윈도우 네트워크 프로그래밍

남동강

목차

목차

기본 정보

- **종단 시스템(end-system)/호스트(host)**
  - 최종 사용자(end-user)를 위한 **어플리케이션을 수행하는 주체**
  - Ex)인터넷이 연결된 PC, 스마트폰 등등
- **라우터(router)**
  - 종단 시스템에 속한 네트워크와 다른 네트워크를 연결, **서로 다른 네트워크에 속한 종단 시스템끼리 상호 데이터를 교환** 할 수 있도록 하는 **장비**
- **프로토콜**
  - 종단 시스템간 통신 수행을 위한 **정해진 절차와 방법**

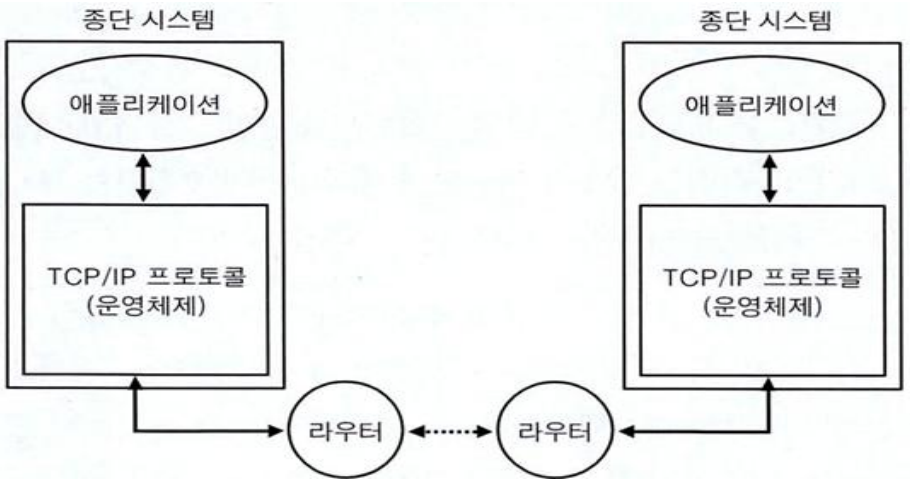
기본 정보

- 프로토콜 TCP와 UDP의 특징 비교

TCP (Transmission Control Protocol)	UDP (User Datagram Protocol)
연결형(connection-oriented) 프로토콜 - 연결에 성공해야 통신이 가능	비연결형(connectionless) 프로토콜 - 연결 없이 통신 가능
데이터 경계를 구분하지 않음 - 바이트 스트림(byte-stream) 서비스	데이터 경계를 구분 -데이터그램(datagram) 서비스
신뢰성 있는 데이터 전송 - 데이터를 재전송함	비신뢰적인 데이터 전송 - 데이터를 재전송하지 않음
1 대 1 통신(unicast)	1 대 1 통신(unicast) 1 대 다 통신(broadcast) 다 대 다 통신(multicast)

TCP/IP

- **TCP/IP 프로토콜**
  - **인터넷의 핵심 프로토콜**이 **TCP와 IP**이며 이를 포함한 **각종 프로토콜**을 TCP/IP 프로토콜이라 한다



TCP/IP 프로토콜을 이용한 통신

TCP/IP

- **TCP/IP 계층(layer) 구조**
  - 일반적으로 프로토콜 계층은 **기능별로** 나뉘어 있다

- **네트워크 액세스 계층(network access layer)**
  - 물리적 네트워크를 통한 실질적인 데이터 전송 담당

- **인터넷 계층(internet layer)**
  - 네트워크 액세스 계층의 도움을 받아, 전송 계층이 내려 보낸 데이터를 종단 시스템(호스트)까지 전달하는 역할
  - 물리적 주소를 사용하지 않고 논리 주소인 IP 주소(internet Protocol address)를 사용
  - 데이터 전송을 위해서는 전송 경로가 필요하기 때문에 **전송 경로를 알아오기 위한 라우팅(routing) 작업이 필요**
  - 라우팅 : 목적지까지 데이터를 보내기 위한 일련의 작업, 전용 컴퓨터를 라우터(router)라 부른다

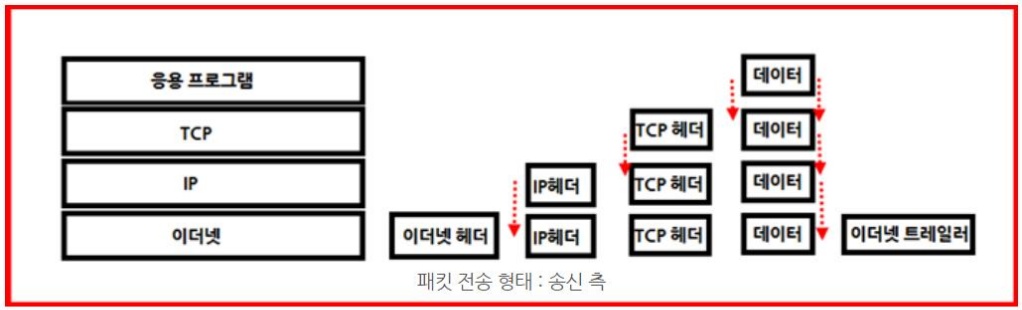
- **전송 계층(transport layer)**
  - 최종적인 통신 목적지를 정하고, **오류 없이 데이터를 전송**하는 역할
  - 해당 **프로세스를 지정하는 일종의 주소**인 **포트 번호(port number)를 사용**
  - TCP(Transmission Control Protocol)과 UDP(User Datagram Protocol)를 사용

- **전송 계층(transport layer)**
  - 최종적인 통신 목적지를 정하고, **오류 없이 데이터를 전송**하는 역할
  - 해당 **프로세스를 지정하는 일종의 주소**인 **포트 번호(port number)를 사용**
  - TCP(Transmission Control Protocol)과 UDP(User Datagram Protocol)를 사용

- **애플리케이션 계층(application layer)**
  - 전송 계층을 기반으로 하는 다수의 프로토콜과 이 프로토콜을 이용하는 응용 프로그램(application)을 포괄한다(Telnet, FTP, HTTP, SMTP…)
  - **소켓을 이용한 네트워크 애플리케이션**도 여기에 속한다

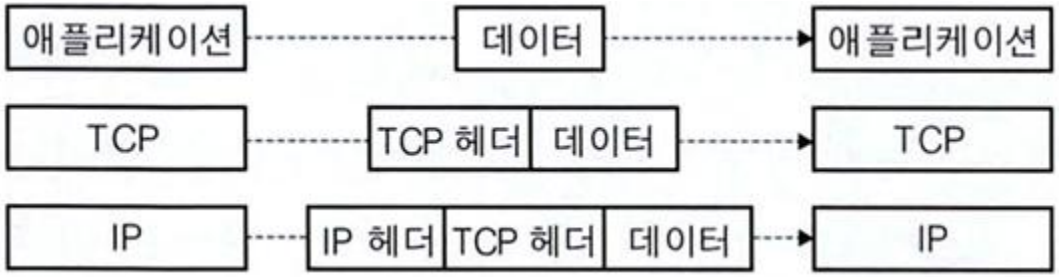
TCP/IP

- **패킷 전송 원리**
- 애플리케이션에서 보내는 데이터를 목적지까지 전송하기 위해서는 각각의 **프로토콜에서 정의한 제어 정보**(*IP주소, 포트 번호, …*)가 필요
- 제어 정보는 위치에 따라 앞에 붙는 **헤더(header)**와 뒤에 붙는 **트레일러(trailer)**로 나누며, 이러한 제어 정보가 결합된 형태의 **실제 전송하는 데이터**를 패킷(packet)이라 부른다
- **패킷(packet) = 제어 정보 + 데이터**

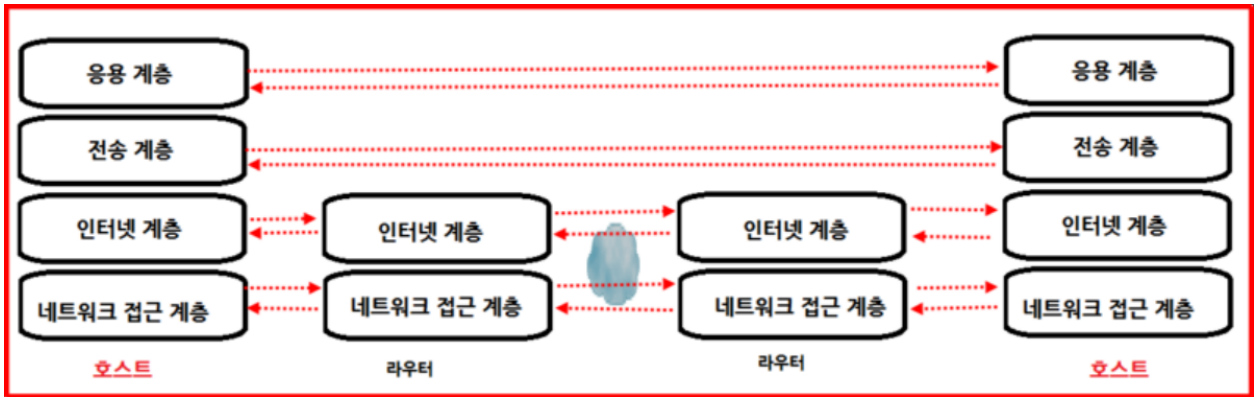


TCP/IP

- 계층 관점에서 본 패킷 전송 형태



- 계층간의 통신



TCP/IP

IP 주소

- IPv4:32비트, 8비트 단위 4부분으로 구분, 10진수로 표기(예)255.255.255.255)
- IPv6:128비트, 16비트 단위 8부분으로 구분, 16진수로 표기 (2020:230:abcd:ffff:0000:0000:ffff:1111)
- 폐쇠된 네트워크이거나 IP를 공유하는 경우가 아니라면 **IP 주소는 전세계적으로 유일한 값**을 가진다
- IPv4형식의 주조가 고갈될 것으로 보여 IPv6형식의 주소가 만들어 졌다

포트 번호

- 전송된 데이터가 **어느 프로세스(process)에서 사용되는지 알기 위한 식별자**
- 16비트의 정수, 0 ~ 65535(2^16개)의 범위를 사용 가능
- 0 ~ 1023은 용도가 정해져 있어 함부로 사용하면 안된다
- 일반적으로 **1024 ~ 49151범위의 값을 사용**한다

포트 번호	분류
0 ~ 1023	Well-known port(유명한, 알려진 서버)
1024 ~ 49151	Registered port(기업 등에서 관리를 위한 포트)
49152 ~ 65535	Dynamic and/or private port(일반 사용자들이 자유롭게 사용 가능한 포트)

소켓(socket)

윈도우 소켓

- 버클리 유닉스(Berkeley Software Distribution UNIX)에서 사용하던 네트워크 인터페이스인(소켓)를 **윈도우 환경에서 사용할 수 있게 만든 것**
- Windows Socket을 줄여 **윈속(winsock)**이라 부른다

소켓(Socket)

- 소프트웨어로 작성된 추상적인 개념의 **통신 접속점**
- 네트워크 애플리케이션은 **소켓을 통하여** 통신망의 **데이터를 송수신** 한다

소켓의 개념을 바라보는 관점

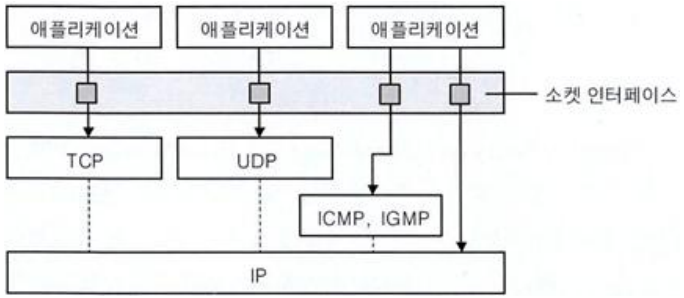
A. 데이터 타입
B. 통신 종단점(communication end-point)
C. 네트워크 프로그래밍 인터페이스

소켓(socket)

- **데이터 타입**
  - 파일 디스크립터(file descriptor) 혹은 핸들(handle)과 유사한 개념(**통신을 위해 관리하는 데이터를 간접적으로 참조할 수 있게 한다**)
  - **파일 입출력과 유사한 형태**를 지녔다
  - 통신과 관련된 다양한 작업을 할 수 있는 간편한 데이터 타입

- **통신 종단점**
  - 통신을 하기 위한 다섯 가지 정보(프로토콜, 송신측 IP 주소, 포트 번호, 수신측 IP주소, 포트번호)의 집합체
  - 클라이언트 소켓이 서버 소켓으로 send()를 호출하여 **데이터를 보내고**, 서버의 소켓이 클라이언트 소켓에서 보낸 데이터를 recv() 함수를 호출하여 **받는다**

- **네트워크 프로그래밍 인터페이스**
  - 하나의 네트워크 프로그래밍 인터페이스
  - **양쪽 모두 소켓을 사용할 필요는 없다**
  - 양쪽 모두 **동일한 프로토콜**을 사용, **정해진 형태와 절차**에 따라 데이터를 주고 받아야 한다
  - 일반적으로 애플리케이션 계층과 전송 계층 사이에 위치하는 것으로 간주하여 전송계층을 건너뛰고 인터넷 계층과 연결하는 것도 가능



윈속 초기화/종료

- **WSAStartup()**
  - 모든 윈속 프로그램에서 소켓 API를 호출하기 전에 반드시 해당 초기화 함수를 호출해야 한다

```
WSADATA wsa;
if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0) return -1;
```

- **WSACleanup()**
  - 윈속 사용 중지함을 운영체제에 알리고 관련 리소스를 반환하는 역할을 한다
  - 함수 호출 실패할 경우 WSAGetLastError() 함수로 알 수 있다

- **새 프로젝트 => 콘솔 응용 프로그램**

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#pragma comment(lib, "ws2_32.lib") // 윈속 라이브러리 사용을 알린다.
#include <WinSock2.h>              // 윈속 사용을 위하여 헤더 파일 추가.
int main()
{
    // 윈속 초기화 (2.2 버전)
    WSADATA wsa;
    if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0) return -1;
    // TODO : 소켓 네트워크 프로그래밍 Code 작성.

    // 윈속 사용이 끝났다.
    WSACleanup();
    return 0;
}
```

오류 처리

- **오류 처리를 할 필요가 없는 경우**
  - 리턴 값이 없거나 호출 시 항상 성공하는 일부 소켓 함수
- **리턴 값만으로 오류를 처리하는 경우**
  - WSAStartup() 함수
- **리턴 값으로 오류 발생을 확인, 구체적인 내용은 오류 코드를 이용하여 확인하는 경우**
  - **대부분의 소켓 함수**
- **WSAGetLastError()**
  - 소켓 함수 호출 결과, 오류가 발생할 경우 해당 함수를 이용하여 **오류 코드**를 얻을 수 있다

```
if (소켓 함수() == 오류)
{
    int error_code = WSAGetLastError();
    std::cout << "error_code에 맞는 오류 메시지 출력";
}
```

오류 처리

- **FormatMessage()**
  - **프로젝트 속성 => 구성 속성 => 고급 => 문자 집합:멀티바이트 문자 집합 사용**
  - WSAGetLastError()에서 얻은 **오류 코드를 오류 메시지로** 자동으로 **변경**시켜 준다

```
FormatMessage(DWORD dwFlags, LPCVOID lpSource, DWORD dwMessageId, DWORD dwLanguageId,
   LPSTR lpBuffer, DWORD nSize, va_list *Arguments);
```

dwFlags : FORMAT_MESSAGE_ALLOCATE_BUFFER (오류 메시지 저장 공간을 함수가 알아서 할당한 다)|
FORMAT_MESSAGE_FROM_SYSTEM(운영체제로부터 오류 메시지를 가져온다)

lpSource : NULL

dwMessageId : **WSAGetLastError()**

dwLanguageId : 오류 메시지를 표시할 언어
MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT) *사용자가 제어판에 설정한 기본 언어를 사용*

lpBuffer : 오류 메시지의 시작 주소가 저장된다.
오류 메시지 사용이 끝나면 **LocalFree()** 함수로 **할당된 메모리를 반환** 하여야 한다

nSize : **0**

Arguments : NULL

오류 처리

```
void err_quit(const char* msg)
{
    LPVOID lpMsgBuf;
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        WSAGetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR)&lpMsgBuf, 0, NULL);

    // msg : 메시지 박스의 타이틀(Caption)
    MessageBox(NULL, (LPCSTR)lpMsgBuf, msg, MB_ICONERROR);

    // 메모리 해제, 핸들 무효화.
    LocalFree(lpMsgBuf);

    exit(-1);
}
```

- **오류 메시지를 메시지** 박스에 **출력** 후 **애플리케이션을 종료** 시킨다

오류 처리

```
void err_display(const char* msg)
{
    LPVOID lpMsgBuf;
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        WSAGetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR)&lpMsgBuf, 0, NULL);

    printf("[%s] %s₩n", msg, (LPCSTR)lpMsgBuf);

    LocalFree(lpMsgBuf);
}
```

- **오류 메시지를 출력**하고 **애플리케이션을 속행**
- **사소한 오류**가 발생할 경우 사용

TCP 서버/클라이언트

- **클라이언트/서버 모델**
  - **두 개의 애플리케이션이 상호 작용**하는 방식을 나타내는 용어
  - **프로세스간 통신(IPC, Inter-Process Communication) 기법**을 이용하여 **상호 정보를 교환**
  - **서로 다른 종단 시스템에서** 실행된 프로세스가 있고 서로 **접속을 하려고 할 때,**
    반드시 서로의 프로세스가 실행 중이어야 하는데 **타이밍의 문제로 접속 실패할 확률이 높아**
    이를 **보안**하기 위한 방법으로 만들어 졌다

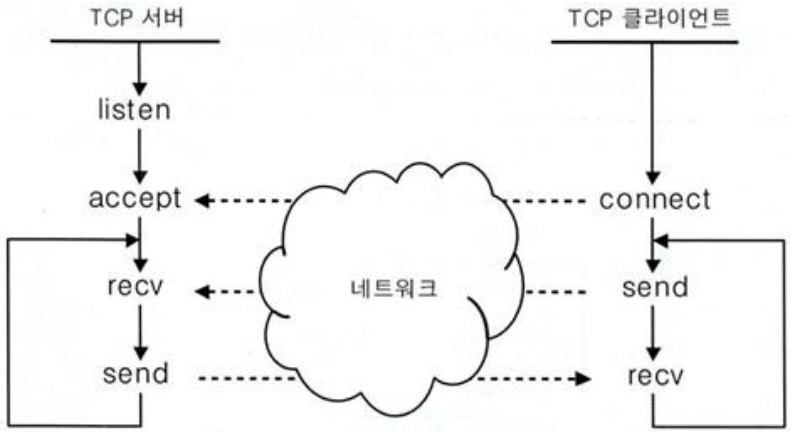
- **서버**
  - 먼저 실행
  - 클라이언트 접속을 대기
  - 클라이언트에서 보내는 패킷에 정보가 있어 **IP 주소(or 도메인 이름)와 포트 번호를 미리 알 필요가 없다**

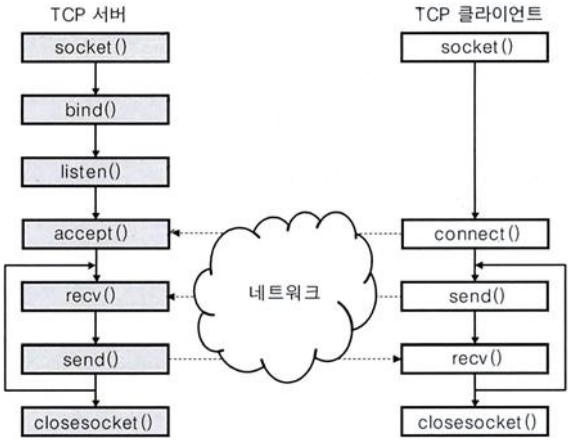
- **클라이언트**
  - 서버보다 나중에 실행
  - 서버에 접속을 시도
  - 서버에 접속하기 위한 **IP 주소(or 도메인 이름)와 포트 번호를 알고있어야 한다**

TCP 서버/클라이언트

- **TCP 서버/클라이언트의 동작 방식**

① 서버 : 먼저 실행되어 **클라이언트 접속을 대기**(listen)
② 클라이언트 : **서버에 접속**(connect)하여 **데이터를 보낸다**(send)
③ 서버 : **클라이언트 접속을 수용**(accept)
④ 서버 : 클라이언트가 보낸 **데이터를 받아**(recv) **처리**
⑤ 서버 : 처리한 **데이터를 클라이언트에 보낸다**(send)
⑥ 클라이언트 : **서버가 보낸 데이터를 받아**(recv) 자신의 **목적에 맞게 사용**





TCP 서버/클라이언트

서버 함수

- **socket()** : **통신을 위한** 소켓을 생성
- **bind()** : 지역 **IP 주소**와 지역 **포트 번호**를 **결정**
- **listen()** : TCP 상태를 LISTENING으로 변경, **접속 받을 준비**
- **accept()** : 접속한 클라이언트와 통신할 수 있는 새로운 소켓(클라이언트) 생성
  원격 IP 주소와 원격 포트 번호가 결정
- **send(), recv()** : 클라이언트와 통신을 수행
- **colsesocket()** : 통신이 끝나면 사용이 끝난 소켓을 닫는다

클라이언트 함수

- **socket()** : **서버 접속을 위한** 소켓을 생성
- **connect()** : 서버에 **접속**
- **send(), recv()** : 서버와 통신을 수행
- **closesocket()** : 서버와의 통신이 끝나면 소켓을 닫는다

- **지역** IP 주소, 포드 번호 : 서버 또는 클라이언트 **자신의 정보**
- **원격** IP 주소, 포트 번호 : 서버 또는 클라이언트가 통신하는 **상대의 정보**

TCP 서버/클라이언트

- **블로킹(blocking) 소켓**
  - 동기 소켓
  - send() 함수를 호출할 때, **송신 버퍼의 여유 공간**이 **보낼 데이터의 크기보다 작을 경우** 해당 프로세스를 **대기 상태(wait state)**로 만들고 송신 버퍼의 여유 공간이 생기면 깨어나 크기만큼 데이터 복사가 이루어 진다

- **논블로킹(nonblocking) 소켓**
  - 비동기 소켓
  - ioctlsocket() 함수를 이용하여 블로킹 소켓을 논블로킹 소켓으로 변경할 수 있다
  - send() 함수가 호출되면 **송신 버퍼의 여유 공간 만큼 데이터 복사** 후 실제 복사된 바이트 수를 리턴한다
  - 프로그램이 복잡해 지며, CPU 사용량이 증가한다

TCP 서버/클라이언트

- SOCKET socket(int af, int type, int protocol)
  - **af** : 주소 체계를 지정, AF_INET(internetwork : **TCP**, UDP, etc를 사용, IPv4)
  - **type** : 소켓 타입 지정, SOCK_STREAM 또는 SOCK_DGRAM 사용
  - **protocol** : 사용할 프로토콜 지정, **프로토콜 결정**에 **모호함이 없다면 0**을 사용

소켓 타입	특성
SOCK_STREAM	**신뢰**성 있는 데이터 전송 제공, **연결형** 프로토콜
SOCK_DGRAM	**비신뢰**적인 데이터 전송 기능 제공, **비연결형** 프로토콜

사용할 프로토콜	주소 체계	소켓 타입
TCP	AF_INET	SOCK_STREAM
UDP	AF_INET	SOCK_DGRAM

- int closesocket(SOCKET s)
  - 소켓을 닫고 관련 리소스를 반환

TCP 서버/클라이언트

- **TCP/IP 프로토콜(IPv4) 구조체**

sin_family : 주소 체계
sin_port : 포트 번호
sin_addr : IP 주소, in_addr 구조체
sin_zero : 사용되지 않는다, **0**.

```
struct SOCKADDR_IN {
 unsigned short sin_family; // 2 바이트.
 unsigned short sin_port;   // 2 바이트.
 IN_ADDR sin_addr;          // 4 바이트. IPv4 Internet address 구조체.
 char sin_zero[8];          // 8 바이트.
} // 16 바이트.
```

- **바이트 정렬** : 메모리 데이터를 저장할 때의 바이트 순서

- ***의도하지 않은 프로세스로 데이터 전달될 위험 방지***
- **빅 엔디안(big-endian)** : 최상위 바이트(Most Significant Byte)부터 차례로 정렬
- **리틀 엔디안(little-endian)** : 최하위 바이트(Least Significant Byte)부터 차례로 정렬
- **IP 주소**, **포트 번호**는 **빅 엔디안**을 사용
- **네트워크 바이트 정렬(network byte ordering)** : **빅 엔디안**을 사용
- **호스트 바이트 정렬(host byte ordering)** : 시스템 내의 고유한 바이트 정렬

```
u_short htons( u_short hostshort); // host to network short
u_long htonl(u_long hostlong);     // host to net work long
u_short ntohs(u_short hostshort);
u_long ntohl(u_long hostlong);
```

Blocking(Sync) Socket

- **TCP/IP Server 새 콘솔 프로젝트 생성**

```
int main()
{
    // 윈속 초기화.
    …

    // socket() listen_socket 생성.
    SOCKET listen_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (INVALID_SOCKET == listen_socket) err_quit("socket");

    //  계속 작업 진행…

    // closesocket(listen_socket)
    closesocket(listen_socket);

    // 윈속 사용이 끝났다.
    …
}
```

Blocking(Sync) Socket

```
int main()
{
    // 윈속 초기화.
    …
    // socket() listen_socket 생성.
    …

    // 서버 정보 객체 설정.
    // INADDR_ANY : 자신의 IP를 알아와 주는 메크로 함수.
    SOCKADDR_IN serveraddr;
    ZeroMemory(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(9000);
    serveraddr.sin_addr.S_un.S_addr = htonl(INADDR_ANY);

    // closesocket(listen_socket)
    …
    // 윈속 사용이 끝났다.
    …
}
```

Blocking(Sync) Socket

```
int main()
{
    ...

   // bind() 소켓 설정.
    if (bind(listen_socket, (SOCKADDR*)&serveraddr, sizeof(serveraddr)) == SOCKET_ERROR)
    {
        closesocket(listen_socket);
        WSACleanup();
       err_quit("bind");
    }

    // listen() 수신 대기열 생성.
    if (listen(listen_socket, SOMAXCONN) == SOCKET_ERROR)
    {
        closesocket(listen_socket);
        WSACleanup();
       err_quit("listen");
    }

    ...
}
```

Blocking(Sync) Socket

```
#define MAX_BUFFER_SIZE 256
int main()
{
      …
    // 데이터 통신에 사용할 변수.
    SOCKADDR_IN clientaddr;
    int addrlen = sizeof(SOCKADDR_IN);
    ZeroMemory(&clientaddr, addrlen);

    SOCKET client_socket;
    int retval;
    char buf[MAX_BUFFER_SIZE + 1];
    while (1) {
        // accept() 연결 대기.
        client_socket = accept(listen_socket, (SOCKADDR*)&clientaddr, &addrlen);
        if (INVALID_SOCKET == client_socket) continue;

        printf("Wn[TCP 서버] 클라이언트 접속 : IP 주소=%s, 포트 번호=%dWn ", inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port));

        // recv()
        // send()

        // 클라이언트 소켓 종료.
        closesocket(client_socket);
        printf("Wn[TCP 서버] 클라이언트 종료 : IP 주소=%s, 포트 번호=%dWn ", inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port));

    }
      …
}
```

Blocking(Sync) Socket

```
int main()
{
      ...
     while (1)
     {
          ...
          // 데이터 통신.
          while (1)
          {
               ZeroMemory(buf, sizeof(buf));
               // 데이터 받기.
               retval = recv(client_socket, buf, sizeof(buf), 0);
               if (SOCKET_ERROR == retval) break;
               else if (0 == retval) break;

               // 받은 데이터 출력.
               buf[retval - 1] = '₩0';
               printf("₩n[TCP/%s:%d] %s₩n ", inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port), buf);

               // 데이터 보내기.
               retval = send(client_socket, buf, retval, 0);
               if (SOCKET_ERROR == retval) break;
          }
          ...
     }
      ...
}
```

Blocking(Sync) Socket

- **TCP/IP Client 새 콘솔 프로젝트 생성**

```
int main()
{
   // 윈속 초기화.
   …

   // socket() 소켓 생성.
   SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
   if (INVALID_SOCKET == sock) return -1;

   // 작업 진행…

   closesocket(sock);

   // 윈속 사용이 끝났다.
   WSACleanup();
}
```

Blocking(Sync) Socket

```
int main()
{
    ...

    // 서버 정보 객체 설정.
    // 127.0.0.1 : 루프백(Loopback)IP
    // IPv4 및 IPv6에서 자기 자신을 가리키기 위한 목적으로 쓰기 위해 예약된 IP 주소.
    SOCKADDR_IN serveraddr;
    ZeroMemory(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(9000);
    serveraddr.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");

    // connect()
    if (SOCKET_ERROR == connect(sock, (SOCKADDR*)&serveraddr, sizeof(serveraddr)))
    err_quit("connect");

    ...
}
```

Blocking(Sync) Socket

```
#define MAX_BUFFER_SIZE 256
int main()
{
    ...
    int len, retval;
    char buf[MAX_BUFFER_SIZE + 1];
    while (1)
    {
        ZeroMemory(buf, sizeof(buf));
        printf("₩n[보낼 데이터]");
        if (fgets(buf, MAX_BUFFER_SIZE, stdin) == NULL) break;

        len = strlen(buf);
        if (buf[len - 1] == '₩n') buf[len - 1] = '₩0';
        if (strlen(buf) == 0) break;

        // 데이터 보내기.
        retval = send(sock, buf, len, 0);
        if (SOCKET_ERROR == retval) break;

       printf("[TCP 클라이언트] %d바이트를 보냈습니다.₩n", retval);
```

Blocking(Sync) Socket

```
        // 데이터 받기.
        ZeroMemory(buf, sizeof(buf));
        retval = recv(sock, buf, sizeof(buf), 0);
        if (SOCKET_ERROR == retval) break;
        else if (0 == retval) break;

        // 받은 데이터 출력.
        buf[retval - 1] = '₩0';
        printf("[TCP 클라이언트] %d바이트를 받았습니다.₩n", retval);
        printf("[받은 데이터]%s₩n", buf);
    }
  ...
}
```

멀티스레드(multi thread)

- **스레드**
  - 실제 CPU 시간을 할당 받아 수행되는 실행단위
- **스레드 함수(thread function)**
  - 스레드 실행 시작점이 되는 함수
  - 예) main()
- **주 스레드(primary thread)**
  - **main(), WinMain()**에서 시작되는 스레드
  - 프로세스가 실행될 때 생성
- **컨텍스트 전환(context switch)**
  - **CPU와 운영체제**의 협동으로 이루어지는 **스레드 실행 상태의 저장과 복원 작업**
  - 각 스레드는 **다른 스레드의 존재와 무관하게 자신의 상태를 유지**하며 실행 가능

❖ **스레드가 많아지면 CPU에 무리를 주기 때문에 사용에 주의!**

멀티스레드(multi thread)

- **스레드 생성**
  - 스레드 생성 함수 : CreateThread(), _beginthread(), _beginthreadex()
  - 스레드 생성시에 CreateThread()를 쓰지 않는 것이 좋다
    - (c/c++ 런타임 함수를 사용할경우, 함수는 실행되지 않고 스레드만 죽는다)

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

lpThreadAttributes : 구조체 변수의 주소값, NULL을 사용
dwStackSize : 새로 생성할 스레드에 할당될 스택의 크기, **0 사용 시 기본 1MB가 할당**
lpStartAddress : 함수 스레드의 시작 주소, DWORD WINAPI ThreadProc(LPVOID lpParameter) {}
lpParameter : 스레드 함수에 전달될 인자, 전달할 인자가 없다면 NULL
dwCreationFlags : 스레드 생성을 제어하는 값, 사용하지 않으면 NULL
lpThreadId : **스레드의 ID**가 저장, 필요 없다면 NULL(Dos에서는 사용 하면 안됨)

멀티스레드(multi thread)

- TCP 서버

```
SOCKET client_socket;
HANDLE hThread;
DWORD threadID;
while (1)
{
    // accept() 연결 대기.
    client_socket = accept(listen_socket, (SOCKADDR*)&clientaddr, &addrlen);
    if (INVALID_SOCKET == client_socket) continue;

    printf("₩n[TCP 서버] 클라이언트 접속 : IP 주소=%s, 포트 번호=%d₩n",
        inet_ntoa(clientaddr.sin_addr),
        ntohs(clientaddr.sin_port));

    // 스레드 생성.
    hThread = CreateThread(NULL, 0, ProcessClient, (LPVOID)client_socket, 0, &threadID);
    if (NULL == hThread) std::cout << "[오류] 스레드 생성 실패!" << std::endl;
    else CloseHandle(hThread);
}
```

멀티스레드(multi thread)

```
DWORD WINAPI ProcessClient(LPVOID arg)
{
    SOCKET client_sock = (SOCKET)arg;
    SOCKADDR_IN clientaddr;
    int addrlen = sizeof(clientaddr);
    getpeername(client_sock, (SOCKADDR*)&clientaddr, &addrlen); // 클라이언트 정보 얻기.
    int retval;
    while (1)
    {
       // recv(), send()
       ...
    }

    // closesocket()
    closesocket(client_sock);
    printf("₩n[TCP 서버] 클라이언트 종료 : IP 주소=%s, 포트 번호=%d₩n",
        inet_ntoa(clientaddr.sin_addr),
        ntohs(clientaddr.sin_port));
    return 0;
}
```

멀티스레드(multi thread)

스레드 종료

- **정상적인 종료 방법**
  - **스레드 함수에서 리턴**한다
  - 스레드 함수 내에서 **ExitThread()** 함수를 호출
- **강제적인 종료 방법**
  - **TerminateThread()** 함수를 호출
  - **주 스레드가 종료**되면서 모든 스레드가 종료된다

소켓 정보 가져오기

- `Int getpeername(SOCKET s, struct sockaddr* name, int* namelen)`
  - 소켓 데이터 구조체에 저장된 **원격** IP 주소와 포트 번호를 알려준다
- `Int getsockname(SOCKET s, struct sockaddr* name, int* namelen)`
  - 소켓 데이터 구조체에 저장된 **지역** IP 주소와 포트 번호를 알려준다

멀티스레드(multi thread)

- **공유 리소스 만들기**
  - 접속한 모든 클라이언트에 데이터 전송을 위한 공유 리소스
  - 접속되어 있는 모든 클라이언트에 데이터를 해주기 위한 스레드 공유 리소스 필요

- **TCP Server**

```
#include <vector>
#include <algorithm>
std::vector<SOCKET> g_vSocket; // 전역 변수.
int main()
{
    ...
    while (1)
    {
        // accept() 연결 대기.
        client_socket = accept(listen_socket, (SOCKADDR*)&clientaddr, &addrlen);
        if (INVALID_SOCKET == client_socket) continue;

        g_vSocket.push_back(client_socket); // 접속한 클라이언트의 소켓 vector에 보관.
        ...
        // 스레드 생성.
        ...
    }
}
```

멀티스레드(multi thread)

```
DWORD WINAPI ProcessClient(LPVOID arg)
{
    ...
    while (1)
    {
        ...
        for (const auto& sock : g_vSocket) // 접속한 모든 클라이언트에 데이터 전송.
        {
            // 데이터 보내기.
            ...
            retval = send(sock, buf, retval, 0); // 수정.
            ...
        }
    }

    // vector에서 접속을 종료한 socket 제거.
    auto itr = std::find(g_vSocket.begin(), g_vSocket.end(), client_sock);
    if(g_vSocket.end() != itr) g_vSocket.erase(itr);

    // closesocket()
    ...
}
```

스레드 동기화

스레드 동기화(thread synchronization)란?

- 멀티스레드 환경에서 발생하는 문제를 해결하기 위한 일련의 작업
- 하나의 자원을 한 번에 하나의 스레드만 사용하도록 하는 기술

필요한 경우

- 두 개 이상의 스레드가 **공유 리소스를 접근할 때**, 오직 한 스레드만이 접근을 허용해야 하는 경우
- 특정 사건 발생을 다른 스레드에 알리는 경우

  ex)한 스레드가 작업을 완료 후, 대기 중인 다른 스레드를 깨우는 경우

동기화 객체(synchronization object)

- 하나의 자원을 사용함에 있어 진행을 계속하거나 대기 하도록 하는 **매개체 역할을 할 수 있는 것들**

종류	주요 용도
임계 영역(critical section)	공유 리소스에 대해 오직 하나의 스레드만 접근 허용 (*한 프로세스에 속한 스레드에만 사용 가능*) *일반적인 동기화 객체보다 빠르고 효율적, 동기화 객체로 분류 X*
뮤텍스(mutex)	공유 리소스에 대해 오직 하나의 스레드만 접근 허용 (*서로 다른 프로세스에 속한 스레드에도 사용 가능*)
이벤트(event)	특정 사건 발생을 다른 스레드에 알린다
세마포어(semaphore)	한정된 개수의 자원을 여러 스레드가 사용하려 할 때 접근 제어
대기 기능 타이머(waitable timer)	특정 시간이 되면 대기 중인 스레드를 깨운다

스레드 동기화

- **뮤텍스**
  - 하나의 프로세스에서 여러 스레드를 사용할 때 사용한다
  - 어떠한 데이터를 사용하고 있는 동안 다른 스레드는 이 데이터를 건드리지 못하게 한다
  - 데이터를 먼저 사용하고 있는 다른 스레드에서 사용이 끝날 때 까지 대기

```
HANDLE g_hMutex; // 뮤텍스 핸들.
// 뮤텍스를 생성.
// NULL:하위 프로세스에 상속 불가.
// false:해당 뮤텍스의 권한을 호출한 스레드가 가지지 못하게 한다.
// NULL:해당 뮤텍스의 이름. 이름 없이 작성.
g_hMutex = CreateMutex(NULL, false, NULL);
// 공용 리소스를 사용,  임의의 시간(INFINITE:작업이 끝날 때 까지)동안 대기.
WaitForSingleObject(g_hMutex, INFINITE);
ReleaseMutex(g_hMutex); // 공용 리소스 사용이 끝났다.
CloseHandle(g_hMutex); // 뮤텍스 사용을 끝내고 제거.
```

스레드 동기화

```
HANDLE g_hMutex; // 전역 변수.
int main()
{
    g_hMutex = CreateMutex(NULL, false, NULL);
    if (NULL == g_hMutex) return -1; // 뮤텍스 생성 실패.
    if (GetLastError() == ERROR_ALREADY_EXISTS) // 이미 생성된 뮤텍스가 있습니다!!
    {
            CloseHandle(g_hMutex);
            return -1;
    }

    ...

    // 공용 리소스를 다른 스레드에서 사용하고 있다면 대기,
    WaitForSingleObject(g_hMutex, INFINITE);
    // 다른 스레드에서도 사용하는 공용 리소스를 사용한다.
    ...
    //공용 리소스 사용이 끝났음을 알린다.
    ReleaseMutex(g_hMutex);

    ...

    CloseHandle(g_hMutex);
}
```

Non-blocking(Async) Socket

- **TCP/IP Server 새 콘솔 프로젝트 생성**

```
void err_display(const int& errcode)
{
    LPVOID lpMsgBuf;
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
        NULL, errcode,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR)&lpMsgBuf, 0, NULL);

    printf("[오류] %s", (LPCTSTR)lpMsgBuf);

    LocalFree(lpMsgBuf);
}
```

Non-blocking(Async) Socket

```
#define BUFSIZE 256
struct SOCKETINFO
{
    SOCKET sock;
    char buf[BUFSIZE + 1];
};

// listen 소켓을 포함한, 현재 접속한 client의 수.
int g_nTotalSockets = 0;
// 논블로킹 소켓으로 만든 소켓의 정보.
SOCKETINFO* arrSocketInfo[WSA_MAXIMUM_WAIT_EVENTS];
// 소켓이 어떤 처리를 할 수 있는지에 대한 이벤트.
WSAEVENT arrEvent[WSA_MAXIMUM_WAIT_EVENTS];

// 논블로킹 소켓으로 사용할 소켓 정보 추가.
bool AddSocketInfo(SOCKET sock);
// 소켓 정보 삭제.
void RemoveSocketInfo(int index);
```

Non-blocking(Async) Socket

```
bool AddSocketInfo(SOCKET sock)
{
    if (WSA_MAXIMUM_WAIT_EVENTS <= g_nTotalSockets) {
        printf("[오류] 소켓 정보를 추가할 수 없습니다!₩n");
        return false;
    }

    SOCKETINFO* ptr = new SOCKETINFO;
    if (NULL == ptr) {
        printf("[오류] 메모리가 부족합니다!₩n");
        return false;
    }

    WSAEVENT hEvent = WSACreateEvent();
    if (WSA_INVALID_EVENT == hEvent) {
        err_display("WSACreateEvent()");
        return false;
    }

    ptr->sock = sock;
    arrSocketInfo[g_nTotalSockets] = ptr;
    arrEvent[g_nTotalSockets] = hEvent;
    g_nTotalSockets++;

    return true;
}
```

Non-blocking(Async) Socket

```
void RemoveSocketInfo(int index)
{
    SOCKETINFO* ptr = arrSocketInfo[index];

    // 클라이언트 정보 얻기.
    SOCKADDR_IN addr;
    int addlen = sizeof(addr);
    getpeername(ptr->sock, (SOCKADDR*)&addr, &addlen);
    printf("[TCP 서버] 클라이언트 종료 : IP 주소=%s, 포트 번호=%d₩n", inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));

    closesocket(ptr->sock);
    delete ptr;
    WSACloseEvent(arrEvent[index]);

    // 배열 공간을 한 칸씩, 앞으로 이동.
    for (int i = index + 1; g_nTotalSockets > i; i++)
    {
        arrSocketInfo[i - 1] = arrSocketInfo[i];
        arrSocketInfo[i] = nullptr;

        arrEvent[i - 1] = arrEvent[i];
        arrEvent[i] = 0;
    }
    g_nTotalSockets--;
}
```

Non-blocking(Async) Socket

```
int main()
{
    // 윈속 초기화.
    ...
    // socket() listen_socket 생성.
    SOCKET listen_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (INVALID_SOCKET == listen_socket) err_quit("socket");

    // 소켓 정보 추가.
    if (FALSE == AddSocketInfo(listen_socket)) return -1;

    // WSAEventSelect() : 콘솔 프로젝트에서 사용, 해당 소켓을 논브롤킹 소켓으로 만들어 준다.
    // FD_ACCEPT : 클라이언트에서 연결 요청이 왔을 때.
    // FD_CONNECT : connect()를 이용하여 서버에서 연결을 수락 했을 때.
    // FD_CLOSE : 해당 소켓이 연결을 해제 했을 때.
    // FD_READ : 소켓을 읽어 들일 데이터가 있을 때.
    // FD_WRITE : send()할 수 있을 때, 소켓의 버퍼가 비어있을 때.
    // listen_socket은 accept()와 closesocket()을 해야할 경우만 이벤트를 받도록 한다.
    int retval = WSAEventSelect(listen_socket, arrEvent[g_nTotalSockets - 1], FD_ACCEPT | FD_CLOSE);
    if (SOCKET_ERROR == retval)
    {
        closesocket(listen_socket);
        err_quit("WSAEventSelect()");
    }
```

Non-blocking(Async) Socket

```
    // bind()
    …
    // listen()
    …

    int index;
    WSANETWORKEVENTS NetworkEvents;
    SOCKET client_sock;
    SOCKADDR_IN clientaddr;
    int addlen = sizeof(SOCKADDR_IN);
    while (true)
    {
        // 아래 페이지에서 작성.
    }

    // closesocket(listen_socket)
    closesocket(listen_socket);
    // 윈속 사용이 끝났다.
    …
}
```

Non-blocking(Async) Socket

```
// TODO: 여기서부터 while문 안의 내용.
// 현재 페이지의 내용은 반드시 while문 제일 첫 줄에, 순서에 맞게 작성 되어야 한다.
// 이벤트 객체 관찰.
// 몇 번째 WSAEVENT배열에 이벤트가 발행했는지 index를 알 수 있다.
index = WSAWaitForMultipleEvents(g_nTotalSockets, arrEvent, FALSE, WSA_INFINITE, FALSE);
if (WSA_WAIT_FAILED == index)
{
    err_display("WSAWaitForMultipleEvents()");
    continue;
}
// 반환 값에 WSA_WAIT_EVENT_0를 뺀 값이 배열의 index
index -= WSA_WAIT_EVENT_0;

// 구체적인 네트워크 이벤트 알아내기.
// 해당 소켓에 발행한 이벤트를 WSANETWORKEVENTS 객체를 통해 알아온다.
retval = WSAEnumNetworkEvents(arrSocketInfo[index]->sock, arrEvent[index], &NetworkEvents);
if (SOCKET_ERROR == retval)
{
    err_display("WSAEnumNetworkEvents()");
    continue;
}
```

Non-blocking(Async) Socket

```
// FD_ACCEPT 이벤트 처리.
if (NetworkEvents.lNetworkEvents & FD_ACCEPT)
{
    // FD_ACCEPT 이벤트에 발생한 오류를 확인.
    if (0 != NetworkEvents.iErrorCode[FD_ACCEPT_BIT])
    {
        err_display(NetworkEvents.iErrorCode[FD_ACCEPT_BIT]);
        continue;
    }

    // accept()
    client_sock = accept(arrSocketInfo[index]->sock, (SOCKADDR*)&clientaddr, &addlen);
    if (INVALID_SOCKET == client_sock)
    {
        err_display("accept()");
        continue;
    }
    printf("[TCP 서버] 클라이언트 접속 : IP 주소=%s, 포트 번호=%d₩n", inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port));

    // 접속 수 제한.
    if (WSA_MAXIMUM_WAIT_EVENTS <= g_nTotalSockets)
    {
        printf("[오류] 더 이상 접속을 받아들일 수 없습니다!₩n");
        closesocket(client_sock);
        continue;
    }
```

Non-blocking(Async) Socket

```
    // 클라이언트 소켓 정보 추가.
    if (!AddSocketInfo(client_sock)) continue;

    // WSAEventSelect()
    // client_sock은 클라이언트에서 send()하여 recv()해야 할 경우와 클라이언트가 종료되어 closesocket()을 해야할 경우만 이벤트를 받도록 한다.
    retval = WSAEventSelect(client_sock, arrEvent[g_nTotalSockets - 1], FD_READ | FD_CLOSE);
    if (SOCKET_ERROR == retval) err_quit("WSAEventSelect()");
} // FD_ACCEPT 이벤트.

// FD_READ 이벤트 처리.
if (NetworkEvents.lNetworkEvents & FD_READ)
{
    if (0 != NetworkEvents.iErrorCode[FD_READ_BIT])
    {
        err_display(NetworkEvents.iErrorCode[FD_READ_BIT]);
        continue;
    }

    SOCKETINFO* ptr = arrSocketInfo[index];
    retval = recv(ptr->sock, ptr->buf, BUFSIZE, 0);
    if (SOCKET_ERROR == retval)
    {
        if (WSAEWOULDBLOCK != WSAGetLastError())
        {
            err_display("recv()");
            RemoveSocketInfo(index);
        }
        continue;
    }
```

Non-blocking(Async) Socket

```
        ptr->buf[retval] = '₩0';
        getpeername(ptr->sock, (SOCKADDR*)&clientaddr, &addlen);
        printf("[TCP/%s:%d]%s₩n", inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port), ptr->buf);

        retval = send(ptr->sock, ptr->buf, retval, 0);
        if (SOCKET_ERROR == retval)
        {
            if (WSAEWOULDBLOCK != WSAGetLastError())
            {
                err_display("send()");
                RemoveSocketInfo(index);
            }
            continue;
        }
    } // FD_READ 이벤트.

    // FD_CLOSE 이벤트 처리.
    if (NetworkEvents.lNetworkEvents & FD_CLOSE)
    {
        if (0 != NetworkEvents.iErrorCode[FD_CLOSE_BIT])
        {
            err_display(NetworkEvents.iErrorCode[FD_CLOSE_BIT]);
        }
        RemoveSocketInfo(index);
    }
```

Non-blocking(Async) Socket

- **TCP/IP Client 새 데스크톱 어플리케이션 프로젝트 생성**
  - [**문자 집합:멀티바이트 문자 집합 사용**]으로 변경

```
// WM_USER 초과한 값을 사용하여 개인적인 윈도우 메시지를 만들어 사용할 수 있다.
#define WM_SOCKET (WM_USER + 1)
#define BUFSIZE 256

SOCKET g_sock;
std::string g_strChat;
std::vector<std::string> g_vLog;

void AddLog(const char* str);
void ErrorQuit(const char* caption);
void ErrorDisplay(HWND hWnd, const char* caption);
void ErrorDisplay(HWND hWnd, const int& errorCode, const char* caption);
void SocketProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);
```

Non-blocking(Async) Socket

```
void AddLog(const char* str)
{
    if (26 <= g_vLog.size()) g_vLog.erase(g_vLog.begin());
    g_vLog.push_back(str);
}

void ErrorQuit(const char* caption)
{
    LPVOID lpMsgBuf;
    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, WSAGetLastError(), MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
     (LPTSTR)&lpMsgBuf, 0, NULL);
    MessageBox(NULL, (LPCTSTR)lpMsgBuf, caption, MB_ICONERROR);
    LocalFree(lpMsgBuf);
    exit(-1);
}

void ErrorDisplay(HWND hWnd, const char* caption) { ErrorDisplay(hWnd, WSAGetLastError(), caption); }

void ErrorDisplay(HWND hWnd, const int& errorCode, const char* caption)
{
    LPVOID lpMsgBuf;
    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, errorCode, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpMsgBuf,
     0, NULL);
    char buf[256];
    ZeroMemory(&buf, sizeof(buf));
    sprintf_s(buf, "[%s] %s", caption, (LPCTSTR)lpMsgBuf);
    AddLog(buf);
    InvalidateRect(hWnd, NULL, true);
    LocalFree(lpMsgBuf);
}
```

Non-blocking(Async) Socket

```
int APIENTRY wWinMain(_In_ HINSTANCE hInstance,
                      _In_opt_ HINSTANCE hPrevInstance,
                      _In_ LPWSTR    lpCmdLine,
                      _In_ int       nCmdShow)
{
    ...
    //if (!InitInstance(hInstance, nCmdShow))
    //{
    //    return FALSE;
    //}
    HWND hWnd = CreateWindowW(szWindowClass, szTitle, WS_SYSMENU, CW_USEDEFAULT, 0, 800, 600, nullptr, nullptr,
    hInstance, nullptr);
    if (!hWnd) return FALSE;

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    // 윈속 초기화.
    WSADATA wsa;
    if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0) return -1;

    // 서버 접속을 위한 소켓 생성.
    g_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (INVALID_SOCKET == g_sock) ErrorQuit("socket()");
```

Non-blocking(Async) Socket

```
    // WSAAsyncSelect() : 데스크톱 응용프로그램에서 사용, 해당 소켓을 논브롤킹(비동기) 소켓으로 만들어 준다.
    int retval = WSAAsyncSelect(g_sock, hWnd, WM_SOCKET, FD_CONNECT);
    if (SOCKET_ERROR == retval)
    {
        closesocket(g_sock);
        ErrorQuit("WSAAsyncSelect()");
    };

    // 서버 접속을 위한, 서버 주소 설정.
    SOCKADDR_IN serveraddr;
    ZeroMemory(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(9000);
    serveraddr.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
    // 비동기 소켓으로 WM_SOCKET에서 오류 및 FD_CONNECT 메시지 처리, 여기서 오류 처리 X.
    connect(g_sock, (SOCKADDR*)&serveraddr, sizeof(serveraddr));

    MSG msg;
    // 기본 메시지 루프입니다:
    ...
    // 윈속 종료.
    WSACleanup();

    return (int) msg.wParam;
}
```

Non-blocking(Async) Socket

```
void SocketProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    if (WSAGETSELECTERROR(lParam)) // 요류 발생 여부 확인.
    {
        ErrorDisplay(hWnd, WSAGETSELECTERROR(lParam), "ERROR");
        closesocket(g_sock);
        return;
    }
    std::string msg;
    switch (WSAGETSELECTEVENT(lParam))
    {
    case FD_CONNECT: msg = "FD_CONNECT";
        if (SOCKET_ERROR == WSAAsyncSelect(g_sock, hWnd, WM_SOCKET, FD_CLOSE | FD_READ))
        {
            closesocket(g_sock);
            ErrorQuit("WSAAsyncSelect()");
        };
        break;
    case FD_CLOSE: msg = "FD_CLOSE : 서버와 연결이 끊겼습니다."; closesocket(g_sock); break;
    case FD_READ: { msg = "FD_READ : ";
        char buf[BUFSIZE];
        int retval = recv(g_sock, buf, BUFSIZE, 0);
        msg.append(buf);
    } break;
    }
    msg.append("WO");
    AddLog(msg.c_str());
    InvalidateRect(hWnd, NULL, true);
}
```

Non-blocking(Async) Socket

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_SOCKET: SocketProc(hWnd, message, wParam, lParam); break;
    case WM_CHAR:
    {
        const char& ch = (TCHAR)wParam;
        if ('₩n' != ch && '₩r' != ch)
        {
            if ('₩b' == ch)
            {
                if (0 < g_strChat.length()) g_strChat = g_strChat.substr(0, g_strChat.length() - 1);
            }
            else  g_strChat += ch;
            InvalidateRect(hWnd, NULL, true);
        }
    } break;
    case WM_KEYUP:
        if (VK_RETURN == wParam)
        {
            if (0 < g_strChat.length())
            {
                char buf[BUFSIZE + 1];
                strcpy_s(buf, g_strChat.c_str());
                send(g_sock, buf, g_strChat.length(), 0);
                g_strChat.clear();
            }
        } break;
```

Non-blocking(Async) Socket

```
case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hWnd, &ps);
        // TODO: 여기에 hdc를 사용하는 그리기 코드를 추가합니다...

        // 발생한 socket 이벤트의 로그 출력.
        for (int i = 0; g_vLog.size() > i; i++)
            TextOut(hdc, 0, i * 20, g_vLog[i].c_str(), g_vLog[i].length());

        // 입력한 문자 출력.
        TextOut(hdc, 0, 540, g_strChat.c_str(), g_strChat.length());

        EndPaint(hWnd, &ps);
    }
    break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        // 멀티바이트 문자 집합 사용으로 DefWindowProc() => DefWindowProcW() 변경.
        return DefWindowProcW(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

IOCP

- ## I/O
  - **입력**(Input)/**출력**(Output)의 약자로, **컴퓨터 및 주변장치에 대하여 데이터를 전송**하는 프로그램 혹은 장치

- ## IOCP(I/O Completion Port)
  - **Windows I/O 모델 중 최고의 성능**을 가진다
  - 특수한 경우, 다른 모델이 IOCP 모델 보다 더 높은 성능을 보이기도 한다
  - 멀티쓰레드에서 동작
  - IOCP 객체는 여러 소켓을 DEVICE LIST에 등록/관리하는 자료구조
  - DEVICE LIST는 유저가 접근/제어 할 수 없다
  - IOCP는 **무조건 OVERLAPPED I/O로만 동작**
  - OVERLAPPED I/O Callback을 사용

- ## OVERLAPPED
  - **비동기(Async)/비봉쇄(non-blocking)** 응용 **모델**
  - OVERLAPPED I/O 중첩을 허용하는 특징이 있어, "중첩 입출력 모델"이라고 부르기도 한다

출처 : 링크

IOCP

- **IOCP Server 새 콘솔 프로젝트 생성**

```
#include <map>
#include <vector>
#include <process.h>
#include <mutex>

enum class OverlappedType { Invalid = -1, Recv, Send, };
struct Overlapped
{
    WSAOVERLAPPED wsaOverlapped;
    WSABUF wsaBuf;
    OverlappedType type;
};
// WSAOVERLAPPED 객체의 포인터를 이용하여 ClientInfo 객체를 얻기 때문에,
// 반드시 WSAOVERLAPPED는 가장 첫 라인에 있어야 한다.
struct ClientInfo
{
    Overlappedoverlapped;
    SOCKET socket;
    char buf[MAX_BUFFER_SIZE];
    int prevBytes;

    ClientInfo()
    {
        prevBytes = 0;
        overlapped.wsaBuf.buf = buf;
        overlapped.type = OverlappedType::Invalid;
    }
};
```

IOCP

- std::mutex
  - #include <mutex>에 포함
  - CreateMutex()를 통해 얻은 Mutex와 **달리 서로 다른 프로세스 간의 동기화에 사용할 수 없기** 때문에 임계 영역(critical section)과 같다고 볼 수 있다

- HANDLE CreateIoCompletionPort(HANDLE FileHandle,
  HANDLE ExistingCompletionPort,
  ULONG_PTR CompletionKey,
  DWORD NumberOfConcurrentThreads)
  - 지정된 파일 핸들에 연결하거나, 파일 핸들과 아직 연결되지 않은 IOCP를 만들어 나중에 연결
  - FileHandle : 연결할 핸들, 새로 만들 경우 INVALID_HANDLE_VALUE
  - ExistingCompletionPort : 기존 IOCP의 핸들, 새로 만들 경우 NULL
  - CompletionKey : 연결할 핸들에 대한 사용자 정의 Key
  - NumberOfConcurrentThreads : O/S에서 IOCP에 대한 I/O 완료 패킷을 동시에 처리할 수 있는 최대 스레드 수
    0이면 시스템에서 시스템에 프로세서가 있는 만큼 동시에 스레드를 실행

- SOCKET WSASocket(int af, int type, int protocol, LPWSAPROTOCOL_INFO lpProtocolInfo, GROUP g, DWORD dwFlags)
  - 세부 정보 : 링크
  - **lpProtocolInof** : 만들 소켓의 특성을 정의하는 WSAPROTOCOL_INFO 구조체에 대한 포인터
  - **g** : 새 소켓 그룹을 만들 때 수행할 소켓 그룹 ID
  - **dwFlags** : 추가 소켓 특성을 지정하는 데 사용, 일반적으로 WSA_FLAG_OVERLAPPED를 사용

IOCP

```
std::mutex g_acceptLock;
std::mutex g_lock;
HANDLE g_hIOCP;
SOCKET g_listenSocket;
std::map<ULONG_PTR, ClientInfo> g_sockInfoList; // 클라이언트의 SOCKET을 DWORD(ULONG_PTR)로 변경해 Key로 사용.

unsigned int WINAPI ThreadMain(LPVOID);
bool CreateWorkerThread();
bool Recv(ClientInfo* info);
void Send(ClientInfo* info);
void CloseSocket(SOCKET socket);

int main()
{
    // 윈속 초기화.
    ...
    // IOCP(Io Completion Port) 객체 생성.
    g_hIOCP =CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
    // socket() 소켓 생성.
    g_listenSocket = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
    if (INVALID_SOCKET == g_listen_socket) err_quit("socket");
    // bind() 소켓 설정.
    ...
    // listen() 수신 대기열 생성.
    ...
    // Worker Thread 생성.
    if (!CreateWorkerThread()) return -1;
```

IOCP

```
    // 데이터 통신에 사용할 변수.
    SOCKADDR_IN clientaddr;
    int addrlen = sizeof(SOCKADDR_IN);
    ZeroMemory(&clientaddr, addrlen);
    while (1)
    {
        g_acceptLock.lock();
        // WSAAccept() : Winsock2에서 지원하는 accept()
        SOCKET clientSocket = WSAAccept(g_listenSocket, (SOCKADDR*)&clientAddr, &addrLen, NULL, NULL);
        g_acceptLock.unlock();
        if (INVALID_SOCKET == clientSocket) continue;

        printf("Wn[IOCP 서버] 클라이언트 접속 : IP 주소=%s, 포트 번호=%dWn “, inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port));

        g_lock.lock();
        ClientInfo* info = &g_sockInfoList[(ULONG_PTR)clientSocket];
        info->socket = clientSocket;
        info->overlapped.type = OverlappedType::Recv;
        g_lock.unlock();
        // 클라이언트의 SOCKET을 IOCP의 HANDLE과 CompletionKey로 사용.
        if (g_hIOCP != CreateIoCompletionPort((HANDLE)info->socket, g_hIOCP, (ULONG_PTR)info->socket, 0)) break;
        if(!Recv(info)) break;
    }
    closesocket(g_listenSocket);
    // 윈속 사용이 끝났다.
    WSACleanup();
}
```

IOCP

- VOID GetSystemInfo(LPSYSTEM_INFO lpSystemInfo)
  - #include <process.h>에 포함
  - 현재 시스템에 대한 정보를 검색
  - LPSYSTEM_INFO : 링크

- uintptr_t __cdecl _beginthreadex(void* _Security,
  unsigned _StackSize,
  _beginthreadex_proc_type _StartAddress,
  void* _ArgList,
  unsigned _InitFlag,
  unsigned* _ThrdAddr)
  - CreateThread()대신 _beginthreadex()를 써야하는 경우
    - 부동 소수형 변수나 함수를 사용
    - C의 malloc과 free나 C++ 의 new와 delete 를 사용
    - stdio.h 나 io.h에서 어떤 함수를 호출
    - strtok() 나 rand() 와 같이 정적 버퍼를 사용 하는 어떤 런타임 함수를 호출
  - Security : SECURITY_ATTRIBUTES 자식 프로세스에서 상속할 수 있는지 여부를 결정하는 구조체에 대한 포인터
  - StackSize : 새로 생성할 스레드에 할당될 스택의 크기
  - StartAddress : 함수 스레드의 시작 주소
  - ArgList : 스레드 함수에 전달될 인자, 전달할 인자가 없다면 NULL
  - InitFlag : 초기 상태를 제어하는 플래그, CREATE_SUSPENDED(ResumThread() 함수가 호출되기 전까지 실행되지 않음)
  - ThrdAddr : 스레드 식별자를 수신하는 변수, NULL을 허용하지 않음

IOCP

```
bool CreateWorkerThread()
{
    // 시스템 정보 가져온다.
    SYSTEM_INFO sysInfo;
    GetSystemInfo(&sysInfo);

    // CPU 갯수 확인.
    printf_s("[IOCP 서버] CPU : %d개₩n", sysInfo.dwNumberOfProcessors);

    // 적절한 작업 스레드의 갯수 = (CPU * 2) + 1
    const int numThread = sysInfo.dwNumberOfProcessors * 2 + 1;
    unsigned int threadId;
    HANDLE hThread;
    for (int i = 0; numThread > i; i++)
    {
        hThread = (HANDLE)_beginthreadex(NULL, 0, &ThreadMain, NULL, CREATE_SUSPENDED, &threadId);
        if (!hThread) false;

        ResumeThread(hThread);
    }

    return true;
}
```

IOCP

- ```
  int WSARecv(SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
    LPDWORD lpFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine)
  ```
  - Winsock2에서 지원하는 recv()
  - overlapped I/O와 부분 데이터그램 수신(Partial Datagram Notification)등 몇 가지 기능이 추가
  - **s** : recv에 연결하는 소켓
  - **lpBuffers** : WSABUF 배열에 대한 포인터
  - **dwBufferCount** : WSABUF 배열의 길이
  - **lpNumberOfBytesRecvd** : 수신이 바로 완료됐을 경우 받은 데이터 크기
  - **lpFlags** : 함수 호출 동작을 수정하는데 사용되는 플래그. 일반적으로 0을 사용
  - **lpOverlapped** : WSAOVERLAPPED 구조체에 대한 포인터
  - **lpCompletionRoutine** : 수신 작업이 완료될 때 호출되는 완료 루틴에 대한 포인터
  ```

IOCP

```
bool Recv(ClientInfo* info)
{
    info->overlapped.wsaBuf.len = MAX_BUFFER_SIZE;
    ZeroMemory(info->buf, MAX_BUFFER_SIZE);
    ZeroMemory(&info->overlapped.wsaOverlapped, sizeof(info->overlapped.wsaOverlapped));

    DWORD flags = 0;
    DWORD recvBytes;
    int retval = WSARecv(
        info->socket,
        &info->overlapped.wsaBuf,
        1,
        &recvBytes,
        &flags,
        &info->overlapped.wsaOverlapped,
        NULL);

    // WSA_IO_PENDING: 겹치는 작업이 성공적으로 시작되었으며 나중에 완료가 표시.
    if ((SOCKET_ERROR == retval) && (WSAGetLastError() != WSA_IO_PENDING))
    {
        err_display("WSARcev");
        info->overlapped.type = OverlappedType::Invalid;
        return false;
    }
    return true;
}
```

IOCP

- ```
  int WSASend(SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine)
  ```
  - Winsock2에서 지원하는 send()
  - **s** : send에 연결하는 소켓
  - **lpBuffers** : WSABUF 배열에 대한 포인터
  - **dwBufferCount** : WSABUF 배열의 길이
  - **lpNumberOfBytesSent** : 송신이 바로 완료됐을 경우 받은 데이터 크기
  - **lpFlags** : 함수 호출 동작을 수정하는데 사용되는 플래그. 일반적으로 0을 사용
  - **lpOverlapped** : WSAOVERLAPPED 구조체에 대한 포인터
  - **lpCompletionRoutine** : 수신 작업이 완료될 때 호출되는 완료 루틴에 대한 포인터
  ```

IOCP

```
void Send(ClientInfo* info)
{
    info->overlapped.type = OverlappedType::Send;
    DWORD sendBytes, flags = 0;
    int retval;
    for (auto& clientInfo : g_sockInfoList)
    {
        retval = WSASend(
            clientInfo.second.socket,
            &info->overlapped.wsaBuf,
            1,
            &sendBytes, flags,
            NULL, NULL);

        if ((SOCKET_ERROR == retval) && (WSAGetLastError() != WSA_IO_PENDING))
        {
            err_display("WSASend");
            clientInfo.second.overlapped.type = OverlappedType::Invalid;
        }
    }
    if (OverlappedType::Invalid != info->overlapped.type) info->overlapped.type = OverlappedType::Recv;
    Recv(info);
}

Void CloseSocket(SOCKET socket)
{
    g_sockInfoList.erase((ULONG_PTR)socket);
    closesocket(socket);
}
```

IOCP

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

- CompletionPort : IOCP의 핸들
- lpNumberOfBytesTransferred : 완료된 I/O 작업에서 전송된 바이트 수
- lpCompletionKey : I/O 작업이 완료된 파일 핸들과 연결된 완료 키 값
- lpOverlapped : 완료된 I/O 작업이 시작될 때 지정한 OVERLAPPED 구조체의 주소를 수신하는 변수에 대한 포인터(WSARecv(), WSASend()에서 지정)
- dwMilliseconds :

  완료 패킷이 완료 포트에 나타날 때까지 호출자가 대기하려는 시간

  완료 패킷이 지정된 시간 내에 나타나지 않으면 FALSE를 반환하고 lpOverlapped는 NULL

IOCP

```
unsigned int __stdcall ThreadMain(LPVOID)
{
    DWORD bytesTrans;
    SOCKET completionKey;
    ClientInfo* clientInfo;
    while (true)
    {
        bool result = GetQueuedCompletionStatus(
            g_hIOCP,
            &bytesTrans,
            (PULONG_PTR)&completionKey,
            (LPOVERLAPPED*)&clientInfo,
            INFINITE);
        int error = GetLastError();

        if ((!result && WAIT_TIMEOUT != error) ||
            (OverlappedType::Invalid == clientInfo->overlapped.type) ||
            0 == bytesTrans)
        {
            g_lock.lock();
            printf_s("[IOCP 서버] socket(%lu)클라이언트 접속 종료₩n", completionKey);
            CloseSocket(completionKey);
            g_lock.unlock();
            continue;
        }
```

IOCP

```
        if (OverlappedType::Recv == clientInfo->overlapped.type)
        {
            g_lock.lock();
            clientInfo->overlapped.wsaBuf.len = bytesTrans;

            printf_s("[수신 socket:%lu] Msg : %s(%d byte)]₩n",
                clientInfo->socket,
                clientInfo->overlapped.wsaBuf.buf,
                clientInfo->overlapped.wsaBuf.len);

            // 받은 패킷 그대로 모든 클라이언트에게 보내기.
            Send(clientInfo);
            g_lock.unlock();
        }
    } // while()

    return 0;
} // ThreadMain()
```

Packet 구조체

- 구조체 메모리 정렬

```
typedef struct
{
    char c; // 4byte
    int i;  // 4byte
    short s;// 4byte
}TEST;      // 12byte
```

struct 메모리 구조

c (1byte)	Empty (3byte)	i (4byte)	s (2byte)	Empty (2byte)

```
#pragma pack(1)
typedef struct
{
    char c; // 1byte
    int i;  // 4byte
    short s;// 2byte
}TEST;      // 7byte
#pragma pack()
```

struct 메모리 구조

c (1byte)	i (4byte)	s (2byte)

- Packet으로 데이터 주고 받기

```
// 데이터 받기.
char buf[sizeof(Packet)];
ZeroMemory(buf, sizeof(buf));
recv(client_sock, buf, sizeof(buf), 0);
Packet* recv_packet = (Packet*)buf;

// 데이터 보내기.
Packet send_packet;
// TODO : 발송할 데이터를 패킷에 적용.
send(sock, (char*)&send_packet, sizeof(Packet), 0);
```

연습문제

WinAPI에서 만들었던

보드 게임(*오목*, *체스*)을

소켓 네트워크 서버를 만들어

2인 플레이가 가능하게 만들어 봅시다

- *첫 번째 접속자가 Player1(오목:흑, 체스:백)*
- *두 번째 접속자가 Player2(오목:백, 체스:흑)*