켓을 저장하고 있는 집합의 배열에 전체 소켓 개수만큼 조사해서 같은 소켓이 없으면 소켓을 비어있는 배열에 저장하고 크기를 1 증가 시킵니다. FD_ZERO 매크로는 집합의 내용을 전부 지우는 기능으로 전체 소켓 개수를 0으로 구성합니다. FD_ISSET 매크로는 집합에 소켓이 저장되어 있는지 확인하는 매크로입니다. select() 함수의 반환은 실패: SOCKET_ERROR, 타임아웃: 0, 성공: 전체 소켓 핸들을 반환합니다.

본격적으로 select 함수를 사용해서 서버의 접속과 수신 부분을 작성해 봅시다. 먼저 리슨 소켓, 읽기 집합을 전역 변수로 구성합니다.

```
// 서버
...
SOCKET  g_scLstn=0;     // 소켓
FD_SET  g_fdSet;        // 읽기 집합 소켓 FD_SET
INT     FrameMove();    // 소켓 I/O 처리
...
```

주- 스레드 main() 함수는 리슨(listen) 소켓을 만들고 나서 리슨 상태가 되기 전에 비동기 소켓으로 전환합니다. 다음으로 읽기 집합을 초기화하고 리슨 소켓에서 발생하는 accept를 비동기

통지로 받기 위해서 읽기 집합에 리슨 소켓을 저장합니다. while 루프에서 주기적으로 네트워크의 입출력을 담당하는 FrameMove() 함수를 호출하고, 자신의 일들을 처리합니다.

```
int main()
…
// create the listen socket
g_scLstn=socket(…);
…
// 비동기 소켓 모드로 설정
u_long on =1;
hr = ioctlsocket(g_scLstn, FIONBIO, &on);
…
hr =listen(…);
…
// 읽기 소켓 집합 초기화
FD_ZERO(&g_fdSet);

// 비동기 Notify가 될 수 있도록 읽기 집합에 리슨 소켓 저장
FD_SET(g_scLstn, &g_fdSet);

while(1)
{
        // 주기적으로 네트워크 입출력 감시
        if(FAILED(FrameMove()))
                break;
                …
}
…
```

<nw31_select.zip>

네트워크 입출력을 담당하는 FrameMove() 함수에서 select() 함수의 인수로 사용할 임시 읽기 소켓 집합 변수를 할당하고 저장된 읽기 소켓 집합을 복사합니다. 임시 읽기 소켓집합과 0으로 설정한 대기 시간을 인수로 사용해서 select() 함수를 호출하고 바로 빠져 나오도록 합니다.

```
INT FrameMove()
…
FD_SET  fdsTmp;         // 감시할 임시 소켓 집합 생성
```

```
TIMEVAL timeout={0};    // 타임아웃 구조체

fdsTmp = g_fdSet;       // 저장된 읽기 소켓 집합을 임시 읽기 소켓 집합으로 복사
timeout.tv_sec  = 0;    // 대기 시간을 0으로 설정
timeout.tv_usec = 0;

// 임시 읽기 소켓 집합을 검사
hr = select(0, &fdsTmp, NULL, NULL, &timeout);
if(SOCKET_ERROR == hr)
{
        // 에러 확인
        hr = WSAGetLastError();
        return -1;
}

// 타임 아웃을 초과하면 함수를 빠져 나감
if(0 == hr)
        return 0;

// 소켓 이벤트가 발생했는지 전체 검사
for(UINT i=0; i<g_fdSet.fd_count; ++i)
{
        // 해당 소켓을 찾음
        if(!FD_ISSET(g_fdSet.fd_array[i], &fdsTmp))
                continue;

        // 리슨 소켓에 발생한 이벤트는 Accept 임. 따라서 accept 함수로 클라이언트의
        // 소켓을 얻어내고 이를 읽기 집합에 저장한다.
        if(g_scLstn == g_fdSet.fd_array[i])
        {
                // 클라이언트 소켓 얻기
                SOCKET scCln=0;
                scCln = accept(g_scLstn, NULL, NULL);

                // 읽기 집합에 저장. g_fdSet.fd_count 1 증가
                FD_SET(scCln, &g_fdSet);
                ...
```

```
                // for 루프를 계속 진행
                continue;
        }

        // 리슨 소켓 이외의 다른 소켓은 전부 recv() 함수로 확인
        …
        iRcv = recv(g_fdSet.fd_array[i], sBufRcv, MAX_BUF, 0);

        // SOCKET_ERROR 또는 받은 데이터가 0이면 close socket
        if(0 >= iRcv)
        {
                …
                // 소켓 해제
                closesocket(g_fdSet.fd_array[i]);

                // 읽기 집합에서 해당 소켓을 제거
                FD_CLR(g_fdSet.fd_array[i], &g_fdSet);
                …
                continue;
        }


        // 만약 보낼 데이터가 있으면 여기서 보낸다
        …
        EchoMsg(sBufSnd, iSnd);
}
…
```

<nw31_select.zip>

접속한 모든 클라이언트에게 데이터를 보내고자 한다면 읽기 집합에 저장된 소켓을 사용해서 같은 메시지를 전부 보냅니다.

```
void EchoMsg(char* s, int iLen)
{
        // 접속한 모든 클라이언트에게 서버 메시지 전송
        for(UINT j=0;j<g_fdSet.fd_count; ++j)
        {
```

```
                if(g_scLstn != g_fdSet.fd_array[j])
                        send(g_fdSet.fd_array[j], s, iLen, 0);
        }
}
```

<nw31_select.zip>

select() 함수를 사용하는 클라이언트는 서버와 유사하게 connection후에 비동기 소켓으로 전환합니다. 읽기 집합을 초기화하고, 소켓을 읽기 집합에 추가합니다. 입출력 담당 함수 FrameMove()를 주기적으로 호출합니다.

```
// 클라이언트
…
SOCKET  g_scHost=0;                                     // 소켓
FD_SET  g_fdSet;                                        // 소켓FD_SET
…
INT     FrameMove();
…
int main()
…
// create the host socket
g_scHost=socket(AF_INET, SOCK_STREAM, 0);
…
hr = connect(g_scHost, (SOCKADDR*)&sdHost, sizeof(SOCKADDR_IN));
…
// 비동기 소켓
u_long on =1;
hr = ioctlsocket(g_scHost, FIONBIO, &on);

// 읽기 집합 초기화
FD_ZERO(&g_fdSet);

// 읽기 집합 카운터를 1 증가
FD_SET(g_scHost, &g_fdSet);

while(g_scHost)
{
        if(FAILED(FrameMove()))
```

```
                break;
}
...
```

<nw31_select.zip>

클라이언트는 데이터 수신만 처리하면 되어 구현하기가 수월합니다. 서버와 마찬가지로 임시 읽기 집합에 읽기 집합을 복사하고 대기시간을 0으로 설정해서 select() 함수를 호출 합니다.

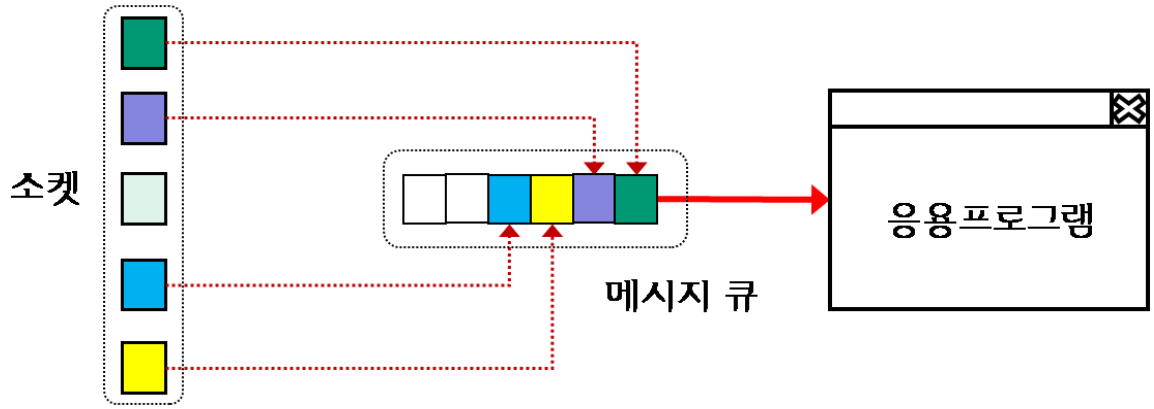
```
INT FrameMove()
...
FD_SET  fdsTmp;
TIMEVAL timeout;
...
// 임시 읽기 집합에 읽기 집합 복사. 대기 시간 = 0 으로 설정
fdsTmp = g_fdSet;
timeout.tv_sec = 0;
timeout.tv_usec= 0;
hr = select(0, &fdsTmp, NULL, NULL, &timeout);
if(SOCKET_ERROR == hr)
        return -1;

// time out
if(0 == hr)
        return 0;

// 설정 되어 있지 않음
if(!FD_ISSET(g_scHost, &fdsTmp))
        return 0;

int     iRcv=0;
char    sBufRcv[MAX_BUF+4]={0};
iRcv = recv(g_scHost, sBufRcv, MAX_BUF, 0);

// 소켓 종료
if(0>=iRcv)
        return -1;
```

```
// 수신된 데이터 출력
printf("Recv: %s₩n", sBufRcv);
...
```

<nw31_select.zip>

3.2 AsyncSelect I/O

앞에서 단순히 스레드만을 사용해서 구현한 서버보다 Select I/O 모델이 시스템의 자원을 덜 소비하지만 여전히 문제는 select 함수가 처리할 수 있는 소켓의 개수가 최대 FD_SETSIZE (64개)로 정해져 있고, 비동기 소켓을 수동으로 설정해야 하고, 대기 시간을 0으로 설정해서 시스템의 프로세서를 계속 사용하고 있다는 것입니다.
윈도우 시스템은 메시지 방식으로 응용프로그램과 시스템이 데이터를 주고 받습니다. 만약 네트워크 메시지를 응용프로그램에게 전달할 수 있다면 메시지 방식으로 프로그램이 가능하고 따라서 프로그램을 작성하는 사람은 네트워크의 전반적인 이해가 없어도 단순히 메시지를 처리하는 프로그램 과정에 지나지 않아 쉽게 네트워크 입출력을 구현할 수 있습니다.



<AsyncSelect I/O 모델>

AsyncSelect I/O 모델은 WSAAsyncSelect() 함수를 사용하는 입출력 모델입니다. WSAAsyncSelect() 함수는 소켓에 대한 네트워크 이벤트를 윈도우 시스템 메시지 기반으로 통지를 요청합니다.

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

간단히 정리한다면 WSAAsyncSelect 함수는 소켓에서 발생하는 I/O 이벤트를 윈도우 시스템이 hWnd로 동작하는 메시지 프로시저 함수로 네트워크 메시지를 전달할 수 있도록 도와주는 함수입니다.
세 번째 인수는 네트워크 이벤트가 발생했을 때 응용프로그램의 메시지 프로시저 함수에서 처리할 메시지 입니다.

네 번째 인수는 응용 프로그램이 수신할 소켓에서 발생하는 이벤트의 비트 조합입니다. 네트워크 이벤트는 다음 표에서 제공한 것을 가지고 조합해서 사용합니다.

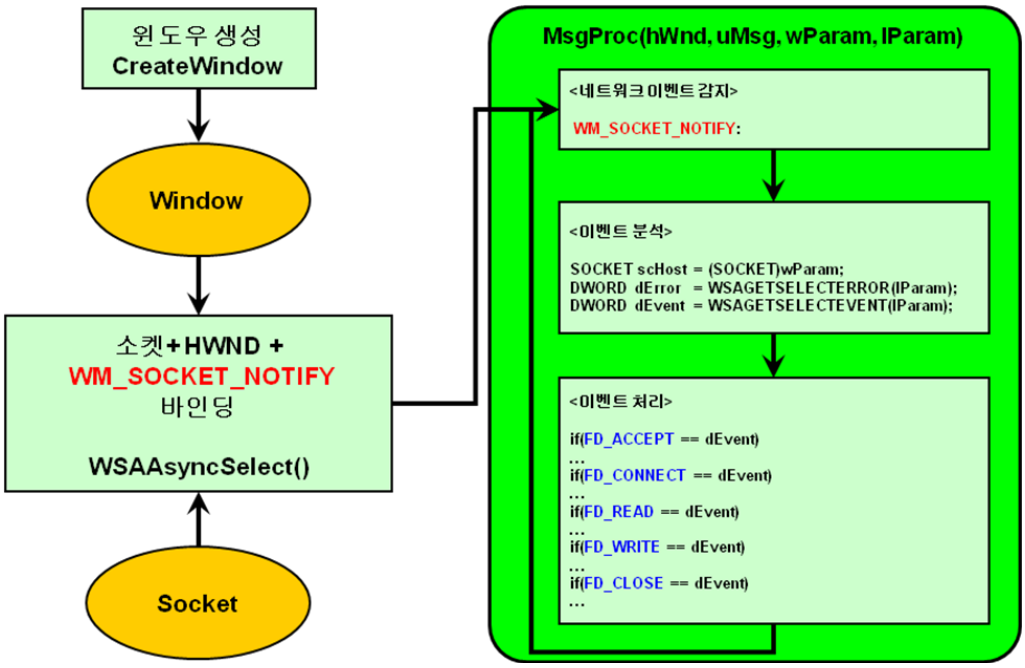
네트워크 이벤트	내용
FD_ACCEPT	새로운 클라이언트 접속
FD_CONNECT	서버 커넥션 완료
FD_CLOSE	접속 종료
FD_READ	수신된 데이터가 존재
FD_WRITE	송신된 데이터가 존재
FD_OOB	OOB(Out of Bound: 긴급 메시지)가 존재

<네트워크 이벤트 상수>

예를 들어 서버의 리슨 소켓이 클라이언트의 접속과 자신의 종료를 메시지로 받고 싶다면 비트 조합은 lEvent = (FD_ACCEPT|FD_CLOSE) 로 합니다. 또 다른 예로 클라이언트가 CONNECTION, RECV, CLOSE를 메시지로 받는다고 한다면 lEvent = (FD_CONNECT|FD_READ|FD_CLOSE) 가 됩니다.

```
WSAAsyncSelect(g_scListen, hWnd, WM_SOCKET_NOTIFY, FD_ACCEPT|FD_CLOSE);
```

<Listen 소켓을 WSAAsyncSelect() 함수에 연결한 예>



<WSAAsyncSelect() 함수를 사용한 네트워크 프로그램 구조>

WSAAsyncSelect() 함수는 입력한 소켓을 비동기 모드 소켓으로 전환합니다. 따라서 모든 소켓 입출력에 대해서 요청한 함수의 반환 값이 SOCKET_ERROR이면 반드시 WSAGetLastError() 함수로 에러의 원인을 파악해야 합니다. 특히, 에러 코드가 WSAEWOULDBLOCK의 경우, 입출력이 진행됨을 의미하기 때문에 진행 중으로 처리해야 합니다.

AsyncSelect 모델은 메시지 프로시저에서 전부 처리가 되므로 WSAAsyncSelect()함수 이외에 특별한 함수는 없습니다. 남아있는 것은 메시지 처리 부분으로 네트워크 이벤트가 발생하면 메시지 프로시저 함수에서 처리하는데 이벤트를 발생한 소켓은 프로시저 함수의 세 번째 인수 WPARAM 변수로 전달되고, 이벤트는 LPARAM 변수의 LOWORD로 전달됩니다. 만약 에러가 발생했다면 LPARAM 변수의 HIWORD 로 전달 됩니다.
이들은 HIWORD, LOWORD 매크로를 가지고 편리하게 분리할 수 있지만 WSAGETSELECTERROR, WSAGETSELECTEVENT 매크로를 사용하는 것이 의미 전달로 좋아 보입니다.

```
LRESULT WINAPI WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
…
if(WM_SOCKET_NOTIFY == msg)
{
        SOCKET  scHost  = (SOCKET)wParam;               // 소켓
        DWORD   dError  = WSAGETSELECTERROR(lParam);    // 에러
        DWORD   dEvent  = WSAGETSELECTEVENT(lParam);    // 이벤트

        if(dError)      // 비 정상적인 에러 체크
                …
                return 0;

        if(FD_ACCEPT == dEvent)         // 클라이언트 접속 메시지(서버)
                accept(…);
                …
        else if(FD_CONNECT == dEvent)   // 접속 완료 메시지(클라이언트)
                …
        else if(FD_CLOSE == dEvent)     // 연결 해제 메시지
                closesocket(…);
                …
        else if(FD_WRITE == dEvent)     // 송신 메시지
                …
        else if(FD_READ == dEvent)       // 수신 메시지
```

```
                recv(…);
                …
}
```

<메시지 프로시저의 프로그램 골격>

WSAAsyncSelect() 함수는 윈도우 핸들을 요구합니다. 따라서 서버, 클라이언트 모두 윈도우를 생성하고 이 함수를 호출합니다.

```
#define WM_SOCKET_NOTIFY    (WM_USER+1000)
```

네트워크 이벤트 'WM_SOCKET_NOTIFY' 사용자 이벤트(WM_USER)에 적당한 값을 더해서 설정합니다. 다음은 서버의 주-스레드 함수에서 네트워크에 관련된 내용만 정리한 코드입니다.

```
// 서버
…
struct RemoteHost
{
        SOCKET          scH;    // 소켓
        SOCKADDR_IN     sdH;    // 소켓어드레스
        …
};

// 사용자가 정의한 네트워크 메시지
#define WM_SOCKET_NOTIFY    (WM_USER+1000)

LRESULT WINAPI WndProc(HWND, UINT, WPARAM, LPARAM);     // 메시지 콜백 함수
LRESULT        NetProc(HWND, UINT, WPARAM, LPARAM);     // 네트워크 메시지 처리 함수

int main()
// 윈도우생성
…
WNDCLASS wc = {0};
wc.lpfnWndProc  = WndProc;
…
RegisterClass( &wc );

HWND hWnd = CreateWindow(…);
```

```
…

// 윈도우를 다 만들고 나서 네트워크 코드를 추가
…
hr = bind(g_scLstn, …);

// AsycSelect에 연결. 이벤트 : Accept -> FD_ACCEPT, Close -> FD_CLOSE
hr = WSAAsyncSelect(g_scLstn, hWnd, WM_SOCKET_NOTIFY, FD_ACCEPT|FD_CLOSE);

// Listen 상태
hr = listen(…);
…
// 윈도우 메시지 처리
MSG msg={0};
while( WM_QUIT != msg.message )
{
        …
}
…
```

<nw32_async.zip>

메시지 콜백 함수에서 직접 이벤트를 처리하는 것 보다 네트워크 메시지 처리 함수를 간접으로 호출하는 것이 유지 보수에 유리합니다.

```
// 메시지 콜백 함수
LRESULT WINAPI WndProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
…
switch( msg )
{
…
        // 네트워크메시지처리
        case WM_SOCKET_NOTIFY:
                NetProc(hWnd, msg, wParam, lParam);
                return 0;
…
```

<nw32_async.zip>

앞에서 네트워크 메시지 처리 함수를 간략하게 보였는데 좀더 자세한 코드를 만들어 봅시다. 먼저 WSAGETSELECTERROR 와 WSAGETSELECTEVENT 매크로를 사용해서 에러 코드와 이벤트를 얻습니다.
만약 에러가 있다면 Listen 소켓이면 SendMessage() 와 같은 함수를 호출해서 프로그램을 종료하고 Listen 소켓이 아니면 소켓과 일치하는 접속한 클라이언트를 해제 합니다.

```
// 네트워크 메시지 처리 함수
LRESULT NetProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
...
SOCKET  scHost  = (SOCKET)wParam;
DWORD   dError  = WSAGETSELECTERROR(lParam);
DWORD   dEvent  = WSAGETSELECTEVENT(lParam);
RemoteHost* pCln = NULL;

// 에러 체크
if(dError)
{
        if(scHost == g_scLstn)          // Listen소켓이면 프로그램 종료
                ...
                SendMessage(hWnd, WM_DESTROY, 0,0);
        else
                ...
                DeleteHost(scHost);     // 클라이언트 인스턴스 제거

        return 0;
}
```

<nw32_async.zip>

정상적인 네트워크 이벤트라면 서버는 긴급 메시지를 제외하고 FD_ACCEPT, FD_CLOSE, FD_READ, FD_WRITE 4가지 정도만 처리하면 됩니다.
이 중에서 FD_WRITE는 연결이 성립되고 connect() 또는 accept() 함수를 호출한 후거나 아니면 send() 함수가 호출된 후에 WSAEWOULDBLOCK으로 실패 메시지가 통지될 때도 나타납니다. 사실 자체적으로 해결이 가능한 부분이기 때문에 특별히 처리할 내용은 없습니다

클라이언트의 접속이 성립되면 FD_ACCEPT로 이벤트가 통지 됩니다. 새로운 클라이언트의 입출력도 동일하게 메시지로 처리하려면 accept() 함수로 얻은 클라이언트 소켓과 윈도우 핸들, 사용자 정의 메시지, 그리고 통지 이벤트를 FD_READ|FD_WRITE|FD_CLOSE 와 같이 비트 연산으로 정해서 WSAAsyncSelect() 함수에 전달합니다.

또한 클라이언트 리스트가 있다면 새로운 클라이언트를 리스트에 추가합니다.

```
// Accept 메시지
if(FD_ACCEPT == dEvent)
{
        // accept() 함수로 접속한 클라이언트 소켓을 얻는다.
        SOCKET  scCln = accept(scLstn, …);

        // 접속한 클라이언트도 메시지로 처리할 수 있도록 AsyncSelect로 설정
        WSAAsyncSelect(scCln, hWnd, WM_SOCKET_NOTIFY, FD_READ|FD_WRITE|FD_CLOSE);

        // 클라이언트 list에 추가
        …
        return 0;
}
```

<nw32_async.zip>

FD_CLOSE는 클라이언트의 접속이 종료된 것이므로 소켓을 해제하고 클라이언트를 리스트에서 제거합니다.

```
// 배열에 있는 클라이언트 주소를 가져옴
pCln = FindHost(scHost);
…
// client를 list에서 제거
if(FD_CLOSE == dEvent)
{
        …
        pCln->Close();
}
```

<nw32_async.zip>

FD_READ는 수신 버퍼에 쌓여 있는 데이터를 꺼내 달라는 메시지 입니다. recv() 함수를 사용해서 수신 버퍼의 내용을 꺼내옵니다. recv() 함수의 결과 값에 따라 처리하는 방법은 비동기 소켓에서 recv() 함수를 사용하는 것과 동일합니다. 만약 SOCKET_ERROR이면 WSAGetLastError() 함수를 호출해서 정확한 에러를 진단하고 WSAEWOULDBLOCK이 아니면 소켓 에러이므로 클라이언트를 종료합니다.
반환 값이 0이면 클라이언트가 종료를 요청한 gracefully close 이므로 클라이언트의 접속을 종료

합니다.
나머지는 0보다 큰 경우이므로 데이터의 전송이 존재하는 의미이고 이것을 출력합니다.

```
// 수신메시지
else if(FD_READ == dEvent)
{
        INT iRcv = 0;   char sBufRcv[MAX_BUF+4] = {0};
        iRcv=recv(scHost, sBufRcv, MAX_BUF, 0); // 수신 버퍼에서 데이터 꺼내오기

        if(SOCKET_ERROR == iRcv)
        {
                hr = WSAGetLastError();
                if(WSAEWOULDBLOCK != hr)
                        pCln->Close();          // 소켓 실패. client를 list에서 제거
                ...
        }
        else if(0 == iRcv)                      // 소켓 종료
                ...
                pCln->Close();
        else                                    // 수신 데이터 출력
                ...
...
```

<nw32_async.zip>

WSAAsyncSelect() 함수는 앞에서도 언급했듯이 소켓을 비동기 소켓 모드로 전환시킵니다. 따라서 send() 함수로 데이터를 전송했을 때 SOCKET_ERROR가 발생하면 WSAGetLastError() 함수로 에러 코드를 얻고 WSAEWOULDBLOCK에는 다시 데이터를 전송합니다.

```
void EchoMsg(char* s, int iLen)
...
        iSnd= 0;
        iTot= 0;
        while(iTot<iLen)
        {
                char* p = s + iTot;
                iSnd = send(pClnT->scH, p, iLen - iTot, 0);
                if(SOCKET_ERROR == iSnd)
```

```
                {
                        iSnd = WSAGetLastError();
                        if(WSAEWOULDBLOCK == iSnd)
                                continue;

                        pClnT->Close();         // 전송 error. client제거
                        break;
                }
                iTot += iSnd;                   // 보낸 데이터 크기 누적
        }
…
```

<nw32_async.zip>

클라이언트는 connect() 실행 전에 WSAAsyncSelect() 를 호출해야 접속을 통지 받을 수 있습니다. 따라서 이벤트 비트 조합은 FD_CONNECT|FD_WRITE|FD_READ|FD_CLOSE 이 되어야 합니다.
주의할 것은 자동으로 비동기 소켓 모드로 전환 되므로 connect()는 대부분 SOCKET_ERROR를 반환할 것이며 WSAGetLastError() 함수로 WSAEWOULDBLOCK이면 접속이 진행 중이므로 프로그램을 진행하고 나머지는 에러이므로 프로세스를 빠져나갑니다.

```
// Client
#define WM_SOCKET_NOTIFY    (WM_USER + 1000)
…
int main()
…
HWND hWnd = CreateWindow(…);

// 네트워크 시작
…
g_scHost = socket(…);
…
// connect() 실행 전 AsycSelect 에 연결.
hr = WSAAsyncSelect(g_scHost, hWnd, WM_SOCKET_NOTIFY
                        , FD_CONNECT|FD_WRITE|FD_READ|FD_CLOSE);
…
hr = connect(g_scHost, (SOCKADDR*)&g_sdHost, sizeof(SOCKADDR_IN));
if(SOCKET_ERROR == hr)
{
```

```
        Sleep(10);
        hr = WSAGetLastError();
        if(WSAEWOULDBLOCK !=hr)
                return -1;
}
...
```

<nw32_async.zip>

클라이언트의 네트워크 메시지처리 서버와 거의 동일합니다. 단지 접속이 실패할 수도 있으므로 FD_CONNECT나 에러를 체크하는 부분에서 접속 에러를 처리합니다.

FD_READ 메시지는 서버와 동일하게 recv() 함수로 데이터를 꺼내오는데 크기가 SOCKET_ERROR이면 WSAGetLastError() 함수로 에러의 원인을 찾고 WSAEWOULDBLOCK이 아니면 종료합니다. 0 이면 서버가 접속을 종료한 것이므로 소켓을 종료합니다.

```
LRESULT NetProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
...
SOCKET  scHost  = (SOCKET)wParam;
DWORD   dError  = WSAGETSELECTERROR(lParam);
DWORD   dEvent  = WSAGETSELECTEVENT(lParam);

// 에러 체크
if(dError)
{
        if(FD_CONNECT == dEvent) // 접속 실패
                SetWindowText(hWnd, "Connection Failed.");
...
}

if(FD_CONNECT == dEvent)        // 접속 메시지
...
if(FD_READ == dEvent)           // 수신메시지
{
        char sBufRcv[MAX_BUF+4]={0};
        INT iRcv=recv(scHost, sBufRcv, MAX_BUF, 0);
        ...
}
else if(FD_CLOSE == dEvent)     // 접속해제메시지
        ...
```

<nw32_async.zip>

단순히 서버가 수신만 하면 AsyncSelect는 모든 것을 메시지 프로시저 함수에서 처리할 수 있습니다. 하지만 송신이 있으면 send() 함수를 호출해야 하는데 이것을 앞의 예제처럼 프로시저 함수에서 호출하게 되면 네트워크의 속도가 프로세서를 따라가지 못하므로 프로그램이 잠시 대기할 수도 있습니다. 또한 네트워크의 지연의 문제 렉(Lag)이 발생하면 즉시 처리 해야 하는 프로그램으로서는 문제가 될 수 있습니다.
따라서 별도의 송신용 스레드를 만들어서 보낼 데이터가 없으면 일시 중지로 만들고 데이터가 있으면 스레드를 재개해서 데이터를 보내도록 합니다.
스레드를 일시 중지, 재개는 이벤트 객체 등을 사용할 수도 있지만 SuspendThread()와 ResumeThread() 함수가 있으므로 이 함수들을 사용하도록 합니다.
또한 송신용 스레드는 네트워크가 진행되기 전에 일시 중지 상태로 생성합니다.

```
// 네트워크 시작전 송신용 스레드 생성. 일시 중지상태
g_hThSnd = (HANDLE)_beginthreadex(… WorkSnd, NULL, CREATE_SUSPENDED, NULL);
…
// 네트워크 시작
WSAStartup(…);
…
```

<nw33_async_th.zip>

따라서 별도의 송신용 스레드를 만들어서 보낼 데이터가 없으면 일시 중지로 만들고 데이터가 있으면 스레드를 재개해서 데이터를 보내도록 합니다.
스레드를 일시 중지, 재개는 이벤트 객체 등을 사용할 수도 있지만 SuspendThread() 와 ResumeThread() 함수가 있으므로 이 함수들을 사용하도록 합니다.
또한 송신용 스레드는 네트워크가 진행되기 전에 일시 중지 상태로 생성합니다.

```
void EchoMsg(char* s, int iLen)
…
EnterCriticalSection(&m_CS);    // Lock
// 송신 버퍼 데이터 쓰기
…
LeaveCriticalSection(&m_CS);    // Unlock
ResumeThread(g_hThSnd);         // 송신 스레드 재개
…
```

<nw33_async_th.zip>

데이터 전송이 모두 끝나면 스레드가 스스로 일시 중시로 전환 되도록 SuspendThread() 함수를 데이터 전송이 완료되면 호출합니다. 참고로 GetCurrentThread() 함수는 스레드 내부에서 의사 핸들(pseudo handle)을 얻는 함수 입니다. 이 함수에서 얻은 핸들을 SuspendThread() 함수에 전달하면 스레드 스스로 일시 중지 상태가 됩니다.

```
// 송신 스레드
DWORD WINAPI WorkSnd(void *pParam)
...
while(1)
{
        EnterCriticalSection(&m_CS);            // Lock
        // 데이터 송신
        ...
        LeaveCriticalSection(&m_CS);            // Unlock
        ...
        HANDLE hThread = GetCurrentThread();    // 스레드의 핸들을 가져옴
        SuspendThread(hThread);                 // 스레드를 일시 중지로 전환
}
...
```

<nw33_async_th.zip>

3.3 Simple GUI Chatting

AsyncSelect는 윈도우 핸들을 필요로 하므로 클라이언트가 메시지 기반으로 작성되어 있다면 AsyncSelect는 가장 좋은 I/O 모델이 될 수 있습니다. 수신 스레드를 생성해도 나머지는 메시지에서 처리가 되기 때문에 시스템의 자원을 적게 사용하고 프로그램이 쉬워서 클라이언트 입장에서는 다른 I/O 모델보다 좋다고 볼 수 있습니다.
이렇게 장점이 많은 AsyncSelect I/O 모델을 그냥 지나칠 수 없다고 생각합니다. 따라서 우리는 GUI 환경의 채팅 프로그램을 만들어 봄으로써 AsyncSelect 를 충분히 익숙해지도록 합니다. 여기에 한발 더 나아가 함수 위주의 프로그램이 아닌 클래스 기반의 채팅 프로그램을 만들어보도록 합니다.

CAsyncSvr는 함수 중심의 AsyncSelect를 클래스화 한 것입니다. 기본적으로 윈도우 핸들, 통지 메시지, 클라이언트 리스트를 포함합니다.

```
class CAsyncSvr
…
        struct RemoteHost
        {
                SOCKET  scH;                    // 소켓
                char    sBuf[MAX_BUF_SND];      // 송신 버퍼
                …
        };

protected:
        HWND                    m_hWnd  ;       // 윈도우 핸들
        UINT                    m_wmNotify;     // 통지 메시지
        vector<RemoteHost*>     m_rmCln;        // 클라이언트 리스트
…
public:
        INT     Create(char* sIp, char* sPt, HWND hWnd, UINT wm);
        void    SendBuf();
        LRESULT NetProc(WPARAM, LPARAM);        // 네트워크 메시지 처리 함수
        static DWORD WINAPI WorkSnd(void *);    // Send용 스레드
…
```

CAsyncSvr클래스의 Create 함수는 외부에서 아이피, 포트, 윈도우 핸들 그리고 네트워크 통지 메시지를 받아서 리슨 소켓을 생성하고, WSAAsyncSelect() 함수를 호출합니다. 이외에 동기화 객체 임계 구역과 송신 스레드도 같이 생성합니다.

```
INT CAsyncSvr::Create(char* sIp, char* sPt, HWND hWnd, UINT wm)
…
m_hWnd          = hWnd;
m_wmNotify      = wm;
…
InitializeCriticalSection(&m_CS);
…
bind(m_scLstn, (SOCKADDR*)&m_sdLstn, sizeof(SOCKADDR_IN));
…
//데이터송신용Thread 생성. 대기상태
m_hThSnd = (HANDLE)_beginthreadex(… CAsyncSvr::WorkSnd, this, CREATE_SUSPENDED, NULL);
…
```

```
// AsycSelect에연결. 소켓은자동으로Non-blocking으로전환
hr = WSAAsyncSelect(m_scLstn, m_hWnd, m_wmNotify, FD_ACCEPT|FD_CLOSE);
…
hr = listen(m_scLstn, SOMAXCONN);
…
```

CAsyncSvr클래스의 NetProc 함수는 네트워크 메시지를 처리하는 함수입니다. 이전에 함수로 구성된 것을 단순히 멤버 함수로 변경했기 때문에 내용이 같아 설명은 생략하겠습니다.

```
LRESULT CAsyncSvr::NetProc(WPARAM wParam, LPARAM lParam)
…
SOCKET scHost = (SOCKET)wParam;
DWORD   dError  = WSAGETSELECTERROR(lParam);
DWORD   dEvent  = WSAGETSELECTEVENT(lParam);
…
```

스레드는 전역 함수 또는 static 함수를 필요로 합니다. 스레드 함수 내에서 객체의 멤버들을 접근해서 처리하는 것보다 객체의 처리 함수를 호출하는 것이 유지 보수에 좀 더 낳은 방법입니다. 송신을 담당하는 WorkSnd 스레드의 내용을 클래스 멤버 함수로 옮기고, 이 멤버 함수를 호출함으로써 프로그램을 단순화 시킬 수 있습니다.

```
DWORD WINAPI CAsyncSvr::WorkSnd(void *pParam)
…
CAsyncSvr* pNet = (CAsyncSvr*)pParam;
while(1)
{
        pNet->SendBuf();                // CAsyncSvr 객체의 송신 처리 함수 호출
        HANDLE hThread = GetCurrentThread();
        SuspendThread(hThread);         // 스레드를 일시 중지 상태로 변경
}
…
```

스레드 내부에서 데이터를 처리하다 보면 클래스의 인스턴스 데이터는 전부 public으로 선언해야 접근이 가능합니다. 이것은 캡슐화가 전혀 안되어 있는 것과 마찬가지가 됩니다. 이럴 때 깊이 고민하지 말고 간단히 멤버 함수를 추가하고 이 멤버 함수를 호출하는 것으로 프로그램을 구성합니다.
CAsyncSvr 클래스의 SendBuf() 함수는 이전 함수 중심으로 구성된 프로그램의 스레드 내부에서

전송에 대한 내용을 클래스의 멤버 함수로 옮겨온 것으로 전송 스레드는 이 함수를 호출하는 간단한 구조가 됩니다.

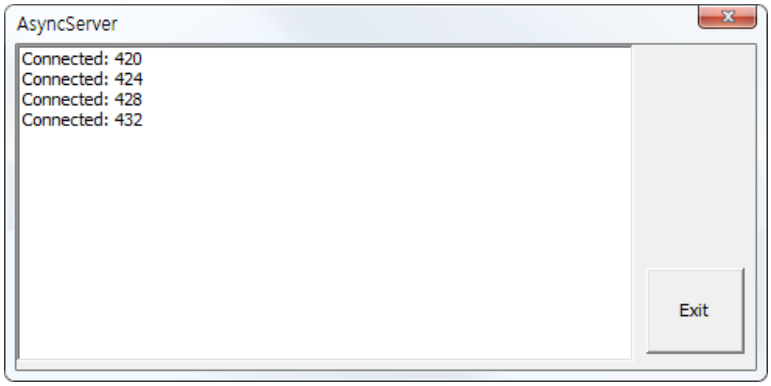
```
void CAsyncSvr::SendBuf()
…
EnterCriticalSection(&m_CS);
        …
        iSnd = send(pCln->scH, p, iLen-iTot, 0);
LeaveCriticalSection(&m_CS);
…
```

클래스를 다 구성했다면 남아 있는 것은 프로그램이 시작될 때 CAsyncSvr 인스턴스를 생성하고 인스턴스의 네트워크 메시지 처리하는 함수를 호출하는 것입니다.
윈도우 프로그램에서 DialogBox를 사용하면 리소스에서 만든 대화 상자를 간단히 출력할 수가 있습니다. WM_INITDIALOG 에서 CAsyncSvr 인스턴스를 생성하고 WM_SOCKET_NOTIFY에서 인스턴스의 메시지 처리 함수 NetProc()를 호출하면 더 이상 윈도우 메시지 프로시저 함수에서 할 일은 없습니다.

```
CAsyncSvr* g_pSvr = NULL;
// 네트워크 메시지
#define WM_SOCKET_NOTIFY    (WM_USER+1000)
…
LRESULT WINAPI WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
…
if(WM_INITDIALOG == uMsg)
{
        g_pSvr = new CAsyncSvr;
        …
}
else if(WM_SOCKET_NOTIFY == uMsg)
{
        if(FAILED(g_pSvr->NetProc(wParam, lParam)))
        {
        }
        …
}
```

프로그램의 진입에서 main(), 또는 WinMain() 에서는 네트워크 프로그램과 상관 없이 단순히 윈도우 프로그램을 호출하는 것으로 함수 내용을 작성합니다.

```
int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
...
DialogBox( (HINSTANCE)GetModuleHandle(NULL)
        , MAKEINTRESOURCE(IDD_SERVER)
        , NULL, (DLGPROC)WndProc);
...
```



<nw34_async_chatt.zip>

채팅 서버를 완성했습니다. 다음으로 채팅 클라이언트를 제작할 차례입니다. 서버를 만들 때와 동일하게 클래스로 작성합니다. 클라이언트는 융통성을 위해서 아이피, 포트를 받아 접속할 수 있도록 인터페이스를 구성합니다. 또한 접속을 끊고 다시 연결하는 것도 가능하도록 합니다.

```
class CAsyncCln
{
...
protected:
        HWND    m_hWnd  ;
        UINT    m_wmNotify;
        ...
public:
        ...
        INT     Create(HWND hWnd, UINT wm);     // Network Message Procedure
        INT     Connect(char* sIp, char* sPt);  // 접속 인터페이스
        INT     DisConnect();                   // 접속 해제
        void    SendBuf();                      // 전송 스레드가 사용할 전송 담당 함수
        LRESULT NetProc(WPARAM, LPARAM);        // 네트워크 메시지 처리 함수
```

```
…
};
```

CAsyncCln 클래스의 Create 함수는 외부에서 윈도우 핸들, 네트워크 통지 메시지를 받아서 네트워크와 임계 구역을 초기화하고 데이터 송신용 스레드를 일시 중지 상태로 생성합니다.

```
INT CAsyncCln::Create(HWND hWnd, UINT wm)
…
m_hWnd          = hWnd;
m_wmNotify      = wm;

InitializeCriticalSection(&m_CS);

hr = WSAStartup(…);

//데이터 송신용 Thread 생성. 일시 중지 상태
m_hThSnd = (HANDLE)_beginthreadex(… CAsyncCln::WorkSnd, this, CREATE_SUSPENDED, NULL);
…
```

Connect 함수는 아이피, 포트를 받아서 소켓을 생성하고 윈도우 핸들과 메시지 통지를 가지고 WSAAsyncSelect() 함수를 실행합니다. connect() 함수를 실행해서 서버와 접속을 시도합니다.

```
INT CAsyncCln::Connect(char* sIp, char* sPt)
…
DisConnect();
…
m_scHost = socket(…);
…
hr = WSAAsyncSelect(m_scHost, m_hWnd, m_wmNotify, FD_CONNECT|FD_READ|FD_WRITE|FD_CLOSE);
…
hr = connect(…);
if(SOCKET_ERROR == hr)
{
        hr = WSAGetLastError();
        if(WSAEWOULDBLOCK != hr)
                return 0;
}
```

…

CAsyncCln의 NetProc() 함수는 서버와 마찬가지로 네트워크 메시지를 처리합니다.

```
LRESULT CAsyncCln::NetProc(WPARAM wParam, LPARAM lParam)
…
SOCKET  scHost = (SOCKET)wParam;
DWORD   dError  = WSAGETSELECTERROR(lParam);
DWORD   dEvent  = WSAGETSELECTEVENT(lParam);
if(dError)
…
if(FD_WRITE == dEvent)
…
else if(FD_CONNECT == dEvent)
…
else if(FD_READ == dEvent)
…
```

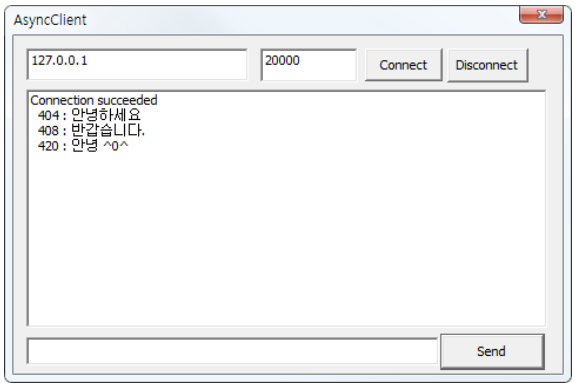
CAsyncCln의 SendBuf()함수는 전송 스레드가 사용하는 함수로 데이터 전송을 담당하는 함수입니다. 객체의 송신 버퍼에 있는 데이터가 전부 전송될 때까지 while 루프로 반복해서 전송합니다.

```
void CAsyncCln::SendBuf()
…
EnterCriticalSection(&m_CS);
…
while(…)
{
        iSnd = send(m_scHost, p, iLen-iTot, 0);
        if(SOCKET_ERROR == iSnd)
        {
                iSnd = WSAGetLastError();
                if(WSAEWOULDBLOCK == iSnd)
                        continue;
        }
        …
}
```

```
LeaveCriticalSection(&m_CS);
...
```

전송 스레드는 객체의 전송 멤버 함수 SendBuf()를 호출하고 스스로 일시 중지 상태로 돌아갑니다.

```
DWORD WINAPI CAsyncCln::WorkSnd(void *pParam)
...
CAsyncCln* pNet = (CAsyncCln*)pParam;

while(1)
        pNet->SendBuf();
        HANDLE hThread = GetCurrentThread();
        SuspendThread(hThread);
...
```



<nw34_async_chatt.zip>

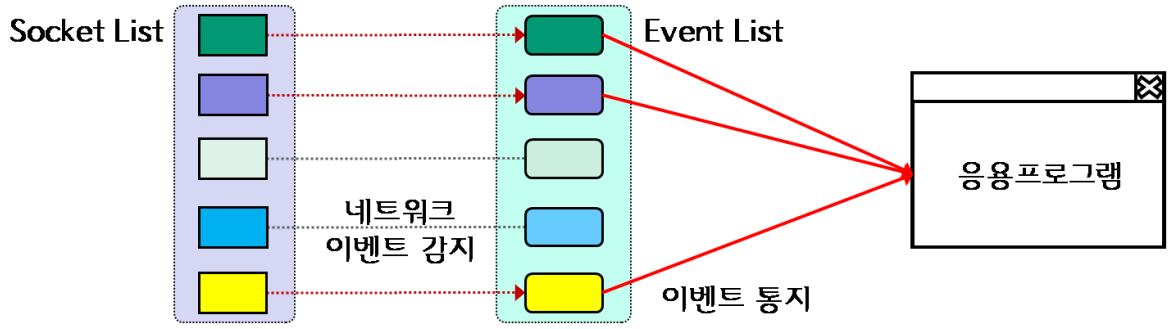
이렇게 채팅 서버와 채팅 클라이언트를 만들어보았습니다. 채팅용 프로그램은 앞으로 EventSelect, Overlapped I/O를 연습하는 파트너 프로그램으로도 유용하게 활용될 수 있으니 꼭 구현해 보기 바랍니다.

3.4 WSAEventSelect I/O

EventSelect I/O 모델은 AsyncSelect 모델을 좀 더 일반화 시킨 모델입니다. 따라서 구동 방식은 거의 같으며 유연성을 높이기 위해서 윈도우의 메시지 큐(Message Queue)를 사용하지 않고, 소켓에 이벤트 객체를 설정해서 네트워크의 입출력에 대한 이벤트를 응용 프로그램에게 직접 통지하는 모델입니다.

이 모델의 가장 큰 장점은 윈도우를 만들지 않고 네트워크 입출력을 구현할 수가 있습니다. 통지

를 받기 위한 별도의 스레드가 필요하지만 이 스레드는 통지 받기 전까지 프로세스가 대기(waiting) 상태가 되므로 시스템의 프로세서 자원을 점유하지 않아 큰 문제가 되지 않으며 이 모델을 테스트한 몇몇 보고서에 의하면 수 천명 정도의 클라이언트 접속에서도 어느 윈속 입출력 모델 보다 우수하다고 보고 되고 있습니다.



<WSAEvent Select I/O 모델>

구현이 간단하고 성능이 좋아서 네트워크 입출력이 빈번하지 않거나 서버의 하드웨어 성능이 낮고, 클라이언트 수가 수 백 명 정도 이하 되는 온라인 게임 서버에서도 간혹 이 모델을 사용한다고 합니다.

시스템은 폴링(polling) 방식으로 소켓의 입출력을 알기 위해서 주기적으로 확인합니다. 폴링 방식의 취약점은 프로세서가 네트워크 입출력이 없어도 계속 확인하는데 있습니다. 이러한 소켓에 이벤트 객체를 바인딩 하면 프로세서는 이벤트 객체가 네트워크 입출력을 감지하고 신호 상태가 되기까지 대기하고 신호 상태(signal) 상태만 입출력을 처리하게 됩니다.

```
WSAEVENT WSACreateEvent()
```

WSACreateEvent 함수는 수동 리셋(manual reset), 비신호(non-signal) 시작 상태인 이벤트 객체를 생성하는 함수로 CreateEvent(NULL, TRUE, FALSE, NULL)로 생성한 이벤트와 동일합니다. 자동 리셋 모드가 아닌 수동 리셋 모드를 사용하는 이유는 WSAWaitForMultipleEvents() 함수에서 설명하겠습니다.

```
int WSAEventSelect(SOCKET s, WSAEVENT hEvent, long lNetworkEvents)
```

소켓의 입출력을 감지하기 위해서 이벤트 객체와 바인딩 해주는 함수입니다. 함수의 호출이 성공하면 0을 반환하고 에러는 WSAGetLastError()로 확인 합니다.

첫 번째 인수는 소켓, 두 번째 인수는 이벤트 객체를, 세 번째 인수는 WSAAsyncSelect() 함수에서 사용한 이벤트의 비트 조합을 동일하게 사용합니다.

비트 조합에 사용될 수 있는 이벤트 상수는 FD_ACCEPT, FD_CONNECT, FD_CLOSE, FD_READ, FD_WRITE, FD_OOB 등이 있으며 일반적으로 서버의 리슨 소켓은 (FD_ACCEPT | FD_CLOSE), 서버에서 accept() 함수로 얻은 클라이언트 소켓은 (FD_READ | FD_WRITE | FD_CLOSE) 을 사용합니다. 클라이언트의 connection 소켓은 (FD_CONNECT | FD_READ | FD_WRITE | FD_CLOSE) 로 비트 조합을 설정합니다.

이 함수를 호출하면 WSAAsyncSelect() 함수와 마찬가지로 소켓이 비동기 소켓으로 전환합니다. 따라서 입출력에서 SOCKET_ERROR가 자주 발생되고 WSAAsyncSelect() 함수로 확인한 에러 코드가 WSAEWOULDBLOCK이면 입출력이 진행하고 있는 것으로 처리해야 합니다.

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

WSAWaitForMultipleEvents(이하 WFME) 함수는 입력된 이벤트 객체 리스트에서 한 개 또는 모두가 신호 상태가 되거나, 지정된 시간만큼(time-out) 기다리는 대기 함수(waiting function) 입니다.
첫 번째 인수는 신호 상태를 감지할 이벤트 리스트 안에 있는 이벤트 개수입니다. 최대 개수는 WSA_MAXIMUM_WAIT_EVENTS 로 한정되어 있습니다. 즉, 최대 WSA_MAXIMUM_WAIT_EVENTS 만큼만 이벤트의 신호 상태를 감지할 수 있다고 볼 수 있습니다. 두 번째 인수는 이벤트 리스트입니다.

세 번째 인수는 TRUE로 설정하면 이벤트 배열의 모든 이벤트가 신호 상태가 되어야 함수가 반환되며 FALSE로 설정하면 이벤트 객체 중에 하나라도 신호 상태가 되면 함수는 반환합니다.
FALSE로 설정했을 때 주의해야 할 것은 두 개 이상의 이벤트 객체가 신호 상태가 되면 가장 낮은 이벤트의 배열 인덱스 + WSA_WAIT_EVENT_0 값을 반환합니다. 따라서 다른 신호 상태 이벤트를 찾기 위해서 WFME 함수를 한 번 더 사용해 합니다.

네 번째 인수는 대기 시간으로 1/1000초 단위를 사용합니다. WSA_INFINITE를 사용하면 신호 상태가 될 때까지 이 함수를 호출한 프로세스를 대기 상태로 만듭니다.
다섯 번째 인수는 completion routine의 처리에 대한 설정 값으로 EventSelect 환경에서는 FALSE로 설정합니다.

함수 실행의 성공에 대해서 3가지로 분류 할 수 있습니다. 먼저 이벤트 객체의 신호 상태를 감지했다면 [WSA_WAIT_EVENT_0, WSA_WAIT_EVENT_0+cEvents-1] => WSA_WAIT_EVENT_0 + [0, cEvents) 범위의 값을 반환합니다. 다음으로 만약 fAlertable을 TRUE로 설정했다면 완료 루틴에 대한 결과 WAIT_IO_COMPLETION도 반환 받을 수 있습니다. 마지막으로 대기 시간을 무한대(WSA_INFINITE)로 설정하지 않았다면 지정된 대기 시간 경과에 대해서 WSA_WAIT_TIMEOUT을 반환 받을 수 있습니다.

우리는 대기 시간을 INFINITE, fAlertable을 FALSE로 해서 사용할 것이므로 WSA_WAIT_EVENT_0 +

[0, cEvents) 범위 값이 아니라면 에러로 처리해도 됩니다.

함수 실행이 실패 하면 WSA_WAIT_FAILED를 반환하며 WSAGetLastError 함수를 호출하여 특정한 에러 코드를 얻어낼 수 있습니다.
참고로 WFME 함수는 WIN API의 시간 대기 함수 WaitForMultipleObjectsEx() 함수와 동일하며, EventSelect를 사용하는 동안 우리는 마지막 Alertable을 FALSE로 사용할 것이므로 WFME 대신 WaitForMultipleObjects() 함수를 사용해도 됩니다.

```
int WSAEnumNetworkEvents(SOCKET s, WSAEVENT hEvent, LPWSANETWORKEVENTS lpNetworkEvents)
```

WSAEnumNetworkEvent() 함수는 소켓에서 발생한 네트워크 이벤트의 종류를 알아내는 함수 입니다. 또한 이벤트 객체에 저장된 이벤트 내용을 초기화하고, WSACreateEvent() 함수로 생성한 수동 리셋 모드 이벤트 객체를 비신호 상태로 변경합니다.
세 번째 인수는 WSANETWORKEVENTS 구조체 변수의 주소입니다. 이 구조체는 이벤트 종류와 이벤트에 대한 에러 코드로 구성되어 있습니다. 따라서 이 함수의 반환과 별개로 에러를 처리해야 합니다.

```
typedef struct _WSANETWORKEVENTS {
    long lNetworkEvents;
    int iErrorCode[FD_MAX_EVENTS];
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS;
```

WSANETWORKEVENTS 구조체의 lNetworkEvents 필드는 네트워크 이벤트 종류를 저장합니다. 이벤트의 종류는 AsyncSelect와 동일하게 FD_READ ~ FD_ADDRESS_LIST_CHANGE으로 지정됩니다.

구조체의 두 번째 필드 iErrorCodes는 lNetworkEvents의 이벤트 종류에 대해서 발생한 이벤트 에러 코드가 저장 됩니다. 예를 들어 recv에서 에러가 발생했다면 lNetworkEvents가 FD_READ일 때 iErrorCode[FD_READ_BIT]에 에러 값이 설정됩니다.
참고로 WinSock2.h 파일에는 에러 코드의 배열 인덱스가 지정되어 있습니다.

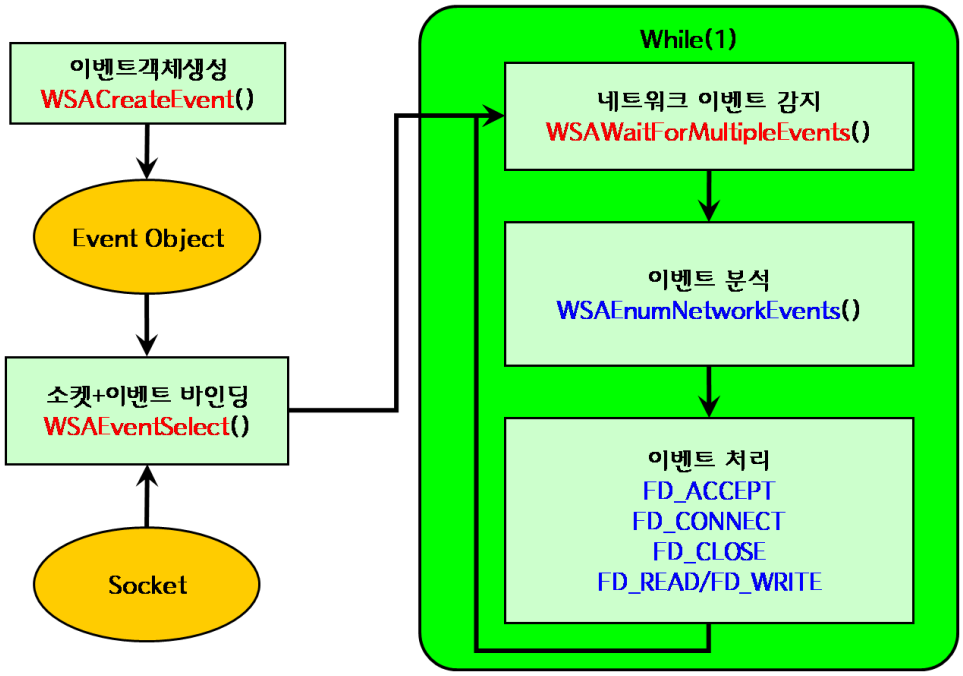
```
#define FD_READ_BIT   0          #define FD_ACCEPT_BIT   3
#define FD_WRITE_BIT  1          #define FD_CONNECT_BIT  4
#define FD_OOB_BIT    2          #define FD_CLOSE_BIT    5
```

지금까지 WSAEventSelect I/O에 대한 함수들을 살펴보았습니다. 본격적으로 프로그램을 작성하기 전에 WSAEventSelect 를 사용하는 프로그램 구조를 파악할 필요가 있습니다.
WSAEventSelect의 프로그램 구조는 WSAAsyncSelect와 상당히 유사합니다. 간단히 <소켓, 이벤트,

이벤트 통지 종류>를 하나의 그룹으로 생각한다면 소켓이 생성이 되면 이벤트 객체를 생성하고, 이벤트 통지를 WSAEventSelect() 함수로 묶어 주는 것입니다.
좀 더 구체적으로 본다면 서버의 리슨 소켓은 bind() 실행 후에 WSACreateEvent() 함수로 이벤트 객체를 생성하고 통지 이벤트를 (FD_ACCEPT|FD_CLOSE)와 같이 비트 조합을 한 다음 "WSAEventSelect (리슨 소켓, 이벤트 객체, FD_ACCEPT | FD_CLOSE)" 와 같이 바인딩 합니다.

서버에 접속한 클라이언트도 이벤트 통지를 받기 위해서 리슨 소켓과 동일한 과정을 거칩니다. accept() 함수로 소켓을 얻고, WSACreateEvent()로 이벤트 객체를 생성하고 통지 이벤트를 (FD_READ|FD_WRITE|FD_CLOSE)와 같이 비트 조합을 한 다음 "WSAEventSelect (클라이언트 소켓, 이벤트 객체, FD_READ|FD_WRITE|FD_CLOSE)" 와 같이 바인딩 합니다.

클라이언트 프로그램은 호스트 소켓 한 개만 있으므로 이벤트 객체에 입출력에 관련된 모든 이벤트를 "WSAEventSelect (호스트 소켓, 이벤트 객체, FD_CONNECT | FD_READ | FD_WRITE | FD_CLOSE)" 와 같이 설정합니다.



<WSAEventSelect() 함수를 사용한 네트워크 프로그램 구조>

본격적으로 WSAEventSelect() 함수를 사용해서 서버를 제작해 봅시다. WFME 함수가 최대 64개까지 이벤트를 감시할 수밖에 없으므로 클라이언트의 최대 숫자도 미리 정해서 사용합니다.

```
#define MAX_HOST        WSA_MAXIMUM_WAIT_EVENTS
```

본격적으로 WSAEventSelect() 함수를 사용해서 서버를 제작해 봅시다. WFME 함수가 최대 64개까지

이벤트를 감시할 수밖에 없으므로 클라이언트의 최대 숫자도 미리 정해서 사용합니다.
클라이언트 소켓 뿐만 아니라 리슨 소켓도 이벤트 객체로 통지 받아야 하므로 소켓과 이벤트가 포함된 같은 구조체를 사용하는 것이 관리하는데 유리합니다.

```
struct RemoteHost
{
        SOCKET          scH;    // 소켓
        WSAEVENT        seH;    // 이벤트 객체
        …
        void Set(SOCKET s, SOCKADDR_IN* d, WSAEVENT v)
                …
                scH= s; seE = v;        // 소켓, 이벤트 설정
                …
        void Close()
                …
                closesocket(scH);       // 소켓 해제
                CloseHandle(seH);       // 이벤트 객체 해제
                …
};

…
RemoteHost      g_rmHost[MAX_HOST];     // Host list: listen + client list
…
```

주-스레드에서 WSAAsyncSelect와 마찬가지로 listen 상태 전에 이벤트 객체를 생성하고 WSAEventSelect()에 리슨 소켓, 이벤트 객체, 이벤트 비트 조합을 인수로 넣고 실행합니다.

```
int main()
…
g_scLstn = socket(…);           // 리슨 소켓 생성
…
hr = bind(g_scLstn, …);         // 바인딩
…
g_seLstn = WSACreateEvent();    // 이벤트 객체 생성

// <소켓, 이벤트 객체, 이벤트 통지 종류(accept, close)> 바인딩.
// 소켓은 자동으로 비동기 소켓으로 전환됨
```

```
hr = WSAEventSelect(g_scLstn, g_seLstn, FD_ACCEPT|FD_CLOSE);

// Listen 소켓과 이벤트 등을 호스트 리스트 0번에 추가.
g_rmHost[0].Set(g_scLstn, &g_sdLstn, g_seLstn);
hr = listen(…);
…
```

WSAWaitForMultipleEvents() 함수 호출 전에 호스트 리스트에 저장된 이벤트 객체와 소켓을 가지고 이벤트 객체 리스트, 소켓 리스트를 구성합니다.
WFME 함수는 성공하면 WSA_WAIT_EVENT_0 + [0, cEvents) 범위 값을 반환하므로 WSA_WAIT_EVENT_0 만큼 감소시키면 배열의 인덱스와 일치하게 됩니다.

```
// main() 계속
while(1)
{
        …
        WSAEVENT        vEvn[MAX_HOST]={0};     // WFME 함수에 사용할 이벤트 객체 리스트
        SOCKET          vSck[MAX_HOST]={0};     // WFME 함수에 사용할 소켓 리스트
        nLst = 0;
        for(i=0; i<MAX_HOST; ++i)
        {
                if(0 == g_rmHost[i].scH)        // 유효한 소켓이 아니면
                        continue;

                vEvn[nLst] = g_rmHost[i].seH;
                vSck[nLst] = g_rmHost[i].scH;
                ++nLst;
        }

        // Accept 포함한 네트워크 이벤트를 기다린다.
        nE = WSAWaitForMultipleEvents(nLst, vEvn, FALSE, WSA_INFINITE, FALSE);
        if(nE == WSA_WAIT_FAILED)
                break;

        // receive event
        nE -= WSA_WAIT_EVENT_0; // 인덱스 재 조정
```

WFME 함수는 두 개 이상의 이벤트가 신호 상태가 되면 가장 낮은 인덱스를 반환하므로 시작 인덱스부터 전체 배열의 숫자까지 for 루프를 사용 나머지 신호 상태 이벤트 객체들을 찾아 냅니다.

신호 상태 이벤트 객체를 찾기 위해 WFME 함수를 한 번 더 사용합니다. 하나의 이벤트 객체만 검사하므로 cEvents = 1로 설정합니다. fWaitAll 값은 관습적으로 TRUE로 설정합니다.(사실 1개만 검사하기 때문에 FALSE로 설정해도 문제가 없습니다.) 또한 단지 신호 상태 여부만 파악하는 것이므로 대기시간 0로 설정합니다.

WFME 함수는 검사할 이벤트 객체가 신호 상태가 아니면 WSA_WAIT_TIMEOUT 을 반환 할 것이므로 반환이 WSA_WAIT_FAILED도 포함해서 continue로 이벤트 처리를 건너 뜁니다.

참고로 이렇게 WFME 함수를 2번 사용하기 때문에 이벤트 객체를 자동 리셋으로 생성하면 첫 번째 WFME 함수 호출에서 비신호 상태가 되어 2 번째에서는 검사가 안됩니다.

WSAEnumNetworkEvents() 함수로 이벤트의 종류, 에러 코드를 얻어 냅니다. 만약 이벤트 종류가 FD_ACCEPT라면 accept() 함수를 호출해서 접속한 클라이언트의 소켓을 알아내고 이벤트 객체를 WSACreateEvent()함수로 생성한 다음, 비동기로 통지 받을 이벤트 비트 조합을 (FD_READ | FD_WRITE | FD_CLOSE) 로 설정해서 WSAEventSelect로 바인딩 합니다. 이렇게 하면 접속한 클라이언트 소켓의 입출력을 비동기로 통지 받을 수 있습니다.

```
// main():: while(1) 계속
        for(i= nE; i<nLst; ++i)
        {
                hr = WSAWaitForMultipleEvents(1, &vEvn[i], TRUE, 0, FALSE);
                if( WSA_WAIT_FAILED  == hr || WSA_WAIT_TIMEOUT == hr)
                        continue;

                // 이벤트 내용, 에러 코드 해석
                hr = WSAEnumNetworkEvents(vSck[i], vEvn[i], &wnE);
                // WSAEnumNetworkEvents() 함수의 에러 반환 체크
                if(SOCKET_ERROR == hr)
                {
                        hr= WSAGetLastError();
                        …
                        if(scHost == g_scLstn)  // listen socket
                                goto END;
```

```
            …
            pHost->Close();
            continue;
        }
        …
        if(FD_ACCEPT & wnE.lNetworkEvents)      // Accept event
        {
            …
            scCln = accept(scLstn, …);      // 클라이언트 소켓, 주소얻기
            seCln = WSACreateEvent();       // 이벤트 객체 생성

            // 접속한 클라이언트도 비동기로 통지 받을 수 있도록
            // <소켓, 이벤트 객체, 비동기 통지 이벤트 종류> 바인딩.
            WSAEventSelect(scCln, seCln, FD_READ|FD_WRITE|FD_CLOSE);

            // 비어있는 호스트를 찾고 추가
            …
            continue;
        }
        …
        else if(FD_CLOSE & wnE.lNetworkEvents)  // 클라이언트 소켓 종료 이벤트
            …
        else if(FD_READ & wnE.lNetworkEvents)   // 수신 이벤트
            …
            iRcv=recv(vSck[i], bufRcv, MAX_BUF, 0);
        …
```

<nw41_ev_sel.zip>

WSAEventSelect를 사용하는 클라이언트도 서버와 거의 동일 합니다. 소켓을 생성하고 이벤트를 생성하고, 소켓, 이벤트, 비동기로 통지 받을 이벤트 비트 조합을 FD_CONNECT | FD_READ | FD_WRITE | FD_CLOSE 로 설정해서 WSAEventSelect 함수로 바인딩 합니다.

바인딩 하고 나서 connect() 함수를 실행하고 WFME 함수로 이벤트 객체가 신호 상태가 될 때까지 무한히 기다리다가 접속(FD_CONNECT), 송신(FD_WRITE), 수신(FD_READ), 접속 종료(FD_CLOSE) 등에 대한 이벤트를 처리하면 됩니다.

AsyncSelect 와 마찬가지로 항상 주의할 것은 소켓의 입출력 결과가 SOCKET_ERROR 로 반환 된다면

반드시 WSAGetLastError() 함수를 호출해서 에러 코드에 대응해야 한다는 것입니다. 특히 WSAEWOULDBLOCK은 적절한 처리가 필요합니다. 또한 모든 이벤트에 대해서 iErrorCode[] 를 확인하는 것이 중요합니다.

다음은 서버에서 보낸 데이터를 수신한 후에 테스트 문자열을 보내는 클라이언트 예제입니다.

```
SOCKET   g_scHost = 0;          // socket
WSAEVENT g_seHost = 0;          // Event
…
int main()
…
g_scHost = socket(…);           // 소켓 생성
…
g_seHost = WSACreateEvent();    // 이벤트 객체 생성

// <소켓, 이벤트 객체, 비동기 통지 이벤트 종류> 바인딩
hr = WSAEventSelect(g_scHost, g_seHost, FD_CONNECT|FD_READ|FD_WRITE|FD_CLOSE);
hr = connect(g_scHost, …);
…
while(…)
        …
        hr = WSAWaitForMultipleEvents(1, &g_seHost, …);         // 이벤트 기다림
        …
        hr = WSAEnumNetworkEvents(g_scHost, g_seHost, &wnE);    // event 분해
        if(SOCKET_ERROR == hr)
                …
        if(FD_CONNECT & wnE.lNetworkEvents)             // 접속 완료
                if(wnE.iErrorCode[FD_CONNECT_BIT])      // 접속 에러 체크
                        …
        else if(FD_WRITE & wnE.lNetworkEvents)          // 송신
                …
        else if(FD_READ & wnE.lNetworkEvents)           // 수신
        {
                if(wnE.iErrorCode[FD_READ_BIT])         // 수신 에러 체크
                        …
                iRcv = recv(g_scHost, …);
                …
```

```
        }
        else if( FD_CLOSE & wnE.lNetworkEvents)         // 접속 종료
                ...
```

<nw41_ev_sel.zip>

AsyncSelect에서 송신을 담당하는 별도의 스레드를 추가했었는데 같은 이유로 EventSelect 를 사용할 때도 마찬가지 입니다. 또한 현재 주-스레드(main 함수)가 완료 통지를 처리하고 있지만 이것도 마찬가지로 별도의 스레드, 수신 스레드에서 처리하도록 하고 주-스레드는 네트워크에서 뭔가 더 처리할 일을 할 수 있도록 비워 둡니다.

먼저 주-스레드에 있는 비동기 통지를 수신 스레드를 생성하고 코드를 옮기고, 덧붙여 리슨 소켓을 포함하는 호스트 리스트를 STL vector 컨테이너로 관리하도록 합니다.

```
#include <vector>
using namespace std;
...
struct RemoteHost
{
        SOCKET          scH;            // 소켓
        WSAEVENT        seH;            // 이벤트
        int             nBuf;           // 송신할 버퍼의 크기
        char            sBuf[MAX_BUF];  // 송신 버퍼
        ...
};
...
vector<RemoteHost*>     g_vHost;        // Host list
...
```

주-스레드에 작성되어 있던 데이터 수신과 비동기 통지를 수신용 스레드로 옮겨 놓고, 수신용 스레드를 생성하고 자신의 컨텐츠를 실행합니다.

```
// 서버: 주-스레드
int main()
...
hr = bind(…);
...
seLstn = WSACreateEvent();      // 이벤트 객체 생성
```

```
// <소켓, 이벤트, 이벤트 통지 종류> 바인딩
hr = WSAEventSelect(scLstn, seLstn, FD_ACCEPT|FD_CLOSE);

// 호스트 리스트에 추가
g_vHost.push_back(new RemoteHost(scLstn, &sdLstn, seLstn));
…
hr = listen(…);
…
//수신 및 이벤트 통지용 Thread 생성
g_hThRcv = (HANDLE)_beginthreadex(… WorkRcv, …);
…
while(1)
        // 서버 컨텐츠 실행
…
```

데이터 수신 및 비동기 통지 스레드는 사용하지 않는 클라이언트를 호스트 리스트에서 제거하고 WFME 함수에서 사용할 이벤트 리스트와 소켓 리스트를 다시 작성합니다.
WFME에서 신호 상태의 이벤트를 기다리고, WFME 함수로 신호 상태의 이벤트 객체를 찾고, WSAEnumNetworkEvents() 함수로 이벤트의 종류를 얻은 다음, FD_ACCEPT, FD_READ, FD_WRITE, FD_CLOSE에 대응하는 이벤트를 처리합니다.

```
DWORD WINAPI WorkRcv(void* pParam)
…
while(1)
…
        // clear the host list (not used)
        DeleteNotUseHost();

        // listen 소켓 이벤트 포함, 클라이언트의 모든 이벤트를 리스트에 설정
        for(i=0; i<(int)g_vHost.size(); ++i)
        {
                RemoteHost* p =  g_vHost[i];
                if(0 == p->scH)
                        continue;

                vEvn[nLst] = g_vHost[i]->seH;
```

```
            ++nLst;
        }

        // Accept를 포함한 네트워크 이벤트를 기다린다.
        nE = WSAWaitForMultipleEvents(nLst, vEvn, FALSE, WSA_INFINITE, FALSE);
        …
        nE -= WSA_WAIT_EVENT_0; // 인덱스 재조정
        for(i= nE; i<nLst; ++i)
        {
            // 신호 상태 이벤트 객체 찾기
            hr = WSAWaitForMultipleEvents(1, &vEvn[i], TRUE, 0, FALSE);
            …
            // 이벤트 객체로 호스트 객체 찾기
            pHost = FindHost(scHost);
            …
            SOCKET scHost = pHost->scH;
            …
            // 이벤트 분해
            hr = WSAEnumNetworkEvents(vSck[i], vEvn[i], &wnE);
            …
            if(FD_ACCEPT & wnE.lNetworkEvents)      // Accept event
            {
                // accept()함수로 클라이언트소켓, 주소얻기
                scCln = accept(scLstn, …);

                // 이벤트객체생성
                seCln = WSACreateEvent();

                // 접속한 클라이언트도 비동기로 통지 받을 수 있도록
                // <소켓, 이벤트 객체, 비동기 통지 이벤트 종류> 바인딩
                WSAEventSelect(scCln, seCln, FD_READ|FD_WRITE|FD_CLOSE);
                …
            }
            …
        }// for
…
```

<nw42_ev_th.zip>

클라이언트도 서버와 마찬가지로 주-스레드는 네트워크 컨테츠로 채우고 비동기 통지를 처리하는 스레드를 별도로 만듭니다.

```
// 클라이언트
…
HANDLE          g_hThRcv;       //
DWORD WINAPI    WorkRcv(void*); // 이벤트 통지를 처리용 스레드
…
int main()
…
g_scHost = socket(…);
g_seHost = WSACreateEvent();    // event 객체 생성
…
// <소켓, 이벤트, 이벤트 통지 종류> 바인딩
hr = WSAEventSelect(g_scHost, g_seHost
                        , (FD_CONNECT|FD_READ|FD_WRITE|FD_CLOSE));

hr = connect(g_scHost, …);
…
g_hThRcv = (HANDLE)_beginthreadex(… WorkRcv, …); // 비동기 통지용 스레드 생성
…
while(g_scHost)
{
        // 클라이언트 컨텐츠 실행
}
…
```

클라이언트의 이벤트는 하나 밖에 없으므로 비동기 처리용 스레드에서 신호 상태의 이벤트를 기다리는 WFME 함수 대신 WaitForSingleObject() 함수를 사용해도 됩니다.

```
// 비동기 통지용 스레드
DWORD WINAPI WorkRcv(void* pParam)
…
while(g_scHost)
…
        WSANETWORKEVENTS wnE={0};
```

```
        // event가 신호 상태가될 때까지 기다림
        //hr = WaitForSingleObject(g_seHost, INFINITE);
        hr = WSAWaitForMultipleEvents(1, &g_seHost, FALSE, WSA_INFINITE, FALSE);
        …
        // event 분해
        hr = WSAEnumNetworkEvents(…);

        // connection event
        if( FD_CONNECT & wnE.lNetworkEvents)
        {
                if(wnE.iErrorCode[FD_CONNECT_BIT])
                        break;
                …
        }

        // 나머지 FD_READ ~ FD_CLOSE 처리
        else if( FD_READ & wnE.lNetworkEvents)
        …
```

<nw42_ev_th.zip>

main() 함수에서 이벤트 통지를 처리하지 않고 별도의 스레드를 두어서 이를 처리하도록 구현해봤습니다. 마지막으로 send용 스레드를 추가하는 작업이 남았습니다. AsyncSelect와 EventSelect는 처리과정이 상당히 유사하므로 AsyncSelect에서 사용했던 send용 스레드를 그대로 가져와 붙여도 됩니다.

```
// 서버
HANDLE          g_hThSnd = NULL;        // Send용스레드핸들
HANDLE          g_hThRcv = NULL;        // 비동기 통지용 스레드 핸들
DWORD WINAPI    WorkSnd(void*);         // Send용스레드
DWORD WINAPI    WorkRcv(void*);         // 비동기 통지용 스레드
…
int main()
…
        hr = listen(…);
…
        //데이터 송신용 스레드 생성. 대기상태
```

```
        g_hThSnd = (HANDLE)_beginthreadex(… WorkSnd, NULL, CREATE_SUSPENDED, NULL);

        // 비동기 통지용 스레드 생성
        g_hThRcv = (HANDLE)_beginthreadex(… WorkRcv, NULL, 0, NULL);
…
```

송신 스레드와 관련된 코드에서 주의할 것은 send가 완료 될 때까지 송신 버퍼가 변경되지 않도록 임계 구역과 같은 동기화 객체를 사용하고 송신 스레드가 CPU를 계속 점유하지 못하도록 송신 버퍼에 보낼 데이터가 있을 때만 재개(Resume)하고 전송이 끝나면 일시 중지 상태(Suspend)로 변경하도록 합니다.

```
// 서버 송신 스레드
DWORD WINAPI WorkSnd(void *pParam)
…
while(1)
{
        EnterCriticalSection(…);
        …
        for(INT i=1; i<iSize; ++i)
        {
                RemoteHost* pCln = g_vHost[i];
                …
                // 데이터 송신
                iSnd = send(pCln->scH, p, iLen-iTot, 0);
                …
                pCln->nBuf = 0;
        }
        LeaveCriticalSection(…);

        // 송신 스레드를 일시 중지 상태로 변경
        HANDLE hThread = GetCurrentThread();
        SuspendThread(hThread);
…
```

<nw43_ev_sn.zip>

클라이언트 역시 서버와 유사하게 송신 스레드를 만들고, 송신 스레드는 보낼 데이터가 있을 때만 재개하고 전송이 끝나면 일시 중지 상태로 전환하도록 하며 송신 버퍼에 데이터를 기록하거나 전

송(send) 하는 경우 동기화 객체로 Lock/Unlock을 진행합니다.

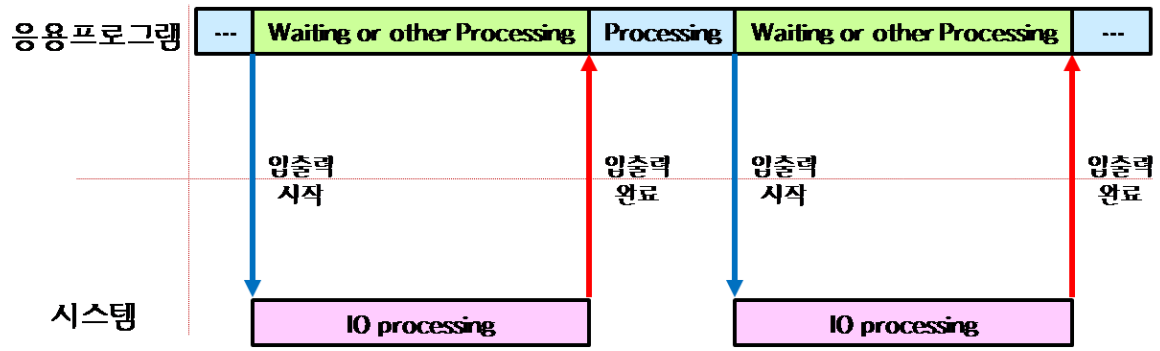
```
// 클라이언트
…
char            g_sBuf[MAX_BUF+4]={0};  // 송신 버퍼
DWORD WINAPI    WorkSnd(void*);         // 송신용 스레드
…
int main()
…
hr = connect(…);
// 데이터 송신용 스레드 생성. 일시 중지 상태
g_hThSnd = (HANDLE)_beginthreadex(… WorkSnd, NULL, CREATE_SUSPENDED, NULL);

while(…)
        …
        fgets(sSnd, MAX_BUF, stdin);            // 채팅 문자열 읽기
        …
        EnterCriticalSection(&m_CS);            // Lock
                g_nBuf= iLen;
                memcpy(g_sBuf, sSnd, iLen);     // 송신 버퍼에 채우기
        LeaveCriticalSection(&m_CS);            // Unlock
        ResumeThread(g_hThSnd);                 // 송신 스레드 재개
…

// 클라이언트 송신용 스레드
DWORD WINAPI WorkSnd(void* pParam)
…
while(…)
{
        EnterCriticalSection(&m_CS);            // Lock
        hr = send(…);
        …
        memset(g_sBuf, 0, g_nBuf);              // 송신 버퍼 지우기
        …
        LeaveCriticalSection(&m_CS);            // Unlock
        …
        HANDLE hThread = GetCurrentThread();    // 송신 스레드 일시 중지
```

```
        SuspendThread(hThread);
...
```

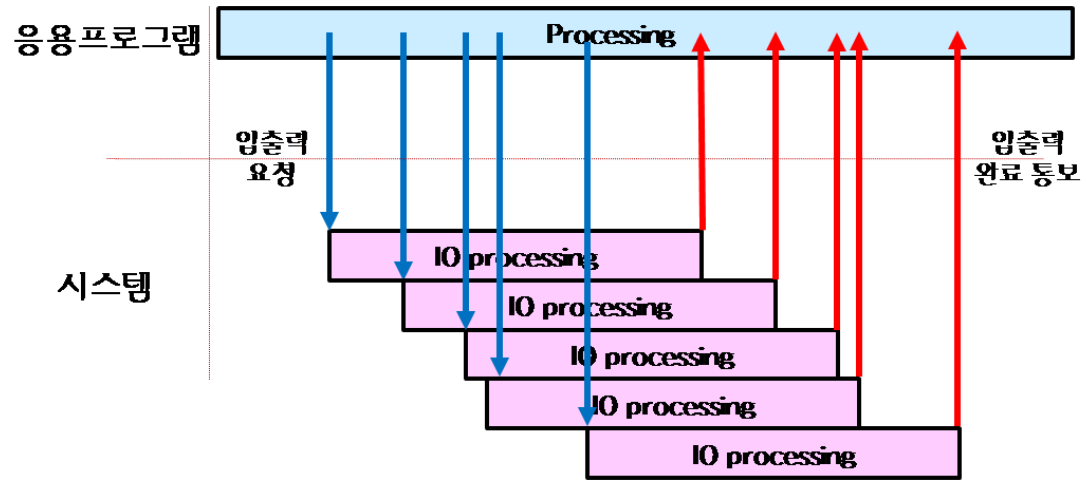
<nw43_ev_sn.zip>

4 Overlapped I/O (중첩 입출력 모델)

윈속 기초에서 마지막으로 소개할 내용은 Overlapped I/O 모델입니다. Overlapped I/O Model(이하 중첩 입출력 모델)은 윈도우 시스템에서 윈속 뿐만 아니라 입출력과 관련된 모든 분야에서 사용되는 윈도우 시스템의 핵심 기술이라고 할 수 있습니다.
중첩 입출력과 동기 입출력 모델을 비교해보면 동기 입출력은 그림과 같이 하나의 입출력이 끝나면 다음 입출력을 진행하는 시스템입니다.



<동기식 입출력 모델>



<중첩 입출력 모델>

중첩 입출력은 이전에 접속한 모든 클라이언트에게 송신과 수신 스레드를 만들어주고 입출력을 진행했던 것과 비슷하게 입출력을 시스템에게 요청하고 나서 입출력 완료를 기다리는 구조가 아니라 이전의 입출력이 완료되지 않아도 다음 입출력을 진행하는 방식입니다.

지금까지 우리가 사용했던 BSD 소켓 함수들의 역할을 보면 send() 함수를 실행하는 것은 보낼 버퍼의 내용을 소켓의 전송 버퍼에 복사하고 소켓은 전송 버퍼의 내용을 전송합니다. recv() 함수의

역할도 비슷하게 소켓의 수신 버퍼에 순차적으로 쌓여 있는 데이터를 사용자가 지정한 버퍼로 복사합니다.

윈속에서 중첩 입출력을 사용하려면 기존의 BSD 함수로 데이터를 송수신 하기가 어렵다라는 결론이 나옵니다. 따라서 시스템에게 중첩 입출력을 요청하는 WSASend(), WSARecv() 함수가 존재하고, GetOverLappedResult, GetQueuedCompletionStatus 함수와 같이 시스템이 입출력을 끝내고 통지를 받을 수 있는 함수들이 존재합니다.

윈속의 중첩 입출력은 강력한 기능을 주지만 주의할 점이 몇 가지 있습니다. 먼저 입출력이 완료되지 않은 상황에서 버퍼의 내용을 변경하거나 버퍼가 유효하지 않으면 문제점이 발생합니다. 예를 들어 송수신 버퍼를 지역 변수로 설정하고 중첩 입출력을 요청하는 경우입니다. 입출력 완료 전에 스택을 빠져 나오게 되면 버퍼가 더 이상 유효하지 않아 에러를 만들어 냅니다.

또 다른 예는 데이터를 전송할 때 같은 버퍼로 다른 내용을 연속해서 보내는 경우 입니다. 네트워크 통신은 프로세서 입장에서 보면 아주 느린 저속장치 입니다. 따라서 송신이 완료되지 않은 상황에서 수신 측은 전혀 다른 데이터를 받을 수가 있습니다. 이 문제는 입출력의 원칙을 정하고 반드시 입출력 요청은 그 횟수만큼 완료 처리를 하거나 다른 버퍼를 사용해서 요청하도록 합니다.

세 번째로 WSASend, WSARecv 함수에 사용되는 WSAOVERLAPPED 변수의 사용입니다. 이 구조체 변수도 사용자의 송수신 버퍼와 마찬가지로 입출력이 완료될 때까지 유효해야 하고 완료될 때까지 다른 입출력 요청의 변수로 사용해서는 안됩니다. 간혹, 송신과 수신 프로그램이 귀찮아 이 변수를 번갈아 사용하는 경우도 있으나 대단히 위험한 코드가 됩니다. 반드시 송신과 수신에 각각 새로 할당해서 사용하는 것이 바람직합니다.

중첩 입출력 원리와 몇 가지 주의사항을 이야기 했습니다. 중첩 입출력을 사용하기 위해서 필요한 WIN API 함수들을 살펴보겠습니다.

```
SOCKET WSASocket(int af, int type, int protocol
                , LPWSAPROTOCOL_INFO lpProtocolInfo, GROUP g, DWORD dwFlags)
```

WSASocket() 함수는 동기, 비동기 소켓 모두를 만들 수 있는 함수 입니다. 이 함수로 중첩을 지원하는 비동기 입출력 소켓을 생성하려면 마지막에 반드시 WSA_FLAG_OVERPAPPED 값을 넣고 다음과 같이 작성합니다.

```
scHost = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, 0, WSA_FLAG_OVERLAPPED)
```

<TCP용 중첩 입출력 소켓>

다행인 것은 이 함수를 사용하지 않아도 윈속에서 지원하는 socket() 함수는 중첩 입출력이 기본이 되는 소켓입니다.

```
int WSASend(SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount
        , LPDWORD lpNumberOfBytesSent, DWORD dwFlags
        , LPWSAOVERLAPPED lpOverlapped
        , LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE)

int WSARecv(SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount
        , LPDWORD lpNumberOfBytesRecvd, LPDWORD lpFlags
        , LPWSAOVERLAPPED lpOverlapped
        , LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine)

BOOL WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite
        , LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped)

BOOL WINAPI ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesToRead
        , LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped)
```

중첩 입출력에서 더 이상 BSD 소켓 함수는 사용할 수 없습니다. 만약 중첩된 데이터 전송을 요청하려면 send() 함수 대신 WSASend()를 사용합니다. WriteFile()를 사용해서 데이터를 전송할 수도 있지만 윈속에 특화된 WSASend()함수를 사용합니다.

WSASend()/WSARecv() 함수의 첫 번째 인수는 소켓, 두 번째 인수는 WSABUF 배열을 지정하고, 세 번째 인수는 WSABUF 배열의 개수를 지정합니다.

```
typedef struct __WSABUF {
    u_long   len;         // 송수신할 데이터 길이
    char FAR *buf;        // 송수신 버퍼 주소
} WSABUF, *LPWSABUF;
```

WSABUF 구조체는 송수신할 버퍼의 주소와 버퍼의 크기를 지정하기 위한 변수입니다. WSASend()/WSARecv() 함수는 이렇게 WSABUF의 배열을 지원되기 때문에 송수신할 버퍼가 연속으로 구성되어 있지 않고 흩어져 있어도 한꺼번에 송수신할 수 있습니다. (Gather/Scatter)

네 번째 변수는 전송된 바이트의 수의 반환 값입니다. WSASend()/WSARecv() 함수는 호출 즉시, 반환 값을 받는데 만약 전송 중이면 이 변수에는 아무 것도 기록이 안되어 있을 것입니다.

5 번째 인수는 OOB와 같은 특별한 전송 형태를 지정할 때 사용합니다. 보통 '0'으로 설정합니다.

6 번째 인수는 WSAOVERLAPPED 구조체 변수 주소로 2 번째의 사용자가 지정한 버퍼 위치와 함께 가장 주의 해야 할 부분입니다.

```
typedef struct _WSAOVERLAPPED {
    ULONG_PTR Internal;
    ULONG_PTR InternalHigh;
    union {
        struct { DWORD Offset;  DWORD OffsetHigh;  };
        PVOID  Pointer;
    };
    HANDLE    hEvent;
} WSAOVERLAPPED, *LPWSAOVERLAPPED;
```

WSAOVERLAPPED 구조체는 OVERLAPPED 구조체와 호환이 되며 이 구조체는 중첩 입출력 실행의 시작과 이후 완료 사이에서의 통신 중계자(매체)로 제공됩니다. 중첩 입출력 시작에서 지정된 WSAOVERLAPPED 구조체 변수의 주소는 완료 시점에서 그대로 반환을 받습니다. 따라서 이 구조체 변수 주소를 완료 시점에서 얻게 된다면 어느 객체가 처음에 입출력을 요청했는지 확인할 수 있는 열쇠가 됩니다.
WSASend()/WSARecv() 함수의 마지막 인수는 완료 통지를 완료 루틴 함수로 처리할 때 사용되며 이후 '완료 루틴 통지'에서 다시 다루겠습니다.

지금까지 중첩 입출력 개념과 관련된 송수신 함수들을 살펴보았습니다. 다음으로 중첩 입출력 결과를 통지 받는 방법에 대해서 알아보겠습니다.
윈속은 중첩 입출력 완료에 대해서 이벤트, 완료 루틴, 그리고 완료 포트 3가지로 통지 받을 수 있습니다. 이들은 파일 입출력과 거의 같기 때문에 중첩 입출력을 좀 더 쉽게 이해할 수 있도록 파일 입출력과 병행해서 설명하겠습니다.

4.1 Event Notify

우리는 EventSelect에서 이벤트 객체로 비동기식 소켓의 통지를 처리해 보았습니다. 만약 중첩 입출력을 이벤트 객체로 통지 받는다면 처리하는 함수만 다를 뿐 전체 프로그램 구조는 EventSelect와 거의 동일하다고 할 수 있습니다.

```
BOOL WSAGetOverlappedResult(SOCKET s, LPWSAOVERLAPPED lpOverlapped
```

```
                , LPDWORD lpcbTransfer, BOOL fWait, LPDWORD lpdwFlags)

BOOL GetOverlappedResult(HANDLE hFile, LPOVERLAPPED lpOverlapped
                , LPDWORD lpNumberOfBytesTransferred, BOOL bWait)
```

WSAGetOverlappedResult() / GetOverlappedResult() 함수는 중첩 입출력에 대해서 이벤트로 통지 받고자 할 때 작업의 결과를 통지 받는 함수입니다.
첫 번째, 두 번째 인수는 각각 입출력을 요청한 소켓과 OVERLAPPED 구조체 변수의 주소를 입력합니다. 4 번째 인수는 입출력 작업이 종료할 때까지의 대기 유 무인데 우리는 EventSelect와 같이 WSAWaitForMultipleEvents() 함수로 이벤트를 기다릴 것이므로 이 값은 FALSE로 설정합니다.
5 번째 인수는 사용하지 않는 변수로 0으로 값을 설정하고 주소를 전단합니다.

이 함수는 요청한 작업을 성공했다면 TRUE를 반환하고 3 번째 인수 lpcbTransfer에 전송한 길이를 저장합니다. 만약 FALSE를 반환하면 작업이 진행 중 (WSA_IO_INCOMPLETE), 또는 작업이 끝났지만 에러가 발생했거나 인수가 문제가 있어서 에러가 발생하는 경우입니다.

이벤트 객체를 사용한 비동기식 소켓의 통지 프로그램은 구조가 EventSelect와 거의 같고 차이점은 WSAEnumNetworkEvents() 함수 대신 WSAGetOverlappedResult() 또는 GetOverlappedResult() 함수를 정도 사용하는 정도 입니다.

파일은 원격 호스트가 필요 없기 때문에 I/O 모델을 쉽게 연습할 수 있습니다. 간단하게 파일에서 특정한 길이만큼 읽어오고 출력하는 중첩 입출력 프로그램을 작성해 보겠습니다.
먼저 다음과 같이 파일 핸들과 OVERLAPPED 구조체를 포함한 5개의 변수를 전역으로 설정하고 입출력 완료를 처리한 작업 스레드(Work thread)를 선언합니다.

```
HANDLE          g_hFile = NULL;         // File Handle
DWORD           g_TotalSize= 0;         // 전체 파일 사이즈
DWORD           g_TotalRead= 0;         // 읽어온 크기
OVERLAPPED      g_rOL   ={0};
char            g_rBuf[MAX_BUF+4]={0}; // 입력 버퍼
DWORD   WINAPI  WorkThread(void*);      // Work 쓰레드
```

메인 함수는 중첩이 가능하도록 FILE_FLAG_OVERLAPPED 상수 값을 사용해서 파일 객체를 생성합니다. 또한 GetFileSize() 함수로 파일의 전체 사이즈를 구합니다.
순차적으로 이벤트 객체를 생성합니다. 여기서는 WaitForSingleObject() 함수로 이벤트를 기다리기 때문에 자동 리셋, 비신호 상태 시작의 이벤트를 생성합니다. 이 이벤트는 OVERLAPPED 구조체 변수 hEvent에 저장합니다.

입출력 통지를 처리할 작업 스레드를 만들고 마지막으로 비동기 입력(Read)을 요청합니다.

```
int main()
…
// 중첩 입출력이 가능한 파일 객체 생성
g_hFile = CreateFile("tiger.txt", GENERIC_READ | GENERIC_WRITE, 0, NULL
                , OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL|FILE_FLAG_OVERLAPPED, NULL);
…
// 전체 파일 길이 얻기
if(0 == (g_TotalSize = GetFileSize(g_hFile, NULL)) )
        return -1;

// 이벤트 객체 생성(자동 리셋, 비신호 상태 시작)
g_rOL.hEvent =CreateEvent(NULL, FALSE, FALSE, NULL);

// Work 쓰레드생성
HANDLE hWork = (HANDLE)_beginthreadex(…WorkThread, …);

DWORD   dTran = 0;
memset(g_rBuf, 0, MAX_BUF); // 버퍼 초기화

// 비동기read 요청
hr= ReadFile(g_hFile, g_rBuf, MAX_BUF, &dTran, &g_rOL);
if(ERROR_SUCCESS == hr)
{
        hr = GetLastError();
        if(ERROR_IO_PENDING != hr)
                return -1;
}
…
```

비동기 입출력 요청에서 에러가 발생했다면 GetLastError() 함수로 에러의 원인을 반드시 찾아야 합니다. 예를 들어 IO가 요청 중이면 ReadFile() 함수는 ERROR_SUCCESS 를 반환할 것이고 에러 코드는 ERROR_IO_PENDING을 반환할 가능성이 큽니다.

다음으로 작업 스레드는 WaitForSingleObject() 함수를 사용해서 이벤트가 신호 상태, 즉 입출력 완료 통지를 기다립니다.

완료가 되면 ReadFile() 세 번째 인수로 지정된 버퍼에 데이터가 채워질 것입니다. GetOverlappedResult() 함수로 읽어온 파일의 길이를 확인 하고 읽어온 누적된 길이가 전체 파일의 길이보다 작다면 ReadFile() 함수를 사용해서 비동기 입출력을 다시 요청합니다.

```
DWORD WINAPI WorkThread(void* pParam)
...
while(1)
{
        // 입출력 완료를 기다림
        hr = WaitForSingleObject(g_rOL.hEvent, INFINITE);
...
        // 완료 결과 해석. 전송된 길이 얻기
        hr = GetOverlappedResult(g_hFile, &g_rOL, &dTran, FALSE);
...
        printf(g_rBuf);                 // 버퍼 출력

        g_TotalRead += dTran;           // 읽어온 전체 길이 누적
        if(g_TotalSize <= g_TotalRead)  // 파일 길이보다 크거나 같으면 종료
                break;

        g_rOL.Offset += dTran;          // offset을 다시 설정
        memset(g_rBuf, 0, MAX_BUF);     // buffer를 초기화

        // 비동기 read 재 요청
        hr= ReadFile(g_hFile, g_rBuf, MAX_BUF, &dTran, &g_rOL);
...
```

<비동기 입출력 이벤트 통지: nw51_file_ev.zip>

막상 코드를 보니 별로 어려움이 없어 보입니다. 이제 서버 부분을 작성해 보도록 합시다. 전에

사용했던 RemoteHost 구조체에 WSAOVERLAPPED, WSABUF를 추가할 것인데 이들 변수는 수신에만 사용하고 송신은 BSD 소켓 입출력 방식으로 전송하겠습니다.

```
struct RemoteHost
{
        WSAOVERLAPPED   ol;
        WSABUF          wsBuf;
        SOCKET          scH;
        char            csBuf[MAX_BUF+4];       // 읽기 버퍼
…
        RemoteHost(SOCKET s, WSAEVENT e)
                …
                wsBuf.len = MAX_BUF;
                wsBuf.buf = csBuf;
                ol.hEvent = e;                  // OVERLAPPED 변수에 이벤트 저장
                scH       = s;
…
SOCKET                  g_scLstn = 0;           // listen socket
vector<RemoteHost* >    g_vHost;                // host list
```

서버는 클라이언트와 다르게 EventSelect처럼 이벤트를 WFME 함수로 기다리고, 다시 WFME 함수로 신호 상태의 이벤트를 찾아야 됩니다. 따라서 이벤트 객체는 수동 리셋 모드 이벤트가 적합합니다. WSACreateEvent() 함수로 리슨 소켓용 이벤트 객체를 생성합니다. 이 이벤트 객체를 신호 상태로 만들어서 작업 스레드가 교착 상태가 되지 않도록 합니다.

```
int main()
…
// 리슨 소켓 용 이벤트 객체 생성
seLstn = WSACreateEvent();

// 호스트 리스트에 리슨 소켓, 이벤트 추가
g_vHost.push_back(new RemoteHost(g_scLstn, seLstn));

// work 스레드생성
HANDLE hWork = _beginthreadex(…WorkThread, …);

// work 스레드에서 dead lock이 안되도록 이벤트를 신호 상태로 전환
```

```
g_vHost[0]->SetEvent();
…
```

클라이어트가 접속하면 이벤트 객체를 생성하고 곧바로 WSARecv() 함수를 호출해서 클라이언트가 송신하는 데이터를 기다립니다.
WSARecv() 함수 실행 후에 리슨 소켓과 같이 저장된 이벤트 객체를 신호 상태로 전환합니다. 이렇게 해야 작업 스레드에서 이벤트 리스트를 다시 갱신하기 때문입니다.

```
//main() 계속
while(1)
{
        …
        scCln = accept(g_scLstn, …);
        …
        // read용이벤트생성
        seCln           = WSACreateEvent();
        RemoteHost* pCln = new RemoteHost(scCln, seCln);
        …
        // 비동기 입출력 요청
        hr = WSARecv(pCln->scH, &pCln->wsBuf, 1
                , &dTran, &dFlag, &pCln->ol, NULL);
        if(SOCKET_ERROR == hr)
                …

        // 작업 스레드에 클라이언트 추가 알림
        g_vHost[0]->SetEvent();
…
```

수신 요청에 대해서 WSARecv() 함수 대신 ReadFile() 함수를 사용할 수도 있지만 좋은 방법은 아닙니다.

```
hr = ReadFile((HANDLE)pCln->scH, pCln->csBuf, MAX_BUF, &dTran, &pCln->ol);
```

MSDN에 의하면 IFS(Installable File System) 소켓은 ReadFile, WriteFile 과 같은 비 윈속 함수들을 사용해도 성능에 영향을 주지 않지만 비-IFS(non-IFS) 소켓은 비 윈속 함수를 사용하게 되면 성능에 심각한 영향을 주는 결과를 만들 수 있다고 합니다.
또한 소켓을 비 윈속 함수로 처리하게 되면 함수의 오버헤드 뿐만 아니라 에러가 발생했을 때 에

러 코드 모두가 윈속 에러 코드로 매핑 되지 않는다고 합니다. 결론적으로 ReadFile, WriteFile 대신 WSARecv, WSASend 함수를 사용하는 것이 바람직하다고 합니다.

입출력을 요청하고 나서 비동기 입출력의 완료를 통지 받기 위해서 WFME 함수로 WSAOVERLAPPED 구조체에 저장된 이벤트 객체가 신호 상태가 될 때까지 무한히 기다리며 이것은 작업 스레드에서 진행합니다.
WFME 함수의 반환이 있으면 EventSelect에서와 동일하게 어떤 이벤트 객체가 신호 상태인지 대기 시간을 0으로 설정하고 각각의 이벤트 객체에 대해서 다시 한번 WFME 함수를 호출해서 신호 상태의 이벤트를 찾습니다.

신호 상태 이벤트를 찾았으면 이벤트를 가지고 호스트 객체와 소켓을 찾고, 수동 리셋 모드 이벤트를 비신호 상태로 전환하며 WSAGetOverlappedResult() 함수로 비동기 입출력 결과를 확인합니다. 참고로 수신된 데이터는 WSARecv() 함수로 입출력을 요청했을 때 지정된 버퍼에 저장되어 있고 이를 가지고 서버의 또 다른 필요한 작업을 진행하면 됩니다.

수신 버퍼를 초기화 하고, WSARecv() 함수로 수신을 다시 요청해서 이후에 클라이언트가 송신하는 데이터를 비동기적으로 받도록 합니다.

```
// 비동기 입출력 완료 처리 함수
DWORD WINAPI WorkThread(void* pParam)
...
while(1)
{
        ...
        // 신호 상태 이벤트 기다림
        nE = WSAWaitForMultipleEvents(nLst, vEvn, FALSE, WSA_INFINITE, FALSE);
        ...
        for(i= nE; i<nLst; ++i)                 // 신호 상태 이벤트 찾기
        {
                hr = WSAWaitForMultipleEvents(1, &vEvn[i], TRUE, 0, FALSE);
                ...
                pHost = FindHost(vEvn[i]);      // 호스트 객체 찾기
                ...
                SOCKET scHost = pHost->scH;
                pHost->ResetEvent();            // 비신호 상태로 전환

                if(g_scLstn == scHost)          // listen socket 이벤트
```

```
                continue;

            // 비동기 입출력 결과 확인
            hr = WSAGetOverlappedResult(scHost, &pHost->ol, &dTran, FALSE, &dFlag);
            …
            if(0<dTran)     // 수신 데이터 존재
                printf("Recv from Client : %d %sWn", (int)scHost, pHost->csBuf);
                …
            // 수신 버퍼 초기화
            memset(pHost->csBuf, 0, MAX_BUF);
            // 다음에 다시 수신할 수 있도록 재 요청
            hr = WSARecv(scHost, &pHost->wsBuf, 1
                , &dTran, &dFlag, &pHost->ol, NULL);
            …
        }
…
```

중첩 입출력에서 다음과 같이 WSASend()함수의 6, 7 번째 인수가 NULL로 설정이 되면 이 함수는 중첩 입출력이 아닌 동기식 입출력을 진행하게 되며 결과적으로 send()함수를 호출한 것과 동일한 결과가 됩니다.

```
WSASend(scHost, &wsBuf, 1, &dSent, 0, NULL, NULL); // send(scHost, s, iLen, 0);
…
```

< 중첩 입출력 이벤트 통지 서버: nw52_ol_ev.zip>

중첩 입출력의 어려운 문제 중에 하나가 WSASend, WSARecv 함수에서 필요한 인수를 제어하기가 어렵고, WSABUF에 사용자가 지정한 버퍼를 연결하고 길이를 쓰는 등 데이터를 송수신하기 위해서 작업할 일이 많다는 것입니다.
여러 방법이 있지만 가장 편리한 방법은 C++의 상속을 이용해서 WSAOVERLAPPED 구조체를 확장하고 WSASend/WSARecv 함수의 인수와 관련된 자료 구조들을 이 확장된 구조체에 추가하는 것입니다. 주의할 것은 가상 테이블(virtual table)을 만들지 않도록 절대로 가상 함수(virtual)를 사용해서는 안됩니다.

```
// 확장된 OVERLAPPED 구조체
struct OVERLAP_EX : public WSAOVERLAPPED
{
```

```
        DWORD           dTran;                  // Transfered
        DWORD           dFlag;                  // Flag
        WSABUF          wsBuf;
        char            csBuf[MAX_BUF+4];       // 송신 또는 수신 버퍼
…
```

만약 C언어로 작성을 하기 위해 상속을 사용하지 않는다면 다음과 같이 구조체의 가장 앞부분에 WSAOVERLAPPED 필드를 선언해도 됩니다.

```
// 또 다른 확장된 OVERLAPPED 구조체
struct OVERLAP_EX
{
        WSAOVERLAPPED   OL;
        DWORD           dTran;                  // Transfered
        DWORD           dFlag;                  // Flag
        WSABUF          wsBuf;
        char            csBuf[MAX_BUF+4];
…
```

이 확장된 구조체를 가지고 클라이언트의 입출력을 처리해 보도록 하겠습니다. 클라이언트는 접속, 송신, 수신 등으로 작업이 분류가 되고 확장된 구조체를 이들 작업에 연결 하기 위해서 다음과 같이 enum으로 작업을 지정하고 OVERLAP_EX 객체를 생성합니다.

```
enum{ OVL_CONNECT=0, OVL_SEND, OVL_RECV, OVL_TOT, };
OVERLAP_EX      g_ol[OVL_TOT];                  // Overlaps
…
```

클라이언트의 메인 함수는 소켓, 이벤트 객체, 중첩 입출력의 통지를 받을 작업 스레드를 순서대로 생성하고 서버에 connection을 요청합니다.

```
// 클라이언트
int main()
…
g_scHost = socket(…);

g_ol[OVL_CONNECT].hEvent = WSACreateEvent();
g_ol[OVL_SEND   ].hEvent = WSACreateEvent();
```

```
g_ol[OVL_RECV   ].hEvent = WSACreateEvent();

// Work 쓰레드생성
hWork = _beginthreadex(… WorkThread, );
…
hr = connect(g_scHost, …);
…
```

접속이 성공하면 비동기 수신을 요청을 포함한 AsyncRcv() 함수를 실행 합니다. 테스트를 위해서 다음과 같이 채팅 메시지를 전송할 수 있도록 코드를 구성해 봅니다.

```
// 클라이언트 main 계속
…
hr = AsyncRcv();                        // 비동기 수신 요청
…
while(g_scHost)                          // 채팅 process
…
        char    sSnd[MAX_BUF]={0};      // Send용버퍼
        fgets(sSnd, MAX_BUF, stdin);
        iLen = strlen(sSnd);
        …
        AsyncSnd(sSnd, iLen);           // 데이터송신
…
```

비동기 통지에 대해서 이벤트는 송신, 수신 두 가지만 필요하지만 처음에 스레드가 교착 상태가 안되도록 접속을 포함해서 3개의 이벤트 객체를 사용합니다.
서버와 비슷하게 WFME 함수로 신호 상태 이벤트 객체를 찾아내고 WSAGetOverlappedResult() 결과를 확인합니다. 수동 리셋 모드 이벤트를 비 신호 상태로 만들고, 성공적인 입출력에 대한 작업 진행합니다. 이벤트가 CONNECTION, RECV에서 만들어졌다면 비동기 수신을 요청합니다.

```
// 클라이언트 비동기 입출력 통지 처리 함수
DWORD WINAPI WorkThread(void* pParam)
…
while(1)
{
        …
        vEvn[OVL_CONNECT] = g_ol[OVL_CONNECT].hEvent;
```

```
        vEvn[OVL_SEND    ] = g_ol[OVL_SEND    ].hEvent;
        vEvn[OVL_RECV    ] = g_ol[OVL_RECV    ].hEvent;
        nE = WSAWaitForMultipleEvents(OVL_TOT, vEvn, …);
        …
        for(i= nE; i<OVL_TOT; ++i)
        {
                hr = WSAWaitForMultipleEvents(1, &vEvn[i], …);
                …
                // 비동기 입출력  결과 확인
                hr = WSAGetOverlappedResult(g_scHost, &g_ol[i]
                                , &dTran, FALSE, &dFlag);
                …
                g_ol[i].ResetEvent();           // 이벤트 비 신호 상태
                …
                if(OVL_CONNECT == i)
                {
                        hr = AsyncRcv();        // 비동기 수신 요청
                        …
                }
                else if(OVL_SEND == i)          // Send Notifiy
                …
                else if(OVL_RECV == i)          // Recv Notifiy
                {
                        …
                        AsyncRcv();             // 비동기 수신 요청
                }
…
```

복잡한 비동기 입출력 함수를 직접 사용하는 것보다 다음과 같이 함수로 wrapping 하거나 확장된 구조체의 멤버 함수로 포함 시키는 것이 좋습니다.

```
// 비동기 송신 함수
INT AsyncSnd(char* csBuf, int iLen)
…
g_ol[OVL_SEND].wsBuf.len = iLen;
memcpy(g_ol[OVL_SEND].csBuf, csBuf, iLen);
```

```
//hr = WriteFile((HANDLE)g_scHost                 // 송신소켓
//               , &g_ol[OVL_SEND].csBuf          // 송신버퍼포인터
//               , iLen
//               , &g_ol[OVL_SEND].dTran
//               , &g_ol[OVL_SEND]);              // OVERLAPPED 구조체포인터
hr = WSASend(g_scHost                           // 송신소켓
               , &g_ol[OVL_SEND].wsBuf          // 송신버퍼포인터
               , 1                              // 송신버퍼의수
               , &g_ol[OVL_SEND].dTran
               ,  g_ol[OVL_SEND].dFlag
               , &g_ol[OVL_SEND]                // OVERLAPPED 구조체포인터
               , NULL );
...
// 비동기 수신 함수
INT     AsyncRcv()
...
//hr = ReadFile((HANDLE)g_scHost                  // 수신소켓
//               , g_ol[OVL_RECV].csBuf           // 수신버퍼
//               , MAX_BUF
//               , &g_ol[OVL_RECV].dTran
//               , &g_ol[OVL_RECV]);              // OVERLAPPED 구조체포인터
hr = WSARecv(g_scHost                           // 수신소켓
               , &g_ol[OVL_RECV].wsBuf          // 수신버퍼포인터
               , 1                              // 수신버퍼의수
               , &g_ol[OVL_RECV].dTran
               , &g_ol[OVL_RECV].dFlag
               , &g_ol[OVL_RECV]                // OVERLAPPED 구조체포인터
               , NULL );
...
```

< 중첩 입출력 이벤트 통지 클라이언트: nw52_ol_ev.zip>

예제 코드에서 ReadFile, ReadFile로 송수신을 작성할 수 있지만 이것은 이들 함수를 사용할 수도 있다는 것을 보여주는 것 뿐이고 앞서 이야기 했듯이 소켓이면 소켓에 특화된 함수를 사용하는 것이 좋습니다.

4.2 Completion Routine

이벤트 객체로 완료를 통지 받기 위해서 이벤트 객체를 생성하고 WSAWaitForMultipleEvents() 함수와 같은 대기 함수로 이벤트의 신호 상태를 기다렸습니다. 소켓 또는 파일의 비동기 입출력에 대해서 완료 루틴을 사용하면 이벤트 객체가 필요 없고, 완료 통지 함수에서 모든 것을 처리하기 때문에 프로그램 작성이 상당히 간단해집니다. 또한 WFME와 같은 함수는 특정한 숫자만큼만 이벤트를 기다리기 때문에 이 숫자보다 많은 접속자가 생길 경우, 작업 스레드를 더 만들어야 합니다. 하지만 완료 루틴은 그러한 작업이 필요하지 않습니다.
완료 루틴을 사용하기 위해서 WSASend, WSARecv 마지막 인수에 함수 포인터를 전달하는 것으로 모든 작업이 끝납니다.

```
WSASend/WSARecv: void CALLBACK CompletionRoutine(DWORD dError
                , DWORD dTransferred, LPWSAOVERLAPPED lpOverlapped, DWORD dFlags)
ReadFileEx/WriteFileEx: void CALLBACK CompletionRoutine(DWORD dError
                , DWORD dTransferred, LPOVERLAPPED lpOverlapped)
```

WSASend, WSARecv는 5개의 인수로 구성된 완료 함수를 사용하고 파일 입출력은 4개의 인수로 구성된 완료 함수를 사용합니다.
첫 번째 인수 dError는 시스템이 완료를 처리하고 에러가 발생 했을 때 에러 코드를 이 인수로 전달합니다. 에러가 없으면 0으로 설정됩니다.
두 번째 인수 dTransferred는 송수신된 바이트 수입니다. 에러가 발생했거나 접속이 종료되면 0으로 설정됩니다.
세 번째 인수는 lpOverlapped 입출력 요청에서 사용된 구조체 변수의 주소입니다. 이 인수가 가장 중요하며 요청한 주체를 찾는 열쇠가 됩니다. 이 부분은 서버에서 다시 살펴 보겠습니다.

완료 루틴이 구현하기가 쉽지만 이전과 마찬가지로 파일을 가지고 먼저 연습한 후에 소켓에 적용해 보도록 하겠습니다. 먼저 다음과 같이 ReadFile() 함수에서 사용할 완료 루틴 함수를 선언합니다.

```
HANDLE          g_hFile = NULL;                         // File
void  CALLBACK  CompletionRoutine(DWORD, DWORD, LPOVERLAPPED);  // 완료 루틴 함수
INT             AsyncRead();
…
```

메인 함수는 파일 핸들과 작업 스레드를 생성하는 것으로 모든 일이 끝납니다.

```
int main()
…
g_hFile = CreateFile(…, FILE_ATTRIBUTE_NORMAL|FILE_FLAG_OVERLAPPED,…);
```

```
// Work 쓰레드생성
hWork = (HANDLE)_beginthreadex(… WorkThread, …);
…
```

비동기 입출력의 완료 루틴을 사용하기 위해서 작업 스레드는 비동기 입출력을 요청하고 alertable wait 상태로 들어가야 합니다. alertable wait 상태가 되면 시스템은 입출력을 진행하고 그 결과를 APC(Asynchronous Procedure Calls)의 큐(Queue)에 저장된 완료 루틴을 호출하게 됩니다. 완료 루틴이 끝나면 스레드는 alertable wait 상태를 빠져 나오게 됩니다.

파일의 입출력을 완료 루틴을 사용할 때 먼저 비동기 입출력을 요청합니다. 그 다음 SleepEx() 함수와 같이 스레드를 altable wait 상태로 만드는 대기 함수를 호출합니다. 대기 함수가 리턴한 후에 읽어올 데이터가 남아 있으면 이를 다시 반복 합니다.

비동기 입출력 요청 -> alertable wait 진입 -> alertable wait 해제 -> 비동기 입출력 요청

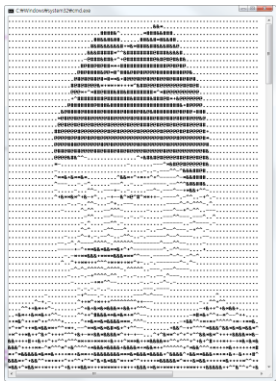
```
DWORD WINAPI WorkThread(void* pParam)
…
while(g_hFile)
{
        // 비동기read 요청
        hr = AsyncRead();
        if(FAILED(hr))
                return -1;

        hr = SleepEx(INFINITE, TRUE);
        if(WAIT_IO_COMPLETION == hr)
                continue;
}
…
INT AsyncRead()
…
        memset(g_rBuf, 0, MAX_BUF);                     // buffer clear
        hr= ReadFileEx(g_hFile, g_rBuf, MAX_BUF         // 비동기read 요청
                , &g_rOL, CompletionRoutine);
…
```

참고로 스레드를 alertable 상태로 만들 수 있는 함수는 SleepEx, WaitForSingleObjectEx, WaitForMultipleObjectsEx, WFME 함수가 있습니다. SleepEx 이외에 나머지 함수들은 이벤트 객체를 요구하기 때문에 alertable 상태가 필요하다면 SleepEx가 가장 무난합니다.

참고로 스레드를 alertable 상태로 만들 수 있는 함수는 SleepEx, WaitForSingleObjectEx, WaitForMultipleObjectsEx, WFME 함수가 있습니다. SleepEx 이외에 나머지 함수들은 이벤트 객체를 요구하기 때문에 alertable 상태가 필요하다면 SleepEx가 가장 무난합니다.

완료 루틴 함수는 읽어온 결과를 출력하고 OVERLAPPED 구조체 변수의 Offset을 조정 합니다. 만약 파일을 다 읽었다면 파일 핸들을 해제합니다. 이렇게 하면 파일 핸들이 유효할 때만 동작하는 스레드가 시간 대기 함수를 빠져나간 후에 파일 핸들을 검사할 것이므로 스레드는 자동으로 종료되게 됩니다.

```
// Completion Routine
void CALLBACK CompletionRoutine(DWORD dErr, DWORD dTran, LPOVERLAPPED pOl)
...
printf(g_rBuf);
g_TotalRead += dTran;
if(g_TotalSize <= g_TotalRead)
{
        CloseHandle(g_hFile);
        g_hFile = 0;
        return ;
}
g_rOL.Offset += dTran;          // offset 조정
...
```



<비동기 입출력 파일 입출력 완료 루틴: nw53_file_cr.zip>

파일 입출력을 통해서 간단히 완료 루틴의 구조를 살펴보았습니다. 기억해야 할 것은 완료 루틴은 비동기 입출력 요청 -> alertable wait 진입 -> alertable wait 해제 -> 비동기 입출력 요청을 반복 한다는 것입니다.

완료 루틴 함수 설명에서 세 번째 인수 lpOverlapped를 가지고 입출력을 요청한 주체(소유주)를 찾는다고 했습니다. 간단히 WSAOVERLAPPED 구조체를 확장시켜 이 구조체를 사용하는 객체의 주소를 저장하는 멤버(Owner)를 추가 시키면 되고, 객체가 생성될 때 Owner를 설정하도록 합니다.

소유주를 추가하는 것 뿐만 아니라 확장 구조체의 쓰임새가 송신인지 수신인지 구분하는 type 필드를 추가합니다. 이 값은 이후에 그 역할에 따라 AsyncSelect에서 사용한 FD_XXX 값으로 설정하도록 합니다.

```
// 확장된 OVERLAPPED 구조체
struct OVERLAP_EX : public WSAOVERLAPPED
{
        void*   pOwn;                   // 이 구조체 소유자
        DWORD   dType;                  // 쓰임새: FD_READ, FD_WRITE
        DWORD   dFlag;                  // 송신 또는 수신용 flag
        DWORD   dTran;                  // 전송 바이트
        WSABUF  wsBuf;                  // WSABUF
        char    csBuf[MAX_BUF+4];        // 송신 또는 수신 버퍼
...
        // 멤버 초기화
        void Reset()
        {
                memset(this, 0, sizeof(WSAOVERLAPPED) );
                memset(csBuf, 0, MAX_BUF+4 );
                dFlag   = 0;
                dTran   = 0;
                wsBuf.len = MAX_BUF;
                wsBuf.buf = csBuf;
        }
        // 버퍼에 데이터 기록
        void SetBuf(char* s, int len)
        {
                wsBuf.len = len;
                memcpy(csBuf, s, len);
```

```
        }

        // 비동기 송신
        int AsyncRcv(SOCKET s, LPWSAOVERLAPPED_COMPLETION_ROUTINE pFunc)
        {
                return WSARecv(s, &wsBuf, 1, &dTran, &dFlag, this, pFunc);
        }

        // 비동기 수신
        int AsyncSnd(SOCKET s, LPWSAOVERLAPPED_COMPLETION_ROUTINE pFunc)
        {
                return WSASend(s, &wsBuf, 1, &dTran, dFlag, this, pFunc);
        }
};
```

멤버 변수 뿐만 아니라 비동기 입출력 함수도 추가해서 사용하기 쉽게 만듭니다.

다음으로 클라이언트 리스트를 포함할 구조체입니다. RemoteHost 구조체 생성자 함수에 송신용 수신용 확장된 OVERLAPPED 구조체의 소유자를 자신(this)로 정하고 FD_XXX 로 역할을 지정합니다.

```
struct RemoteHost
{
        OVERLAP_EX      olRcv;  // 수신용
        OVERLAP_EX      olSnd;  // 송신용
        SOCKET          scH;    // 소켓
...
        RemoteHost(SOCKET s)
        {
                olRcv.pOwn = this;
                olSnd.pOwn = this;
                olRcv.dType= FD_READ;
                olSnd.dType= FD_WRITE;
                scH     = s;
                nUse    = 1;
        }
...
```

만약 alertable wait 상태를 만들기 위해 SleepEx() 함수를 사용한다면 전역으로 설정된 더미 이벤트는 필요가 없으나 여기서는 WFME 함수를 사용하기 때문에 이 이벤트가 필요합니다. WFME 함수는 한 번만 호출되므로 자동 리셋, 비 신호 상태로 이벤트를 생성합니다.

클라이언트가 접속하면 새로운 객체를 생성하고 비동기 수신을 요청합니다. 그리고 더미 이벤트를 신호 상태로 만들어서 작업 스레드가 alertable wait 상태로 다시 진입할 수 있도록 합니다.

```
WSAEVENT        g_eDmmy[1];                     // dummy event
…
void  CALLBACK  CompletionRoutine(DWORD, DWORD
                ,LPWSAOVERLAPPED,DWORD);        // completion routine
…
int main()
…
g_eDmmy[0] = CreateEvent(0,0,0,0);              // 자동 리셋, 비신호 상태
…
while(1)
{
        scCln = accept(g_scLstn, …);
        …
        pCln = new RemoteHost(scCln);           // 새로운 클라이언트 생성
        …
        OVERLAP_EX* polRcv = &pCln->olRcv;
        …
        polRcv->Reset();                        // 초기화
        hr =polRcv->AsyncRcv(pCln->scH          // 수신 요청
                , CompletionRoutine);
        …
        // 이벤트를 신호 상태로 만들어 작업 스레드가
        // alertable 상태로 다시 진입하도록 함
        WSASetEvent(g_eDmmy[0]);
}
```

입출력 결과처리는 완료 루틴에서 하기 때문에 작업 스레드는 특별히 할 일은 없습니다. 만약 WFME 함수로 작업 스레드를 alertable wait 상태로 진입시키기 위해서는 마지막 인수를 TRUE로 설정 합니다.

…

```
DWORD WINAPI WorkThread(void* pParam)
...
while(1)
{
        // hr = SleepEx(INFINITE, TRUE);
        hr = WSAWaitForMultipleEvents(1, g_eDmmy, FALSE
                                , INFINITE, TRUE);      // <- alertable: TRUE
}
```

네트워크 입출력의 완료 루틴 함수 하나로 입출력에 대한 모든 것을 다 넣을 필요는 없습니다. 사실 각각 만드는 것이 더 편리합니다. 아무튼, 많은 예제가 이렇게 작성되어 있고, 그리 긴 코드가 아니기 때문에 여기서도 하나로 처리해 보겠습니다.

완료 루틴을 함수를 구현하는 과정에서 먼저 결정해야 할 것은 입출력을 요청한 소유주를 파악하는 것과 송신 또는 수신에 대한 작업의 타입을 찾는 일입니다.
다행히 OVERLAP_EX 구조체에 입출력 작업의 타입과 소유주를 포함 시켰으므로 다음과 같이 시스템에서 설정한 LPWSAOVERLAPPED 포인터 변수를 OVERLAP_EX 포인터로 형 변환(casting)합니다.
OVERLAP_EX 구조체의 소유주는 void 형 포인터 이므로 이를 RemoteHost 포인터로 형 변환 하면 앞에서 RemoteHost 생성자가 this 포인터를 넘겨주었기 때문에 접속한 클라이언트 객체를 알아 낼 수 있습니다.
이렇게 형 변환을 통해서 입출력을 요청한 객체, 작업 타입, 소켓 등을 찾을 수 있서 에러부터 입출력 작업의 완료에 대한 나머지 일을 진행 할 수 있습니다.

참고로 한 번의 요청은 한 번의 완료만 수행되기 때문에 수신 완료에서 비동기 수신을 다시 요청해서 다음에 원격에서 보낸 데이터를 수신할 수 있도록 합니다.

```
// 완료 루틴
void CALLBACK CompletionRoutine(DWORD dErr, DWORD dTran
                        , LPWSAOVERLAPPED pOl, DWORD dFlag)
...
OVERLAP_EX* pExOl  = (OVERLAP_EX*)pOl;
RemoteHost* pCln   = (RemoteHost*)pExOl->pOwn;
SOCKET      scHost = pCln->scH;

if(dErr != 0 || dTran == 0)             // 에러 또는 접속 종료
        ...
if(FD_WRITE == pExOl->dType)            // 송신 완료
```

```
        ...
else if(FD_READ == pEx01->dType)          // 수신 완료
        ...
        hr =pEx01->AsyncRcv(scHost, CompletionRoutine);  // 수신 요청
...
```

데이터 송신은 AsyncSnd 함수에 객체의 소켓 그리고 함수 포인터를 전달합니다.

```
// echo message
void EchoMsg(char* s, int l)
        ...
        hr = polSnd->AsyncSnd(scHost, CompletionRoutine);
...
```

<중첩 입출력 완료 루틴 서버: nw54_ol_cr.zip>

클라이언트는 소켓 하나만 처리하기 때문에 파일 입출력과 거의 유사합니다. 서버 프로그램에서 사용한 OVERLAP_EX 구조체를 그대로 사용합니다. 다음은 문자열을 보내는 프로그램입니다.

```
// 클라이언트
OVERLAP_EX      g_olRcv(FD_READ );
OVERLAP_EX      g_olSnd(FD_WRITE);
...
int main()
...
while(g_scHost)                                 // 채팅 process
{
        char    sSnd[MAX_BUF]={0};              // Send용버퍼
        ...
        g_olSnd.Reset();                        // 초기화
        g_olSnd.SetBuf(sSnd, iLen);             // 버퍼에 복사
        hr = g_olSnd.AsyncSnd(g_scHost          // 데이터 송신용
                , CompletionRoutine);
...
```

작업 스레드는 먼저 수신 요청을 한 후에 alertable wait 상태로 진입하도록 합니다. 만약 시스템에서 완료가 처리되면 대기 함수를 빠져 나올 것이고 while 루프에 의해 자동으로 수신을 요청하게 됩니다.

```
DWORD WINAPI WorkThread(void* pParam)
...
while(g_scHost)
{
        ...
        g_olRcv.Reset();                        // 초기화
        hr =g_olRcv.AsyncRcv(g_scHost           // 수신 요청
                , CompletionRoutine);
        ...
        hr = SleepEx(INFINITE, TRUE);           // alertable wait 상태 진입
        if(WAIT_IO_COMPLETION == hr)
                continue;
...
```

클라이언트는 OVERLAP_EX 포인터로 형 변환해서 작업 타입을 확인하고 나머지 일을 처리 합니다.

```
void CALLBACK CompletionRoutine(DWORD dErr, DWORD dTran
                        , LPWSAOVERLAPPED pOl, DWORD dFlag)
...
OVERLAP_EX* pExOl = (OVERLAP_EX*)pOl;

if(dErr != 0 || dTran == 0)                     // 에러. 접속 종료
        ...
if(FD_WRITE == pExOl->dType)                    // 전송 완료
        ...
else if(FD_READ == pExOl->dType)                // 수신 완료
        ...
```

<중첩 입출력 완료 루틴 클라이언트: nw54_ol_cr.zip>

WSAOVERLAPPED 구조체를 확장할 것이라면 완료 루틴 함수 포인터도 확장 구조체에 포함 시키는 것이 좋습니다. LPWSAOVERLAPPED_COMPLETION_ROUTINE 이름이 길어서 다음과 같이 typedef로 짧은 이름을 지정하고 확장구조체에 함수 포인터 변수를 포함 합니다.
타입에 따라 송신 또는 수신이 결정되므로 입출력 요청 함수는 AsyncProc()와 같이 하나로 합쳐서 타입에 따라 WSARecv, WSASend 함수를 호출 합니다.

```
typedef LPWSAOVERLAPPED_COMPLETION_ROUTINE    PWSA_COMP_R;
```

```
struct OVERLAP_EX : public WSAOVERLAPPED
...
void*           pOwn;                   // 이 구조체 소유자
DWORD           dType;                  // 쓰임새: FD_READ, FD_WRITE
DWORD           dFlag;                  // 송신 또는 수신용 flag
DWORD           dTran;                  // 전송 바이트
WSABUF          wsBuf;                  // WSABUF
char            csBuf[MAX_BUF+4];       // 송신 또는 수신 버퍼
PWSA_COMP_R     pcsFunc;                // 완료 루틴 함수 포인터

int AsyncProc(SOCKET s)                 // 입출력 프로세스
{
        if(FD_READ == dType)
                return WSARecv(s, &wsBuf, 1, &dTran, &dFlag, this, pcsFunc);
        else if(FD_WRITE == dType)
                return WSASend(s, &wsBuf, 1, &dTran, dFlag, this, pcsFunc);
        return -1;
}
...
```

RemoteHost의 생성자 함수에 송신, 수신 함수 포인터를 전달 받아 OVERLAP_EX 멤버 변수의 작업과 함수 포인터를 연결합니다.

```
struct RemoteHost
...
RemoteHost(SOCKET s, PWSA_COMP_R pFuncSnd, PWSA_COMP_R pFuncRcv)
{
        olSnd.pOwn = this;
        olRcv.pOwn = this;
        olSnd.dType= FD_WRITE;
        olRcv.dType= FD_READ;
        olSnd.pcsFunc= pFuncSnd;
        olRcv.pcsFunc= pFuncRcv;
...
}
...
```

앞서 완료 루틴을 하나로 처리하는 것보다 송신과 수신을 분리해서 처리하는 것이 좋다고 했습니다. 분리하게 되면 타입을 검사 안 하고 바로 처리할 수가 있어서 코드가 간결해집니다.

```
// 송신 완료 루틴
void CALLBACK CompletionSnd(DWORD dErr, DWORD dTran,LPWSAOVERLAPPED pOl, DWORD)
{
        OVERLAP_EX* pExOl = (OVERLAP_EX*)pOl;
…
        printf("Complete sending: %d byte₩n", dTran);
}

// 수신 완료 루틴
void CALLBACK CompletionRcv(DWORD dErr, DWORD dTran,LPWSAOVERLAPPED pOl, DWORD)
{
        OVERLAP_EX* pExOl = (OVERLAP_EX*)pOl;
…
        printf("Recv[%3d]: %s₩n", dTran, pExOl->csBuf);
}
```

서버에서는 클라이언트가 접속하고 새로운 호스트 객체를 생성할 때 할 때마다 이들 함수를 연결하고 클라이언트 코드는 두 개의 전역 변수 g_olSnd , g_olRcv 에 생성할 때 역할, 함수 포인터를 설정합니다.

```
// 서버 코드
scCln = accept(g_scLstn, …);
RemoteHost* pCln = new RemoteHost(scCln, CompletionSnd, CompletionRcv);
…
// 클라이언트 코드
OVERLAP_EX      g_olSnd(FD_WRITE, CompletionSnd);
OVERLAP_EX      g_olRcv(FD_READ, CompletionRcv);
…
```

<송수신 완료 루틴이 분리된 중첩 입출력: nw55_ol_rw.zip>

코드가 쉬우니 나머지는 nw55_ol_rw.zip 예제를 참고하기 바랍니다.

4.3 IOCP (I/O Completion Port)

MSDN의 IOCP 설명을 보면 IOCP는 다중처리 시스템에서 다중 비동기 IO 요청을 처리하는데 효율적인 스레딩 모델을 제공합니다. 만약 프로세스가 IOCP를 만들면 시스템은 입출력 관련된 요청을 처리하기 위해서 요청과 관련된 큐(Queue)객체를 생성합니다.
프로세스의 비동기 IO 요청이 있으면 스레드 풀에 미리 생성해 놓은 스레드와 IOCP를 가지고 작업을 처리하기 때문에 다른 어떤 처리 방식 보다 빠르고 효율적이라 합니다.

간단히 IOCP는 함수 중심의 완료 루틴을 좀 더 발전 시킨 형태로 입출력(I/O) 완료 포트(Completion Port) 객체가 입출력을 처리하고 (완료)결과를 이를 통보하는 시스템입니다.
IOCP의 장점은 완료 루틴과 비교해 보면 금세 알 수 있습니다. 완료 루틴은 alertable wait 상태로 진입하는 스레드에서만 완료를 반환 받을 수 있지만 IOCP는 완료 포트의 결과를 호출하는 모든 스레드(작업 스레드)에서 완료를 반환 받을 수 있습니다. 따라서 스레드의 제약이 없고, 스레드 또한 미리 만들어 놓고 쉬고 있는 스레드를 사용할 수도 있는 구조를 가지고 있습니다.

이렇게 윈도우 시스템에서 다중 입출력 요청에 대해서 가장 효율인 IOCP는 수 백 명 이상 처리해야 하는 네트워크 프로그램에 적용한다면 좀 더 성능이 좋은 서버를 만들 수가 있게 됩니다.
IOCP 프로그램을 위해서 필요한 함수들을 살펴 보겠습니다.

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

이 함수는 두 가지의 기능이 있습니다. 첫 번째는 IOCP 객체를 생성합니다.

```
HADLE hIocp = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
```

CreateIoCompletionPort() 함수의 두 번째는 역할은 IOCP 객체와 입출력 대상이 되는 객체(File Handle), 그리고 사용자가 지정한 완료 키(Completion Key)를 바인딩 하는 함수 입니다.
완료 키는 입출력 대상이 되는 객체에 대한 모든 입출력 완료 요청을 해석하는 열쇠입니다. IOCP는 완료를 끝내고 나서 함수로 완료 작업에 대한 정보를 되돌려 주는데 이 정보에는 입출력 대상이 없고 대신 사용자가 지정한 완료 키를 반환해줍니다.

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

작업 스레드는 입출력의 결과를 통지 받는 함수입니다. 이 스레드에 어떤 특별한 내용을 전달할 필요도 있을 수 있습니다. 이러한 경우 PostQueuedCompletionStatus() 함수를 사용합니다. 이 함

수는 마치 WIN API의 PostMessage()가 메시지 큐에 정보를 전달하는 것처럼 작업 스레드의 큐 영역에 메시지를 전달합니다.

```
BOOL GetQueuedCompletionStatus(HANDLE CompletionPort
                , PDWORD lpNumberOfBytes , PULONG_PTR lpCompletionKey
                , LPOVERLAPPED *lpOverlapped, DWORD dwMilliseconds)
```

GetQueuedCompletionStatus(이하 GQCS) 가장 중요한 함수로 IOCP가 처리한 입출력의 결과를 통보받는 함수입니다. 만약 어떤 스레드가 이 함수를 포함하고 있다면 이 스레드를 작업(자) 스레드라 합니다.

함수의 첫 번째 인수는 입출력을 처리하는 IOCP 객체를 지정합니다. 2 ~ 4 인수는 IOCP 객체가 작업을 처리하고 정보를 반환 하는 인수입니다. 5 번째 인수는 대기 시간으로 보통 INFINITE 값으로 설정합니다.
두 번째 인수 lpNumberOfBytes는 GQCS 가 처리한 byte 수 입니다. 만약 이 값이 0으로 되어 있다면 recv() 함수의 0을 반환 받은 것과 같이 상대방이 접속을 종료한 것입니다.
세 번째 인수 lpCompletionKey는 CreateIoCompletionPort() 함수에서 ICOCP와 연결할 때 FileHandle에 대한 사용자 설정한 완료 키의 주소가 저장됩니다.
네 번째 인수 lpOverlapped는 WSASend, WSARecv 함수에서 사용한 OVERLAPPED 변수의 주소가 저장됩니다.
간단히 인수에 대해서 정리한다면 세 번째 인수는 IOCP에 작업을 요청한 객체를, 네 번째 인수는 송신 또는 수신 작업을 구분해주는 타입이라고 생각해도 됩니다.

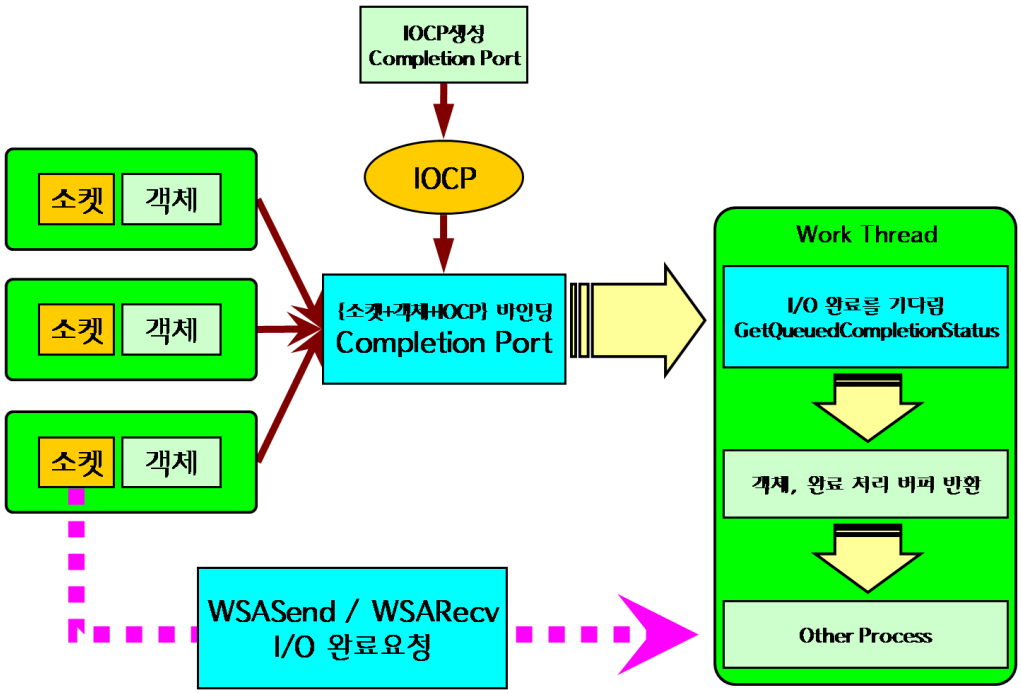
GQCS 함수에 대해서 좀더 살펴 본다면 완료 포트(completion port)가 요청한 입출력을 에러 없이 성공했다면 OGQCS 함수는 주어진 인수 lpNumberOfBytes에는 IOCP가 처리한 바이트 수, lpCompletionKey에는 IOCP와 바인딩 된 객체 주소, 그리고 lpOverlapped에는 입출력 작업 타입을 저장하고 TRUE를 반환합니다.

실패 하면 OGQCS 함수는 FALSE를 반환 합니다. 실패는 (*lpOverlapped)의 값이 NULL 이거나 NULL이 아닌 2 가지 유형이 있습니다. 만약 (*lpOverlapped)의 값이 NULL로 설정되어 있다면 이것은 IOCP가 작업 자체를 완료 하지 못해 lpNumberOfBytes, lpCompletionKey, lpOverlapped에 정보도 기록되지 않는 상황입니다. 두 번째 NULL이 아닌 경우 실패한 입출력 처리의 완료입니다. 이 때는 lpNumberOfBytes, lpCompletionKey, lpOverlapped의 인수에 값이 저장됩니다.

IOCP를 가지고 몇 가지 실험을 해보면 (*lpOverlapped)가 NULL인 경우는 잘못된 또는 유효하지 않은 인수를 GQCS에 적용했거나 대기 시간의 초과로 인해서 발생할 수 있습니다. 예를 들어 프로그램 중간에 GQCS 함수에 요청한 IOCP 객체가 해제되면 GQCS 함수는 실패를 반환하고

(*lpOverlapped)을 NULL로 설정합니다. 이것을 GetLastError() 함수로 에러 코드를 확인해 보면 ERROR_ABANDONED_WAIT_0 에러 코드를 얻을 수 있습니다.
어찌 되었건 실패에 대해서 무조건 GetLastError()로 에러 코드를 얻고 처리하도록 합니다.

지금까지 IOCP에서 사용되는 함수들을 살펴보았습니다. 간단히 전체 프로그램 구조를 정리해 보면 서버는 먼저 CreateIoCompletionPort() 함수로 IOCP를 생성 합니다. 다음으로 GQCS 함수를 호출하는 작업 스레드를 2 * 프로세서 개수만큼 만들고 실행시킵니다.
만약 accept 함수에서 새로운 클라이언트의 접속을 받으면 소켓을 포함한 클라이언트 객체를 생성하고 이 객체를 완료 키로 설정합니다. <소켓, IOCP, 완료 키>를 CreateIoCompletionIoPort 함수로 바인딩 하고 클라이언트의 수신을 받기 위해 WSARecv 함수를 호출합니다.



<IOCP를 사용한 프로그램 구조>

클라이언트는 서버와 접속이 완료되고 나서 작업 스레드를 만들고, 소켓을 완료 키로 설정한 후에 <소켓, IOCP, 소켓>을 CreateIoCompletionIoPort 함수로 바인딩 합니다. 바인딩 후에 WSARecv 함수로 서버의 수신을 기다립니다.

이전의 완료 루틴을 사용한 파일 입출력 예제와 마찬가지로 IOCP 프로그램 구조를 쉽게 이해하기 위해서 파일 입출력을 연습해 보겠습니다.

먼저 다음과 같이 작업 스레드의 숫자를 정하기 위해서 시스템의 Processor 개수를 얻는 함수를 다음과 같이 GetSystemInfo() 함수를 사용해서 작성합니다.

```
int GetSystemProcessorCount()
{
        SYSTEM_INFO SystemInfo;
        GetSystemInfo(&SystemInfo);
        return (int)SystemInfo.dwNumberOfProcessors;
}
```

파일을 읽기 위해서 IOCP를 가지고 다음과 같은 순서대로 프로그램을 작성합니다.

파일 핸들과 IOCP 핸들을 선언하고 메인 함수는 이들을 생성합니다.
다음으로 작업 스레드를 ' 2 * 프로세서 개수'만큼 생성합니다.
파일 핸들을 완료 키로 설정하고 바인딩 합니다.
바인딩이 끝나면 ReadFile() 함수로 비동기 파일 읽기를 요청 합니다.

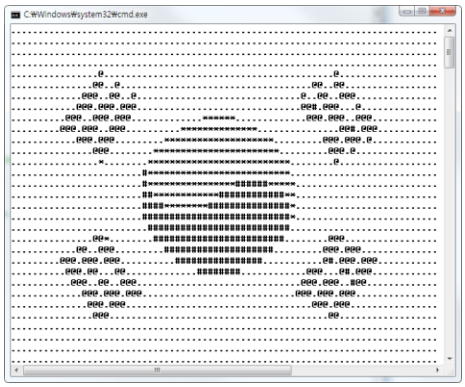
```
HANDLE  g_hFile = NULL; // 파일 핸들
HANDLE  g_hIocp = NULL; // IOCP 핸들
…
int main()
…
g_hFile = CreateFile(…); // 파일 객체 생성
g_hIocp = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0); // IOCP 객체생성
…
int nWrkPrc = 2 * GetSystemProcessorCount();
for(int i=0; i<nWrkPrc; ++i)
        HANDLE hThWrk = (HANDLE)_beginthreadex(… WorkThread, …);
…
ULONG_PTR pIoKey = (ULONG_PTR)&g_hFile;                 // IO Key is File
CreateIoCompletionPort(g_hFile, g_hIocp, pIoKey, 0);    // <IOCP, File, key> 바이딩
…
hr = AsyncRead();                                       // 비동기 Read 요청
```

작업 스레드는 GQCS 함수로 완료를 기다립니다. GQCS 함수의 반환 값에 따라 FALSE 이면 스레드를 빠져 나갑니다.
반환이 TRUE이면서 전송된 데이터가 없으면(0 == dTran) 파일을 다 읽어온 상태이므로 스레드를 빠져나갑니다.
나머지는 IOCP가 파일을 읽어온 내용을 버퍼에 저장해 놓은 상태이므로 버퍼의 내용을 화면에 출

력하고 지금까지 읽어온 바이트 수를 OVERLAPPED 구조체 변수 g_rOL.Offset에 누적하고 전체 파일 길이와 비교해서 크거나 같으면 스레드를 빠져 나갑니다.
전체 파일 길이보다 작은 값이면 입력 버퍼를 초기화 하고 다시 비동기 읽기 요청을 합니다.

```
DWORD WINAPI WorkThread(void* pParam)
...
while(g_hFile)
{
        ULONG_PTR       pIoKey  = NULL;
        OVERLAPPED*     pOL     = NULL;
        DWORD           dTran   = 0;
        DWORD           OLType  = 0;

        hr = GetQueuedCompletionStatus(
                  g_hIocp                       // Completion Port
                , &dTran                        // 전송된바이트수
                , (PULONG_PTR)&pIoKey           // 완료키 주소
                , (LPOVERLAPPED*)&pOL           // OVERLAPPED 구조체 변수 주소
                , INFINITE);
        if(0 == hr)                             // 실패
        {
                hr = GetLastError();
                break;
        }
        if(0 == dTran)                          // 전송할 데이터 없음
        {
                printf("Read Complete.Wn");
                break;
        }

        printf("%s", g_rBuf);                   // 버퍼 출력

        g_rOL.Offset += dTran;                  // offset 조정(읽어온 길이)
        if(g_TotalSize <= g_rOL.Offset)         // 전체 파일 크기 비교
                break;
```

```
        memset(g_rBuf, 0, MAX_BUF);             // 버퍼 초기화
        hr = AsyncRead();                       // 비동기 입출력 요청
        ...
}
...
```



<IOCP 파일 입출력: nw61_iocp_file.zip>

파일 입출력을 IOCP로 연습해보았습니다. 중첩 입출력에서 완료 통지를 이벤트를 사용했을 때와 비슷하게 서버는 수신만 IOCP로 처리하고 송신은 BSD send() 함수를 사용하겠습니다. 서버에서 클라이언트 객체를 저장할 구조체를 OVERLAPPED, WSBUF, 그리고 전송 버퍼를 포함해서 다음과 같이 선언하고 STL 벡터를 사용해서 클라이언트 리스트를 생성 합니다

```
// 클라이언트 객체용 구조체
struct RemoteHost
{
        SOCKET          scH;                    // 소켓
        OVERLAPPED      olRcv;                  // For WSARecv
        WSABUF          wsBuf;
        char            csBuf[MAX_BUF+4];       // Io Completion Buffer
        ...
        void Reset()
        {
                memset(&olRcv, 0, sizeof(OVERLAPPED));
                memset(csBuf, 0, MAX_BUF);
                wsBuf.len = MAX_BUF;
                wsBuf.buf = csBuf;
        }
...
};
```

```
vector<RemoteHost* >    g_vHost;                // Client list
HANDLE  g_hIocp         = NULL;                 // IOCP Handle
…
```

서버의 주-스레드는 CreateIoCompletionPort() IOCP 객체를 생성하고 "2 * 프로세서 개수"만큼 작업 스레드를 생성합니다.
클라이언트가 접속을 하면 클라이언트 객체를 생성하고 이 객체를 완료 키(key)로 설정한 후에 CreateIoCompletionPort() 함수를 사용해서 <클라이언트소켓, IOCP, key>를 바인딩 합니다.
접속한 클라이언트에게서 보낸 데이터를 수신할 수 있도록 WSARecv() 함수를 호출합니다.
WSARecv() 함수의 반환이 SOCKET_ERROR 이면 반드시 WSAGetLastError() 함수를 사용해서 에러 코드를 확인합니다.
WSARecv() 함수 호출 이 성공했다면 vector 컨테이너에 클라이언트 객체를 추가합니다.

```
// 서버 주-스레드
int main()
…
// IOCP 객체생성
g_hIocp = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);

// Work 쓰레드 생성
for(int i=0; i<nWrkPrc; ++i)
        hThWrk = (HANDLE)_beginthreadex(…)
…
while(…)
{
        …
        scCln = accept(g_scLstn, …);

        RemoteHost* pCln = new RemoteHost(scCln);       // 클라이언트 인스턴스 생성
        …
        ULONG_PTR  pIoKey = (ULONG_PTR)pCln;    // Key is client Object
        …
        CreateIoCompletionPort(                 // <클라이언트 소켓, IOCP, key> binding

                (HANDLE)scCln, g_hIocp, pIoKey, 0);
```

```
        // 수신 요청
        hr = WSARecv(pCln->scH, &pCln->wsBuf, 1
                        , &dTran, &dFlag, &pCln->olRcv, NULL);

        // 에러 반환 처리
        if(SOCKET_ERROR == hr)
        {
                hr =  WSAGetLastError();
                if(WSA_IO_PENDING != hr…)
                        …
        // add the client list
…
```

서버의 GQCS 함수를 포함하고 있는 작업 스레드는 먼저 이 함수가 반환한 성공/실패에 대해서 먼저 실패를 처리합니다.
GQCS 함수의 실패는 크게 IOCP객체, 또는 클라이언트 객체에서 발생하기 때문에 IOCP의 처리 결과를 저장하는 PULONG_PTR 타입의 완료 키 인수와 요청한 작업 종류를 파악할 수 있는 LPOVERLAPPED 인수에 어떤 정보도 기록되어 있지 않으면 IOCP객체의 문제이므로 스레드를 빠져나갑니다.
만약 클라이언트와 통신하고 있는 중에 실패하는 경우라면 완료 키를 사용해서 형 변환 등을 사용해서 클라이언트 객체를 찾고 찾은 클라이언트 객체를 해제합니다.

```
// 서버 작업 스레드
DWORD WINAPI WorkThread(void* pParam)
…
RemoteHost*     pCln    = NULL; // 완료 키: 클라이언트 객체 주소
OVERLAPPED*     pOlEx   = NULL; // WSARecv에서 사용된 OVERLAPPED 주소
DWORD           dTran   = 0;    // IOCP가 처리한 바이트 수
while(0 < g_scLstn)
{
        hr = GetQueuedCompletionStatus( g_hIocp, &dTran
                , (PULONG_PTR)&pCln, (LPOVERLAPPED*)&pOlEx, INFINITE);

        if(0 == hr)     // 실패를 반환 받을 때
        {
                …
```

```
            if(NULL == pOlEx && NULL == pCln)       // IOCP에서 에러 발생
                    break;
            if(pOlEx && pCln)               // 클라이언트에서 에러 발생
                    …
            continue;
    }
```

GQCS 함수의 성공 결과를 처리하는 과정에서 GQCS 함수의 두 번째 인수인 IOCP가 처리한 바이트 수가 0 이 되면 클라이언트가 접속을 종료한 것이므로 클라이언트 객체를 해제합니다.
다음으로 WSARecv 함수로 수신 요청된 데이터는 WSABUF에서 지정한 버퍼 주소에 저장될 것입니다. 이 버퍼에 저장된 데이터를 가지고 나머지 일을 처리합니다.
다시 클라이언트가 보낸 데이터를 수신할 수 있도록 WSARecv() 함수를 호출합니다.

```
// 서버 작업 스레드 WorkThread 계속
…
        SOCKET  scHost  = pCln->scH;
        // 접속 클라이언트 종료 처리
        if(0 == dTran)
        {
                printf("Disconnect Client: %dWn", (int)scHost);
                …
        }
        …
        // 수신완료 처리
        printf("Recv from Client[%4d]: %sWn", (int)scHost, pCln->csBuf);
        …
        // 비동기 수신 요청
        hr = WSARecv(…);
        …
```

<IOCP 서버 수신: nw62_iocp_basic.zip>

서버에서 IOCP를 사용해서 데이터를 수신을 해봤습니다. 클라이언트는 구조가 간단하기 때문에 수신 뿐만 아니라 송신도 IOCP 사용해보겠습니다.

IOCP를 사용하기 위해서 다음과 같이 IOCP 핸들, 송수신 OVERLAPPED 구조체 변수, 송수신 버퍼 등을 선언합니다.

```
HANDLE          g_hIocp  = NULL;                // IOCP 핸들

// 송신에 필요한 변수들
OVERLAPPED      g_sndOL   ={0};
DWORD           g_sndTran = 0;
WSABUF          g_sndWsBuf={0};
char            g_sndBuf[MAX_BUF+4]={0};       // 송신용 버퍼

// 수신에 필요한 변수들
OVERLAPPED      g_rcvOL   ={0};
DWORD           g_rcvTran = 0;
WSABUF          g_rcvWsBuf={0};
char            g_rcvBuf[MAX_BUF+4]={0};       // 수신용 버퍼
```

클라이언트에서 필요한 비동기 입출력 함수 사용은 이벤트 통지를 사용한 중첩 입출력과 완료 루틴에서 살펴보았으므로 설명을 생략하겠습니다.

```
INT AsyncSend(char* s, int l)
…
        memcpy(g_sndBuf, s, l);
        g_sndWsBuf.len = l;
        g_sndWsBuf.buf = g_sndBuf;
        int hr= WSASend(g_scHost, …);
…
        return hr;
…
INT AsyncRecv()
…
        g_rcvWsBuf.len = MAX_BUF;
        g_rcvWsBuf.buf = g_rcvBuf;
        return WSARecv(g_scHost, …);
…
```

클라이언트의 주-스레드는 접속을 성공하면 IOCP, 작업 스레드를 생성하고, 완료 키를 소켓으로 설정해서 이들을 CreateIoCompletionPort 함수로 바인딩 합니다.
서버에서 보낼 데이터를 수신하기 위해 비동기 함수로 수신을 요청합니다.

```
// 클라이언트
int main()
…
hr = connect(…);                        // 서버 접속
…
g_hIocp = CreateIoCompletionPort(…);    // IOCP 객체생성
…
_beginthreadex(…);                      // Work 쓰레드 생성
…
ULONG_PTR pIoKey = (ULONG_PTR)&g_scHost; // 소켓을 완료 키로 설정

// <소켓, IOCP, key> 바이딩
CreateIoCompletionPort((HANDLE)g_scHost, g_hIocp, pIoKey, 0);
…
AsyncRecv();                            // 비동기수신요청
…
```

작업 스레드에서 실패와 성공에서 전송된 바이트 수가 0일 경우는 IOCP 객체가 문제이거나 서버와 접속이 해제된 상황이므로 스레드를 빠져 나갑니다.
성공에서 GQCS 함수에서 얻은 LPOVERLAPPED 인수와 송신 또는 수신 OVERLAPPED 구조체 변수의 주소와 비교에서 송신, 수신 완료를 처리합니다.
마지막으로 수신 완료 처리에서 수신을 요청해서 이후에 서버에서 보낸 데이터를 수신할 수 있도록 합니다.

```
// 클라이언트 작업 스레드
DWORD WINAPI WorkThread(void* pParam)
…
while(…)
{
        hr = GetQueuedCompletionStatus( g_hIocp , &dTran
                , (PULONG_PTR)&pIoKey,  (LPOVERLAPPED*)&pOL, INFINITE);

        if(0 == hr)
                …
                break;
```

```
        if(0 == dTran)                          // 접속 종료
                ...
                break;

        // 송신 완료
        if( ((void*)&g_sndOL) == ((void*)pOL) )
        {
                ...
                printf("Write Successed.₩n");
        }
        // 수신 완료
        else if( ((void*)&g_rcvOL) == ((void*)pOL) )
        {
                ...
                printf("Recv from Server(%3d): %s₩n", dTran, g_rcvBuf);
                AsyncRecv();                    // 수신 요청
        }
}
...
```

<IOCP: nw62_iocp_basic.zip>

지금까지 IOCP 를 사용해서 서버에서는 수신을 클라이언트는 송수신을 해보았습니다. 다음으로 채팅 프로그램을 만들어서 서버의 송수신을 완벽하게 IOCP를 사용하고 클라이언트도 서버에서 사용한 구조체로 프로그램을 구성해 보겠습니다.
완료 루틴에서 사용했던 것처럼 WSAOVERLAPPED 구조체를 확장해서 송신과 수신을 처리할 수 있도록 확장 구조체 OVERLAP_EX를 구성합니다.

```
struct OVERLAP_EX : public WSAOVERLAPPED
{
        DWORD   dType;                  // IO 타입. FD_READ 또는 FD_WRITE
        DWORD   dTran;                  // 전송된 바이트 수
        DWORD   dFlag;                  // Flag = 0
        WSABUF  wsBuf;                  // WSABUF
        char    csBuf[MAX_BUF+4];       // 송신 또는 수신 버퍼
...
        void SetBuf(char* s, int l=0)   // 버퍼에 데이터 설정
        {
```

```
            if(s && 0<l)
            {
                    wsBuf.len = l;
                    memcpy(csBuf, s, l);
            }
            else
            {
                    wsBuf.len = MAX_BUF;
            }
    }

    int AsyncRcv(SOCKET s)          // 비동기 수신
    {
            return WSARecv(s, &wsBuf, 1, &dTran, &dFlag, this, NULL);
    }

    int AsyncSnd(SOCKET s)          // 비동기 송신
    {
            int hr = WSASend(s, &wsBuf, 1, &dTran, dFlag, this, NULL);
            if(0 == hr)             // 성공하면 작업 큐에 저장 안됨
                    printf("Send complete: %d byteWn", dTran);

            return hr;
    }
};
```

클라이언트 객체에 대한 구조체에서 완료 루틴 때와 유사하게 송신용, 수신용 확장 구조체 필드를 구성합니다. 그리고 소켓을 설정하는 생성자 함수에서 확장 구조체의 작업을 FD_XXX 값으로 설정합니다.
멤버 함수로 AsyncSend, AsyncRecv 함수를 포함합니다.

```
// 접속자용 구조체
struct RemoteHost
{
    OVERLAP_EX      olSnd;  // WSASend 함수용
    OVERLAP_EX      olRcv;  // WSARecv 함수용
```

```
…
        RemoteHost(SOCKET s)
        {
                …
                olSnd.dType     = FD_WRITE;
                olRcv.dType     = FD_READ;
        }
…
        INT     AsyncSend(char* s, int l)
        {
                olSnd.Reset();
                olSnd.SetBuf(s, l);
                return olSnd.AsyncSnd(scH);
        }
        INT     AsyncRecv()
        {
                olRcv.Reset();
                return olRcv.AsyncRcv(scH);
        }
};
…
vector<RemoteHost* >    g_vHost;        // Client 리스트
HANDLE  g_hIocp         = NULL;         // IOCP 핸들
```

확장 구조체의 'dType' 이라는 멤버 변수로 송신 요청인지 수신 요청인지 판별할 수가 있습니다. 이전의 서버 코드에 이 변수를 비교해서 작업을 분류하고 입출력에 대한 나머지 과정을 처리합니다. 특히, 수신 요청에 대한 작업이라면 마지막 단계에서 비동기 수신을 다시 요청합니다.

```
// 서버 작업 스레드
DWORD WINAPI WorkThread(void* pParam)
…
while(…)
{
        hr = GetQueuedCompletionStatus(…);

        // IO Failed
        …
```

```
        // 접속 종료(0 == dTran)
        …
        // 송신완료
        if(FD_WRITE == pOlEx->dType)
        {
                // 송신완료 처리
                …
        }
        // 수신완료
        else if(FD_READ == pOlEx->dType)
        {
                …
                // 비동기 수신 요청
                pCln->AsyncRecv();
        }
…
```

<IOCP 채팅 서버: nw63_iocp_chat.zip>

클라이언트는 서버에서 사용한 확장 구조체를 그대로 이용합니다. 작업 스레드도 확장 구조체 변수를 이용해서 송신 또는 수신 요청 작업을 분리하고 이에 대한 나머지 작업을 수행합니다. 수신 요청이면 마지막에 비동기 수신을 요청합니다.

비동기 입출력은 송수신 처리 이외에 나머지 connection, accept 를 처리하기가 어렵습니다. 만약 이들을 작업스레드에서 처리하고자 한다면 작업 스레드에게 메시지를 전달해야 합니다. 이전에 소개한 PostQueuedCompletionStatus () 함수는 이런 상황에 대처하는데 알맞은 함수입니다.

다음 클라이언트 예제는 접속을 성공하고 나서 작업 스레드에게 접속을 알리고 있습니다. Post-QueuedCompletionStatus() 함수의 인수로 사용하는 키에 FD_ACCEPT를 설정하고 작업 스레드에서 키의 값을 FD_ACCEPT로 비교하고 있습니다.
이런 방법은 서버에서도 사용될 수 있으며 작업 스레드에게 뭔가를 요청하고 싶을 때 주저하지 말고 PostQueuedCompletionStatus () 함수를 사용하기 바랍니다.

```
// 채팅 클라이언트
SOCKET          g_scHost = 0;                   // listen socket
OVERLAP_EX      g_olSnd(FD_WRITE);
OVERLAP_EX      g_olRcv(FD_READ);
```

```
HANDLE          g_hIocp  = NULL;                // IOCP Handle

INT     AsyncSend(char* s, int l)               // 데이터 송신
{
        g_olSnd.Reset();
        g_olSnd.SetBuf(s, l);
        return g_olSnd.AsyncSnd(g_scHost);
}

INT     AsyncRecv()                             // 데이터 수신
{
        g_olRcv.Reset();
        return g_olRcv.AsyncRcv(g_scHost);
}

…
int main()
{
        hr = connect(…);

        ULONG_PTR pIoKey = (ULONG_PTR)&g_scHost; // 완료키 = 소켓
        CreateIoCompletionPort(…);              // <소켓, IOCP, key> 바이딩

        // 작업 스레드에게 접속 알림을 설정
        ULONG_PTR       dKey = FD_CONNECT;      // 키를 FD_CONNECT로 설정
        WSAOVERLAPPED   tOl={0};                // dummy
        hr = PostQueuedCompletionStatus(g_hIocp, sizeof(ULONG_PTR), dKey, &tOl);

        // 비동기수신요청
        AsyncRecv();
…


DWORD WINAPI WorkThread(void* pParam)
…
while(g_scHost)
{
```

```
        hr = GetQueuedCompletionStatus(…);

        // IO Failed
        …
        // 접속 종료
        …
        // PostQueuedCompletionStatus() 함수로 보낸 접속 메시지
        if(sizeof(ULONG_PTR) == dTran && FD_CONNECT == pIoKey)
        {
                printf("Connection SuccessedWn");
                continue;
        }

        OLType= pOL->dType;                     // 타입
        if(FD_READ == OLType)                   // receiving complete
        {
                // 수신 작업 처리
                …
                AsyncRecv();                    // 비동기수신요청
        }
        else if(FD_WRITE == OLType)             // Sending complete
        {
                // 송신 작업 처리
        }
…
```

<IOCP: nw63_iocp_chat.zip>