

1. 네트워크 프로그램 기초	2
01. TCP/IP 프로토콜	3
02. 윈도우 소켓	6
03. 환경 구축하기	7
2. 윈도우 네트워크 프로그램 시작하기	10
01. IP 주소 변환	10
02. 바이트 정렬	12
03. 소켓 구조체	14
04. TCP 소켓 서버/클라이언트 프로그램 흐름과 구조	16
socket() 함수	16
bind() 함수	17
listen() 함수	17
accept() 함수	18
send () 함수	18
recv() 함수	19
connect () 함수	20
TCP 서버	20
3. 스레드를 이용한 다중 에코 서버	32
01. 다중 에코 서버	33
02. 다중 에코 클라이언트	38
4. Select 입출력 모델	44
01. 년블로킹 에코 서버	46
02. Select 모델	51
5. WSASyncSelect 입출력 모델	63
01. WSASyncSelect 모델의 이해	64
02. WSASyncSelect 모델 에코 서버	66
6. WSAEventSelect 입출력 모델	75
01. WSAEventSelect 모델의 이해	76
02. WSAEventSelect 모델 에코 서버	81
7. Overlapped 모델	90
01. Overlapped 모델과 비동기 입출력 함수	91
02. Overlapped 이벤트 입출력 모델 서버	95
03. Overlapped 완료 루틴(Completion Routine) 입출력 모델 서버	106
8. IOCP(I/O Completion Port) 입출력 모델	116
01. IOCP 모델의 이해	117
02. IOCP 입출력 모델 에코 서버	120

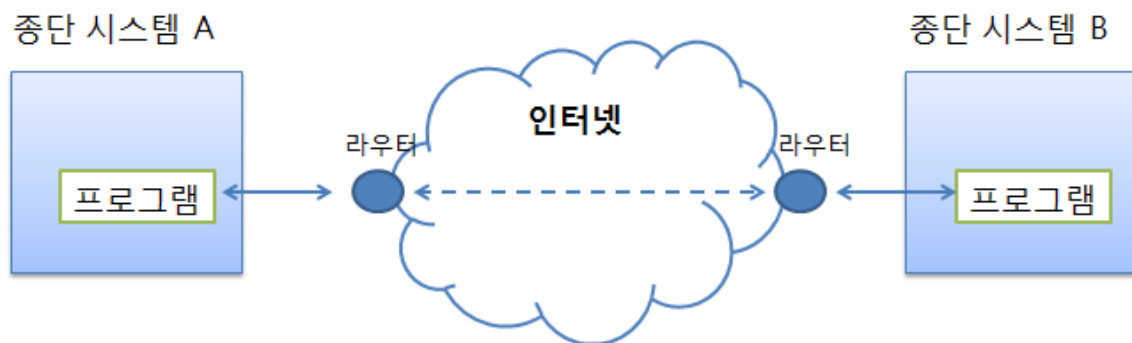
<장제목>

1. 네트워크 프로그램 기초

</장제목>

네트워크란 두 대 이상의 종단 시스템(end-user)이 연결된 형태를 말한다. 여기서 종단 시스템은 네트워크에 연결된 PC, PDA, 휴대폰 등이다. 이런 네트워크가 전세계적으로 연결되어 있는 시스템이 인터넷(Internet)이다. 인터넷은 여러 네트워크 망들의 집합이라 할 수 있는데 네트워크와 또 다른 네트워크를 연결하는 장비를 라우터(router)라하고 하나의 종단 시스템과 다른 종단 시스템이 통신하기 위해서는 서로간의 약속을 따라야 하는데 이 약속을 프로토콜(protocol)이라한다. 또 이 두 시스템이 서로 통신할 수 있도록 사용되는 프로그램을 네트워크 프로그램이라 한다.

다음은 설명을 간단한 그림으로 표현한 것이다.



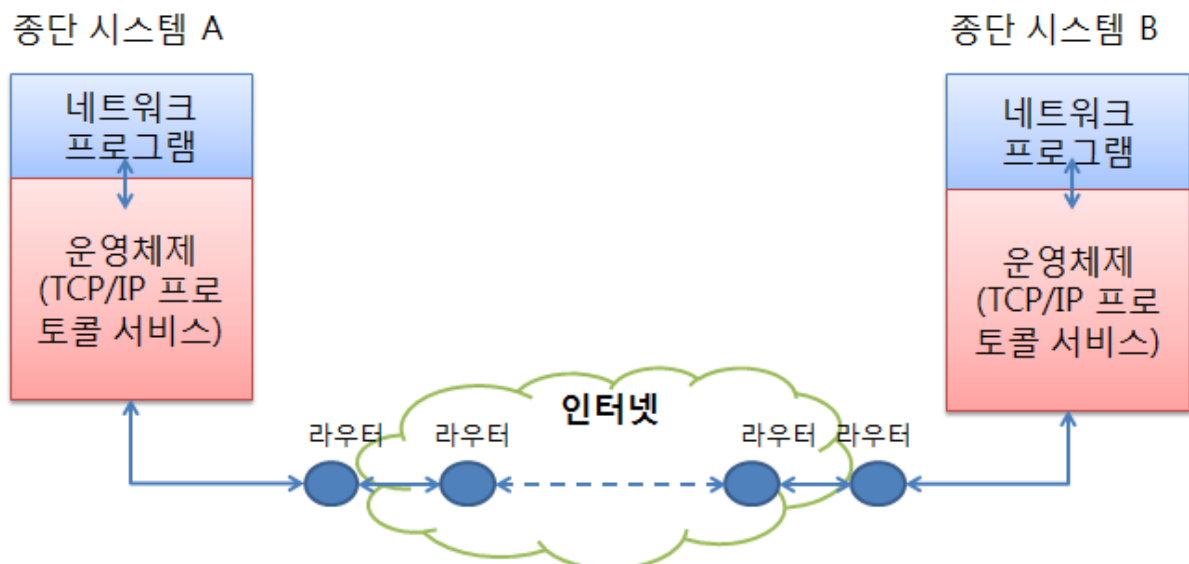
<절제목>

01. TCP/IP 프로토콜

</절제목>

인터넷을 이루는 가장 기본적인 프로토콜이 TCP 와 IP 이며 이 둘을 총칭하여 TCP/IP 라 한다. TCP/IP 프로토콜은 운영체제가 서비스를 제공하므로 네트워크 프로그램(애플리케이션)은 운영체제가 제공하는 TCP/IP 프로토콜 서비스를 사용하여 하고자 하는 일에 집중 할 수 있다.

다음은 네트워크 프로그램과 운영체제 TCP/IP 프로토콜 서비스를 표현한 그림이다.



TCP/IP 는 네 계층으로 나뉜다.

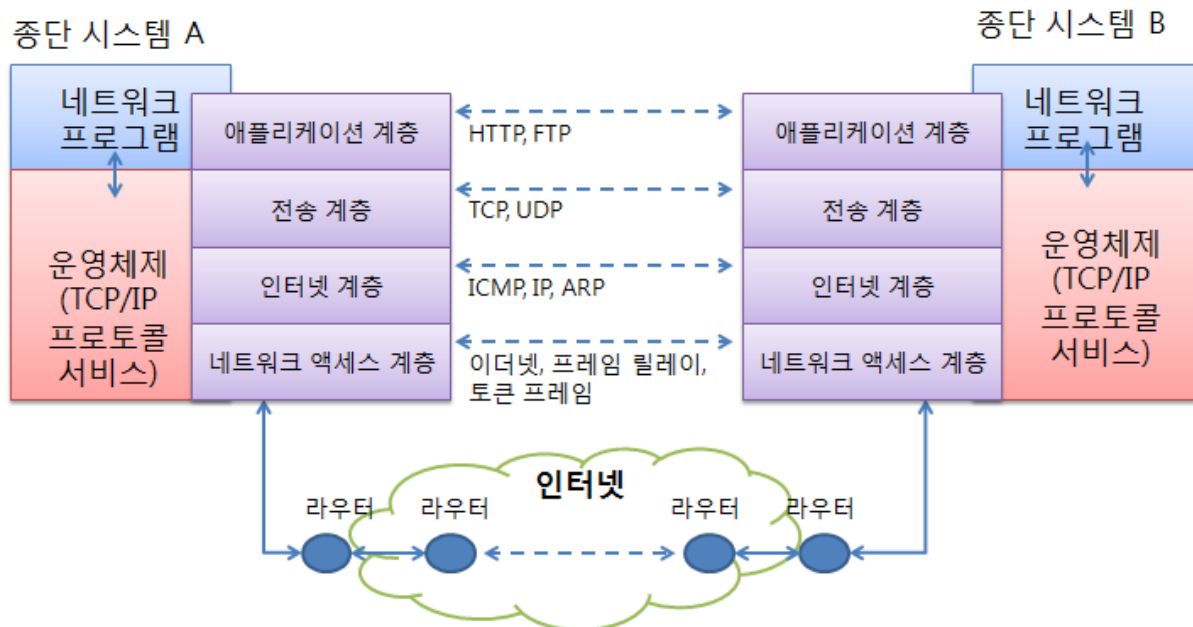
첫째 네트워크 액세스 계층은 시스템을 연결하기 위한 물리적인 규격을 정의 한다. 전기 신호를 처리하고 데이터를 송수신하는 네트워크 하드웨어와 디바이스 드라이버가 이에 해당한다. 하드웨어마다 물리 주소가 할당되며 이 물리 주소를 사용하여 통신한다.

둘째 인터넷 계층은 네트워크 액세스 계층과 전송 계층의 인터페이스를 수행하며 데이터를 종단 시스템까지 전달하는 역할을 수행한다. 인터넷 계층의 핵심은 IP 주소이며 IP 주소는 소프트웨어적인 논리 주소로 전 세계적으로 유니크하게 할당된다. 라우터는 두 종단 시스템을 이 IP 주소를 사용하여 통신한다.

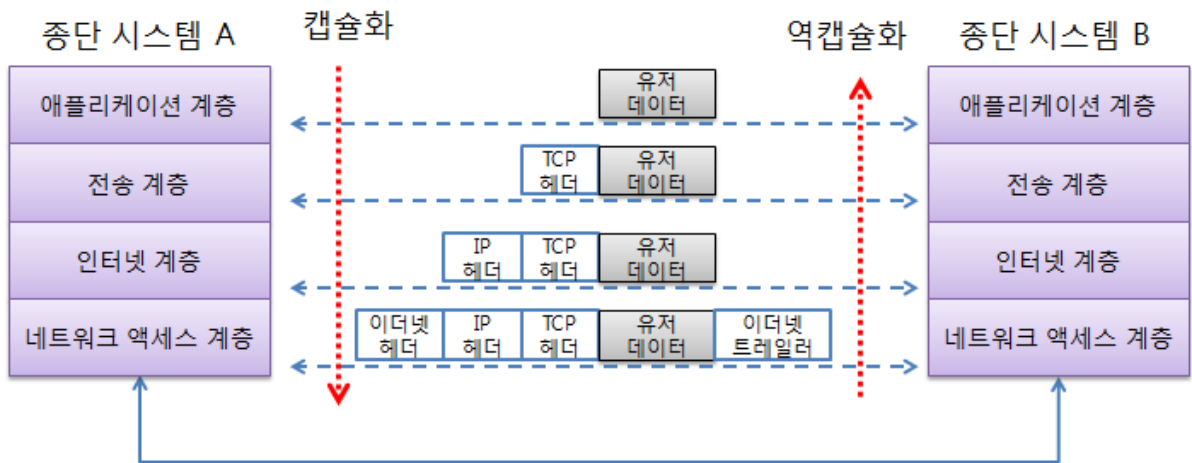
셋째 전송 계층은 신뢰성 있는 데이터 전송을 담당하며 최종 프로그램 목적지까지 통신하는 역할을 담당한다. 종단 시스템까지의 데이터 전송을 인터넷 계층이 담당한다면 종단 시스템에 도착한 데이터를 종단 시스템의 여러 프로그램 중 최종 목적지 프로그램 포트라고 부르는 번호로 구분하여 전달한다. 전송 계층의 프로토콜은 TCP 와 UDP 가 있으며 TCP 는 연결형 프로토콜로 데이터의 경계가 없고 신뢰성이 보장되며 1:1 통신하는 특징을 갖는다. UDP 는 TCP 와 상대적으로 비연결형 프로토콜이며 데이터 경계가 있고 비신뢰적이며 1:1 통신뿐만 아니라 1 대다 통신이나 다대다 통신을 지원한다.

넷째 애플리케이션 계층은 대표적으로 HTTP 프로토콜과 FTP 프로토콜이 이에 해당하며 우리가 작성할 네트워크 프로그램도 이 계층에 해당한다. 앞으로 우리는 소켓 라이브러리를 사용하여 TCP/IP 프로토콜을 기반의 윈도우 네트워크 프로그램에 대한 공부를 진행할 것이다.

다음은 TCP/IP 4 계층에 대한 설명을 그림으로 표현한 것이다.



데이터가 전송되기 위해서는 다음 그림과 같은 각 계층마다 각각의 정보인 헤더가 붙으며 이와 같은 과정을 캡슐화라고 하며 이 결합된 통합 데이터를 패킷(packet)이라 부른다. 애플리케이션 계층의 유저 데이터는 전송 계층에 전달되며 TCP 헤더가 결합되고 인터넷 계층에 전달되어 IP 헤더가 결합된다. 또 네트워크 액세스 계층에 전달되어 이더넷 헤더(header)와 이더넷 트레일러(trailer)가 결합되어 하나의 패킷이 전송된다. 다른 종단 시스템에 패킷이 도착하면 결합과정의 역을 수행(역캡슐화)하여 애플리케이션 계층에 유저 데이터가 도착한다.



<절제목>

02. 윈도우 소켓

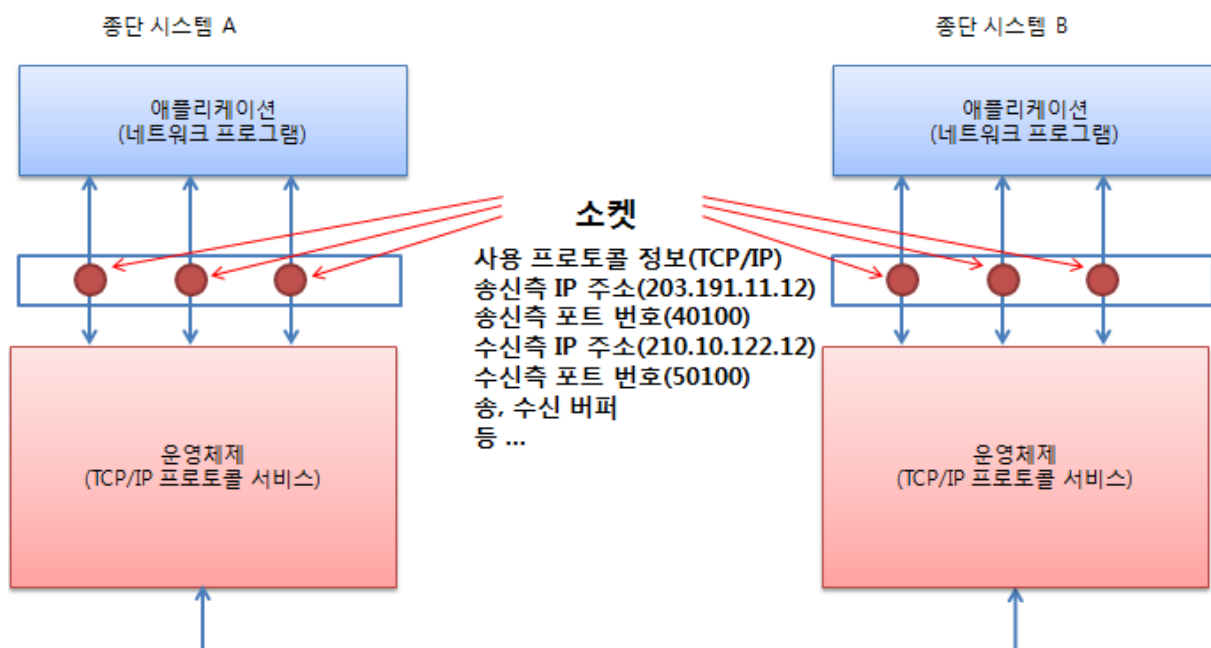
</절제목>

윈도우 소켓은 윈도우 운영체제가 제공하는 네트워크 API 이름으로 버클리 소켓을 기반으로 윈도우 운영체제만의 기능을 추가하며 확장되었다. 버클리 소켓 API 를 그대로 수용하고 95년도에 윈도우 소켓 2.0을 발표했으며 최신 버전으로 윈도우 소켓 2.2가 가장 널리 사용된다.

소켓이란 단어는 라이브러리 이름 외에도 네트워크 통신을 위한 프로그램 인터페이스 객체 자체를 가리키는 단어로도 사용된다.

여기서의 소켓은 통신에 필요한 종단 시스템의 정보(윈도우즈에서 이것을 핸들로 관리한다)를 의미하며 두 종단 시스템이 통신을 하기 위한 프로그램 인터페이스로 사용된다.

다음은 소켓의 개념을 그림으로 표현한 것이다.



<절제목>

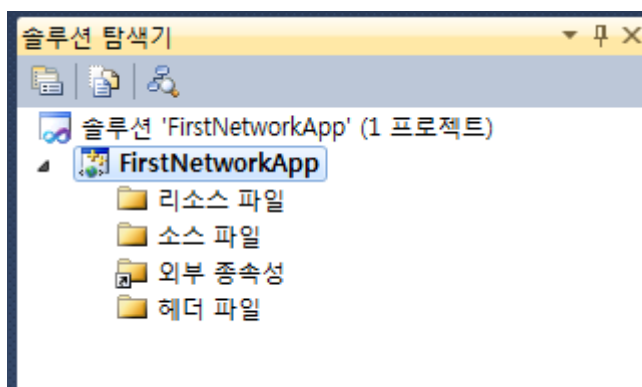
03. 환경 구축하기

</절제목>

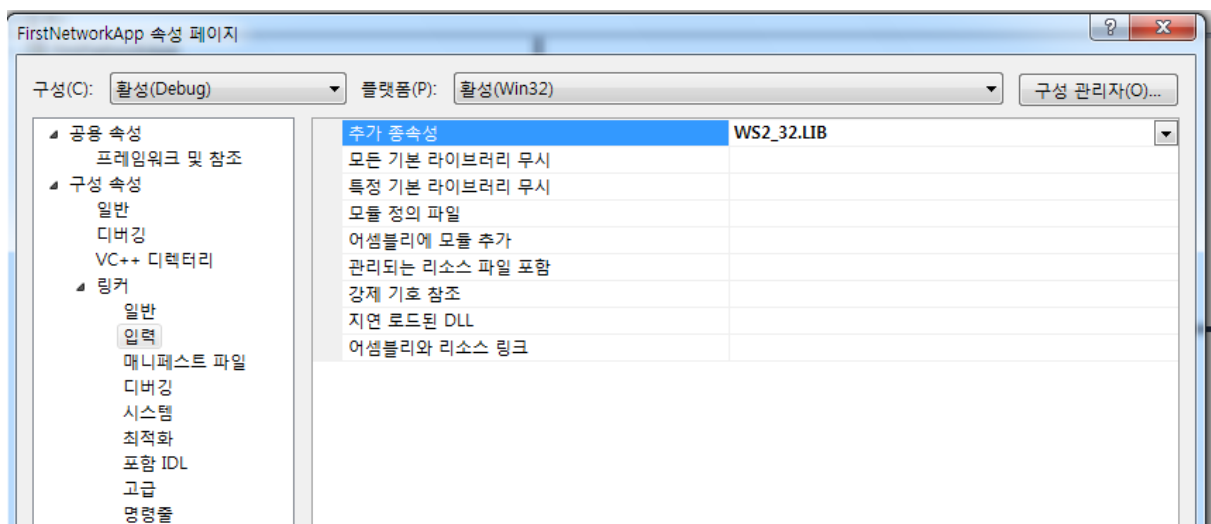
윈도우 소켓을 사용하기 위해서는 가장 먼저 소켓 라이브러리(WS2_32.lib: WS2_32.DLL 의 스텝 코드)를 링크에 추가해야 하며 소켓 라이브러리를 사용하는 소스 코드에서 헤더파일(windows.h)을 포함한다. 마지막으로 소켓 라이브러리(WS2_32.DLL)을 초기화하고 마무리하는 함수를 호출한다.

윈도우 소켓 프로그램 환경 설정하기.

1단계 : VisualStudio2010에서 빈 프로젝트를 생성한다.



2단계 : 링크에 WS2_32.LIB 추가한다.



3단계 : 소스 코드에 헤더(winsock2.h)를 추가하고 윈도우 소켓 DLL 을 초기화, 마무리하는 함수를 호출한다.

<코드>

[예제 01-01] 윈도우 소켓 초기화

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

int main(int argc, char* argv[])
{
    WSADATA wsa;

    if(WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("윈도우 소켓 초기화 실패!\n");
        return -1;
    }

    printf("윈도우 소켓 초기화 성공!\n");

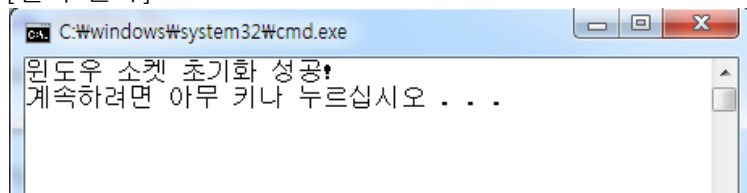
    WSACleanup();

    return 0;
}
```

</코드>

<결과>

[출력 결과]



</결과>

WSAStartup() 함수는 윈속 라이브러리(DLL) 초기화 함수이며 WSACleanup() 함수는 리소스를 반환하는 마무리 함수다.

WSAStartup() 함수의 인수와 반환값은 다음과 같다.

```
int WSAStartup (    WORD wVersionRequested,    LPWSADATA lpWSADATA) ;
```

- wVersionRequested
 - 프로그램이 요구하는 최상위 윈속 버전. 하위 8비트에 주(major) 버전을, 상위 8비트에 부(minor) 버전을 넣어서 전달함
- lpWSADATA
 - WSADATA 타입 변수의 주소. 시스템에서 제공하는 윈속 구현에 대한 세부 사항을 얻을 수 있음

성공은 0, 실패는 오류 코드를 반환한다.

다음은 마무리 함수다. `int WSACleanup (void)` ; 성공은 0, 실패는 `SOCKET_ERROR` 를 반환한다.

`MAKEWORD(a,b)`는 1Byte a 와 1Byte b 를 2Byte 로 결합한다. 2Byte 두 정수를 결합하여 4Byte 정수를 만드는 `MAKELONG()` 과 비슷한 매크로함수다.

<장제목>

2. 윈도우 네트워크 프로그램 시작하기

</장제목>

이 장에서는 네트워크 프로그램 필요한 가장 기본적인 사항들을 공부한다. 서로 다른 종단 시스템간의 통신을 위한 바이트 정렬이나 문자열 IP 주소를 정수로 변환하고 소켓 구조체에 대한 기본 내용과 기본적인 소켓 프로그램 구조에 대해 공부한다.

<절제목>

01. IP 주소 변환

</절제목>

네트워크 프로그램에서 IP 주소는 4바이트 정수(IPV4)로 사용되지만 사람들은 IP 주소를 문자열로 확인하고 이해하는 것이 편하므로 서로간의 변환이 필요하게 되고 이런 변환을 위한 함수가 `inet_addr()`과 `inet_ntoa()` 다.

다음은 정수와 문자열 IP 를 변환하는 간단한 예제다.

<코드>

[예제 02-01] IP 주소 변환

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

int main(int argc, char* argv[])
{
    WSADATA wsa;

    if(WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("윈도우 소켓 초기화 실패!\n");
        return -1;
    }

    // 문자열 주소 출력
    char *ipaddr = "230.200.12.5";
    printf("IP 문자열 주소 : %s\n", ipaddr);

    // 문자열 주소를 4byte 정수로 변환
    printf("IP 문자열 주소 = > 정수 = 0x%08x\n", inet_addr(ipaddr));

    // 4byte 정수를 문자열 주소로 변환
```

```

    IN_ADDR in_addr;
    in_addr.s_addr = inet_addr(ipaddr);
    printf("IP 정수 => 문자열 주소 = %s\n", inet_ntoa(in_addr));

    WSACleanup();
    return 0;
}

```

</코드>

<결과>

[출력 결과]

```

C:\windows\system32\cmd.exe
IP 문자열 주소 : 230.200.12.5
IP 문자열 주소 = > 정수 = 0x050CC8E6
IP 정수 => 문자열 주소 = 230.200.12.5

```

</결과>

inet_addr() 함수는 문자열 형태의 IP 주소를 바이트 정렬된 4바이트 정수로 반환한다.

inet_ntoa() 함수는 4바이트 정수를 문자열 형태의 IP 주소로 반환한다.

IN_ADDR 구조체는 다음과 같이 정의되어 IP 주소를 1바이트 형태나 2바이트 형태, 4바이트 형태의 정수로 사용하기 쉽도록 정의된 구조체다.

```

typedef struct in_addr {
    union {
        struct { UCHAR s_b1,s_b2,s_b3,s_b4; } S_un_b;
        struct { USHORT s_w1,s_w2; } S_un_w;
        ULONG S_addr;
    } S_un;
#define s_addr S_un.S_addr /* can be used for most tcp & ip code */
#define s_host S_un.S_un_b.s_b2 // host on imp
#define s_net S_un.S_un_b.s_b1 // network
#define s_imp S_un.S_un_w.s_w2 // imp
#define s_impno S_un.S_un_b.s_b4 // imp #
#define s_lh S_un.S_un_b.s_b3 // logical host
} IN_ADDR, *PIN_ADDR, FAR *LPIN_ADDR;

```

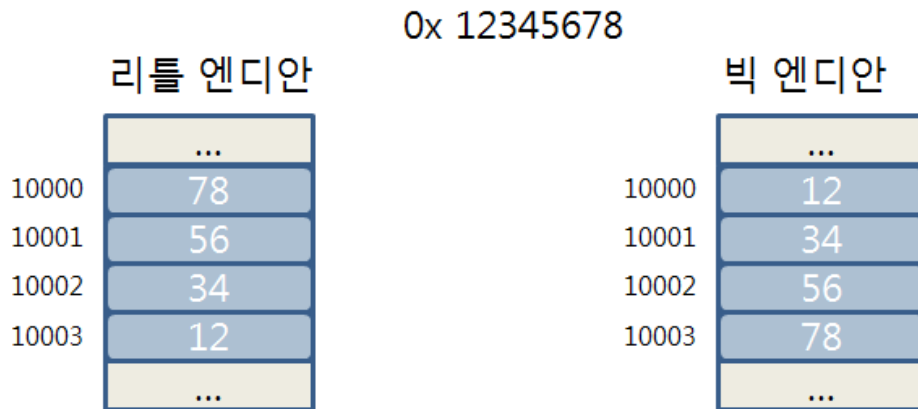
<절제목>

02. 바이트 정렬

</절제목>

데이터를 메모리에 저장하는 표준 방식은 크게 두 가지다. 첫 번째 리틀 엔디안(little-endian)은 작은 주소에 데이터의 끝 값부터 저장하며 두 번째 빅 엔디안(big-endian)은 큰 주소에 데이터의 끝 값부터 저장하는 방식이다. 하나의 시스템에서 프로그램 한다면 문제가 없겠지만 네트워크 프로그램처럼 서로 다른 두 시스템에서 데이터를 주고 받아야 한다면 데이터 저장 방식 때문에 문제가 될 수 있다. 그러다 보니 소켓 프로그램에서 네트워크의 데이터 표준 방식은 빅 엔디안을 사용하자고 약속되어 있으며 IP 와 포트 번호를 모두 빅 엔디안 방식으로 설정하고 이것을 네트워크 바이트 정렬이라 한다. 상대적으로 시스템 고유의 바이트 정렬 방식을 호스트 바이트 정렬이라 한다.

다음은 리틀 엔디안과 빅엔디안에 4바이트 0x12345678이 저장되는 방식을 표현한 것이다.



다음 예제는 바이트 정렬에 사용되는 네 함수를 정리한 예제다.

<코드>

```
[예제 02-02] 바이트 정렬
#include <winsock2.h>
#include <stdio.h>

int main(int argc, char* argv[])
{
    WSADATA wsa;
    if(WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("윈도우 소켓 초기화 실패!\n");
    }
}
```

```

        return -1;
    }

    unsigned short us = 0x1234;
    unsigned long ul = 0x12345678;

    // 호스트 바이트를 네트워크 바이트로 변환한다.
    printf("0x%08x = > 0x%08x\n", us, htons(us));
    printf("0x%08x = > 0x%08x\n", ul, htonl(ul));

    unsigned short n_us = htons(us);
    unsigned long n_ul = htonl(ul);
    // 네트워크 바이트를 호스트 바이트로 변환한다.
    printf("0x%08x = > 0x%08x\n", n_us, ntohs(n_us));
    printf("0x%08x = > 0x%08x\n", n_ul, ntohl(n_ul));

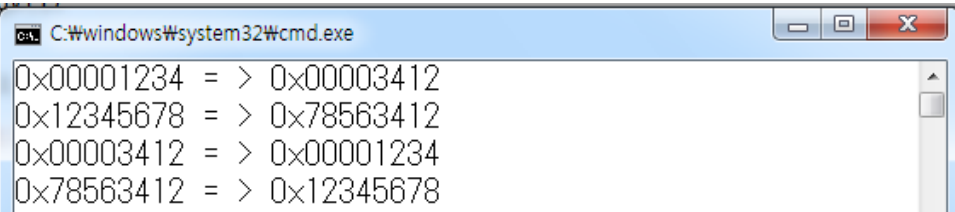
    WSACleanup();
    return 0;
}

```

</코드>

<결과>

[출력 결과]



```

C:\windows\system32\cmd.exe
0x00001234 = > 0x00003412
0x12345678 = > 0x78563412
0x00003412 = > 0x00001234
0x78563412 = > 0x12345678

```

</결과>

출력 결과만 보더라도 어렵지 않게 알 수 있다.

<절제목>

03. 소켓 구조체

</절제목>

소켓을 사용하기 위해 필수적으로 사용되는 `sockaddr` 구조체와 `sockaddr_in` 구조체를 공부하도록 하겠다.

다음은 `sockaddr` 구조체의 정의다.,

```
typedef struct sockaddr {
```

```
    u_short sa_family; // Address family.  
    CHAR sa_data[14]; // Up to 14 bytes of direct address.  
} SOCKADDR, *PSOCKADDR, FAR *LPSOCKADDR;
```

`sa_family` 는 2바이트 크기로 주소 체계를 지정한다. TCP , UDP 프로토콜은 `AF_INET(PF_INET)`을 사용하게 된다.

`sa_data[14]`는 6바이트에 IP 주소와 포트 번호가 지정되며 나머지 8바이트는 예약되어 사용되지 않는다.

위 구조체는 프로그램에서 IP 주소와 포트 번호를 지정하기 위해 비트연산을 수행해야 하므로 좀더 사용하기 편한 `sockaddr_int` 구조체를 사용할 수 있다.

다음은 `sockaddr_int` 구조체의 정의다.

```
typedef struct sockaddr_in {
```

```
    short sin_family;  
    USHORT sin_port;  
    IN_ADDR sin_addr;  
    CHAR sin_zero[8];  
} SOCKADDR_IN, *PSOCKADDR_IN;
```

로 정의되어 IP 주소와 포트 번호를 쉽게 지정하여 사용할 수 있다.

`sin_family`에는 2바이트 주소 체계를 지정한다.

`sin_port`에는 2바이트 정수 포트 번호를 지정한다.

`sin_addr`에는 4바이트 정수 IP 주소를 지정한다.

`sin_zero[8]`는 예약되어 사용되지 않는다.

다음은 `IN_ADDR` 구조체의 정의로 IP 주소를 1,2,4 바이트 단위로 쉽게 접근할 수 있도록 제공하는 구조체다.

```
typedef struct in_addr {
```

```
    union {  
        struct { UCHAR s_b1,s_b2,s_b3,s_b4; } S_un_b;  
        struct { USHORT s_w1,s_w2; } S_un_w;  
        ULONG S_addr;
```

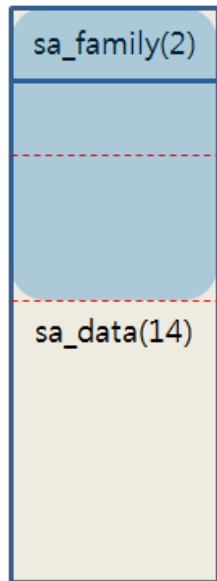
```

    } S_un;
#define s_addr S_un.S_addr /* can be used for most tcp & ip code */
#define s_host S_un.S_un_b.s_b2 // host on imp
#define s_net S_un.S_un_b.s_b1 // network
#define s_imp S_un.S_un_w.s_w2 // imp
#define s_impno S_un.S_un_b.s_b4 // imp #
#define s_lh S_un.S_un_b.s_b3 // logical host
} IN_ADDR, *PIN_ADDR, FAR *LPIN_ADDR;

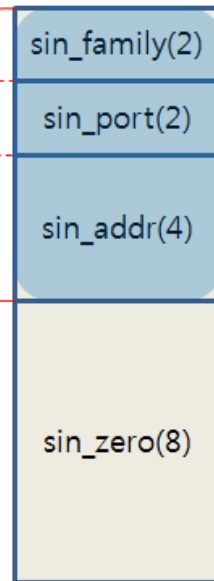
```

다음은 sockaddr 구조체와 sockaddr_in 구조체를 비교한 그림이다.

sockaddr 구조체



sockaddr_in 구조체



실제 사용 영역

<절제목>

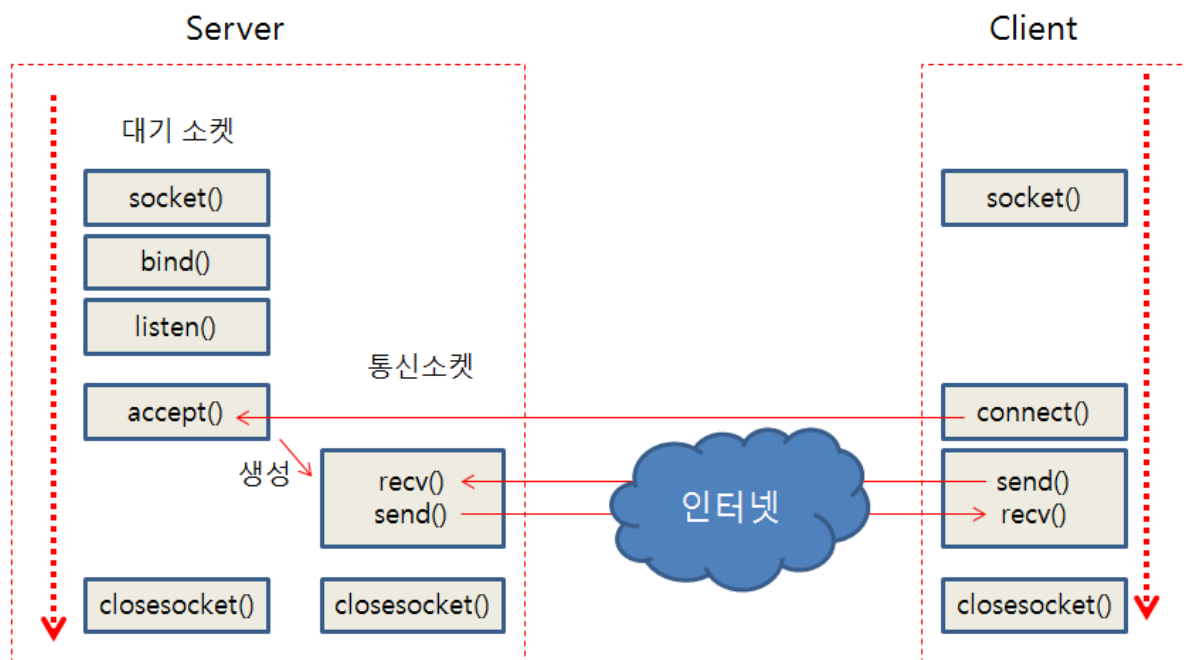
04. TCP 소켓 서버/클라이언트 프로그램 흐름과 구조

</절제목>

TCP 서버는 조금 복잡하며 크게 두 소켓의 흐름으로 구성된다. 하나는 대기 소켓의 흐름이며 하나는 통신 소켓의 흐름이다. 대기 소켓은 클라이언트의 접속을 대기하고 클라이언트 요청을 수락하며 통신 소켓을 생성하는 역할을 담당한다. 통신 소켓은 대기 소켓에 의해 자동으로 생성되며 클라이언트 소켓과 통신을 담당한다.

TCP 클라이언트는 소켓을 생성하고 서버에 접속을 요청하여 요청이 받아들여지면 자동으로 통신할 수 있는 소켓으로 변화하며 이 소켓을 사용하여 서버와 통신한다.

다음은 서버와 클라이언트가 통신하는 흐름을 표현한 그림이다.



<중제목>

socket() 함수

</중제목>

소켓 프로그램에서 가장 먼저 해야 할 일은 소켓을 생성하는 것이다. 소켓은 두 종단 시스템이 통신하기 위한 연결점이다. 윈도우 소켓은 통신에 사용되는 정보 집합이며 핸들을 사용하여 관리된다.

소켓 함수 SOCKET socket(int af, int type, int protocol);의

af 는 주소 체계로 프로토콜 마다 주소를 체계를 지정하는 방법이 달라지며 TCP, UDP 프로토콜을 사용한다면 AF_INET 을 지정한다.

type 은 데이터 통신을 위한 프로토콜 유형으로 SOCKET_STREAM, SOCK_DGRAM, SOCK_RAW 를 지정한다. SOCKET_STREAM 은 연결형 프로토콜을 SOCK_DGRAM 은 비연결형 프로토콜을 나타낸다. SOCK_RAW 는 애플리케이션 수준에서 프로토콜 헤더를 직접 다룰 수 있는 소켓을 생성한다.

protocol 은 프로토콜을 결정한다. TCP 프로토콜은 IPPROTO_TCP, UDP 프로토콜은 IPPROTO_UDP 를 지정한다. 만약 주소 체계와 소켓 타입이 정확히 결정되면 protocol 인수는 0으로 생략할 수 있다.

반환은 소켓의 핸들이며 실패 시 INVALID_SOCKET 을 반환한다.

<중제목>

bind() 함수

</중제목>

bind() 함수는 대기 소켓의 로컬 IP 주소와 로컬 포트 번호를 결정한다.

소켓 함수 int bind(SOCKET s, const struct sockaddr* name, int namelen);의

s 는 socket() 함수로 생성한 소켓의 핸들이며 이 소켓에 로컬 IP 와 포트를 할당한다.

name 은 소켓 구조체를 사용하여 로컬 IP 와 포트를 할당한다.

namelen 은 소켓 구조체의 크기다.

성공 시 0을 반환하며 실패 시 SOCKET_ERROR 를 반환한다.

<중제목>

listen() 함수

</중제목>

listen() 함수가 호출되면 대기 소켓은 클라이언트의 접속을 받을 수 있는 상태가 되며 클라이언트의 정보를 저장하는 접속 대기 큐(connection queue)가 만들어 진다.

소켓 함수 int listen(SOCKET s, int backlog); 의

s 는 대기 소켓의 핸들이다.

backlog 는 동시에 접속 가능한 클라이언트의 개수로 접속 대기 큐의 크기다. 서버가 클라이언트의 접속을 처리하지 않더라도 접속 대기 큐의 크기만큼 클라이언트 접속이 성공한다. 최대값으로 SOMAXCONN 값을 사용한다.

반환값은 성공 시 0이며 실패 시 SOCKET_ERROR 다.

<중제목>

accept() 함수

</중제목>

accept() 함수는 접속 대기 큐를 확인하여 클라이언트가 접속하면 클라이언트와 통신할 수 있는 새로운 통신 소켓을 반환한다. out parameter 인수로 접속한 클라이언트의 IP 주소와 포트 번호를 반환받을 수 있다.

소켓 함수 SOCKET accept(SOCKET s, struct sockaddr* addr, int* addrlen);의

s 는 대기 소켓의 핸들이다.

addr 은 클라이언트의 IP 주소와 포트 번호를 받아오는 sockaddr 구조체 out parameter 변수다. 소켓의 핸들로 언제든지 클라이언트 정보를 확인할 수 있으므로 바로 클라이언트 정보를 반환받고 싶지 않다면 NULL 로 지정한다.

addrlen 은 addr 의 크기를 초기화하여 인수로 하면 실제 초기화된 구조체의 크기를 반환하는 in, out parameter 다.

반환값은 새로운 통신 소켓의 핸들이며 실패 시 INVALID_SOCKET 을 반환한다.

<중제목>

send () 함수

</중제목>

send() 함수는 애플리케이션 버퍼의 데이터를 TCP 프로토콜의 송신 버퍼(이하 송신 버퍼)로 복사한다. 데이터를 송신 버퍼로 모두 복사한 후 실제 복사한 크기를 반환한다. send()함수는 블로킹 소켓과 넌블로킹 소켓에 따라 다르게 동작하며 두 소켓 방식은 뒤 장에서 공부한다. 블로킹 소켓일 때 send() 함수는 애플리케이션 버퍼의 데이터를 모두 TCP 프로토콜의 송신 버퍼에 복사할 때까지 블록 상태가 되며 모두 복사가 완료한 후에 반환한다. 넌블로킹 소켓일 때

send 함수는 애플리케이션 버퍼의 데이터가 송신 버퍼의 크기보다 커서 모두 복사할 수 없다면 송신 버퍼의 남은 크기만큼만 복사하고 바로 반환하며 실제 복사한 크기값을 반환한다.

소켓 함수 `int send(SOCKET s, const char* buf, int len, int flags);`의

s 는 통신 소켓의 핸들이다.

buf 는 애플리케이션 버퍼의 주소다.

len 은 송신 버퍼에 쓸 크기다.

flags 는 보낼 데이터의 소켓 옵션을 지정할 수 있으며 대부분 0이다. MSG_DONTROUTE 나 MSG_00B 등을 지정할 수 있으며 거의 사용되지 않는다.

반환값은 실제 보낸 크기(바이트)이며 실패 시 SOCKET_ERROR 를 반환한다.

<중제목>

recv() 함수

</중제목>

recv() 함수는 TCP 프로토콜 수신 버퍼(이하 수신 버퍼)의 데이터를 애플리케이션 버퍼로 복사한다. recv() 함수도 블로킹 소켓과 논블로킹 소켓에 따라 다르게 동작하며 블로킹 소켓이라면 recv() 함수는 수신 버퍼에 데이터가 도착할 때까지 블로킹된다. 논블로킹 소켓이라면 recv() 함수는 수신 버퍼에 데이터가 없으면 에러값(SOCKET_ERROR)으로 반환되며 수신 버퍼에 데이터가 있으면 애플리케이션 버퍼에 데이터를 복사하고 반환된다. recv() 함수는

소켓 함수 `int recv(SOCKET s, char* buf, int len, int flags);`의

s 는 통신 소켓의 핸들이다.

buf 는 수신 버퍼의 데이터를 복사할 애플리케이션 버퍼의 주소다.

len 은 수신 버퍼에서 애플리케이션 버퍼로 복사할 최대 데이터 크기다. 반환값은 len 보다 작거나 같다.

flags 는 받는 데이터의 옵션을 설정할 수 있으며 대부분 0이다. MSG_PEEK 나 MSG_00B 를 사용할 수 있으며 거의 사용되지 않는다.

반환값은 성공 시 두 가지이며 데이터가 수신 버퍼에 도착하여 데이터를 애플리케이션 버퍼로 복사했다면 복사한 데이터의 크기값을 반환한다. 또 연결된 소켓이 closesocket() 함수를

호출하여 접속을 종료하면 TCP 프로토콜(운영체제)은 접속 종료를 위한 절차에 들어가며 이때 recv() 함수는 0을 반환한다. 이런 정상 종료를 우아한 종료(graceful close)라 한다.

<중제목>

connect () 함수

</중제목>

connect() 함수는 클라이언트가 서버에 접속을 요청하며 TCP 프로토콜(운영체제) 수준의 접속을 위한 절차에 들어간다. 이때 서버는 listen() 함수가 호출되어 있어야 접속이 성공한다. 접속이 성공하면 connect() 함수 호출 이후의 소켓은 통신이 가능한 통신 소켓이 된다.

소켓 함수 int connect(SOCKET s, const struct sockaddr* name, int namelen);의

s 는 서버와 통신할 통신 소켓의 핸들이다.

name 은 서버의 원격 IP 와 원격 포트 번호를 설정한 sockaddr 구조체 주소다.

namelen 은 name(소켓 주소 구조체) 크기를 지정한다.

반환값은 성공 시 0이며 실패 시 SOCKET_ERROR 를 반환한다.

<중제목>

TCP 서버

</중제목>

지금까지 공부한 소켓관련 함수를 사용하여 클라이언트가 전송한 문자열을 화면에 출력하는 간단한 TCP 서버를 만들어 보도록 하자. 이 서버는 앞으로 공부할 여러 TCP 서버에 기본이 되므로 잘 알아두어야 한다.

(예제는 ITC00KB00K 윈도우 네트워크 프로그래밍 김선우님의 예제 코드를 참고하여 작성되었다.)

<코드>

[예제 02-03] TCPServer

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

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

```
#define BUFFERSIZE 1024
```

```
// 소켓 함수 오류 출력
```

```
void DisplayMessage()
```

```
{
```

```
    LPVOID pMsg;
```

```
    FormatMessage(
```

```
        FORMAT_MESSAGE_ALLOCATE_BUFFER | // 오류 메시지 저장 메모리를 내부에서
```

할당하라

```
        FORMAT_MESSAGE_FROM_SYSTEM, //운영체제로 부터 오류 메시지를 가져온다
        NULL,
        WSAGetLastError(), //오류 코드
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), //언어(제어판 설정 언어)
        (LPTSTR)&pMsg, // 오류 메시지 outparam
        0, NULL);

    printf("%s\n", pMsg); // 오류 메시지 출력

    LocalFree(pMsg); // 오류 메시지 저장 메모리 반환
}

int main(int argc, char* argv[])
{
    int retval;

    WSADATA wsa;
    if(WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("윈도우 소켓 초기화 실패!\n");
        return -1;
    }

    // 대기 소켓 생성
    SOCKET listenSock;
    listenSock = socket(
        AF_INET, //주소체계: 프로토콜마다 주소체계 다름, 인터넷 사용, IPv4
        SOCK_STREAM, //프로토콜유형: TCP/IP 기반 사용
        0 //앞 두 인자로 프로토콜 결정이 명확하면 0사용, IPPROTO_TCP, IPPROTO_UDP
    );
    if(listenSock == INVALID_SOCKET)
    {
        DisplayMessage();
        return -1;
    }

    // 대기 소켓의 로컬 주소, 포트 설정
    SOCKADDR_IN serveraddr;
    ZeroMemory(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET; //주소체계
    serveraddr.sin_port = htons(40100); //지역포트번호
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY); //지역IP 주소
    retval = bind(listenSock, (SOCKADDR *)&serveraddr, sizeof(serveraddr));
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return -1;
    }

    // 대기 소켓의 접속 대기 큐 생성 및 클라이언트 접속 대기
    retval = listen(
        listenSock,
        SOMAXCONN //접속대기 큐의 크기
    ); // TCP 상태를 LISTENING 변경
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
    }
}
```

```

    return -1;
}

// 데이터 통신에 사용할 변수
SOCKET clientSock;
SOCKADDR_IN clientaddr;
int addrLen;
char buf[BUFFERSIZE];

while(1){
    addrLen = sizeof(clientaddr);
    // 접속 대기
    clientSock = accept(
        listenSock,
        (SOCKADDR *)&clientaddr, //클라이언트의 정보 out param
        &addrLen //주소구조체형식의크기, in(크기지정), out(초기화한크기반환) param
    ); //통신소켓 생성: 원격 IP, 원격 포트 결정
    if(clientSock == INVALID_SOCKET)
    {
        DisplayMessage();
        continue;
    }
    printf("\n[TCP 서버] 클라이언트 접속: IP 주소=%s, 포트 번호=%d\n",
        inet_ntoa(clientaddr.sin_addr), //문자열로 IP주소 변환
        ntohs(clientaddr.sin_port) // 포트번호 네트워크 바이트 정렬을 호스트 바이트
        정렬로 변환
    );

    // 클라이언트와 데이터 통신
    while(1){
        // 데이터 받기
        retval = recv(
            clientSock, //통신소켓핸들
            buf, //받을 애플리케이션 버퍼
            BUFFERSIZE, //수신 버퍼의 최대 크기
            0 //대부분 0 or MSG_PEEK와 MSG_OOB를 사용 가능
        );
        if(retval == SOCKET_ERROR)
        {
            //소켓 비정상 종료
            DisplayMessage();
            break;
        }
        else if(retval == 0)
        {
            //소켓 정상 종료
            DisplayMessage();
            break;
        }
        else{
            // 받은 데이터 출력
            buf[retval] = '\0';
            printf("[TCP 서버] IP 주소=%s, 포트 번호=%d의 받은 메시지:%s\n",
                inet_ntoa(clientaddr.sin_addr),
                ntohs(clientaddr.sin_port),
                buf
            );
        }
    }
}

```

```

        // 통신 소켓 닫기
        closesocket(clientSock);
        printf("\n[TCP 서버] 클라이언트 종료: IP 주소=%s, 포트 번호=%d\n",
               inet_ntoa(clientaddr.sin_addr),
               ntohs(clientaddr.sin_port));
    }

    // 대기 소켓 닫기
    closesocket(listenSock);

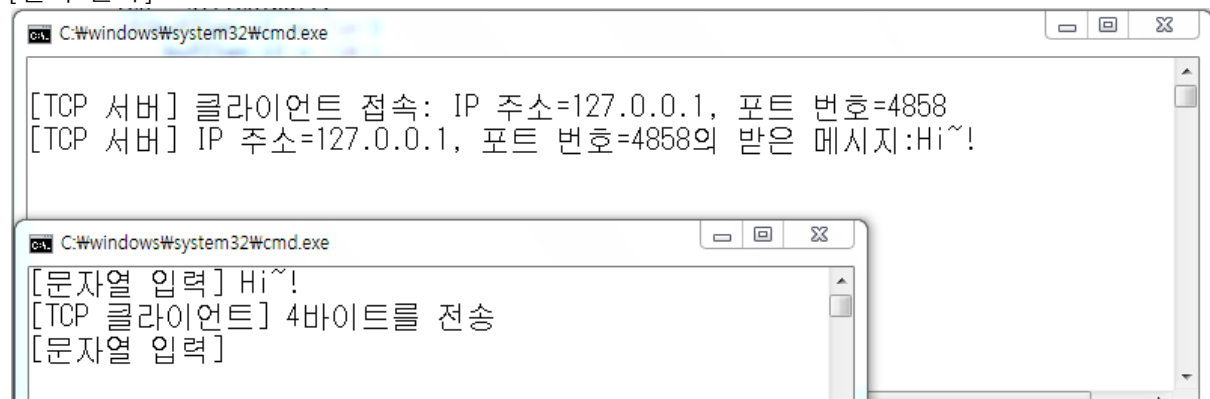
    WSACleanup();
    return 0;
}

```

</코드>

<결과>

[출력 결과]



</결과>

출력결과는 TCP 클라이언트가 간단한 문자열을 TCP 서버에 보냈을 때의 출력결과다.

먼저 `void DisplayMessage(){ }`는

에러 코드 값(정수)을 사람들이 이해할 수 있는 문자열로 해석해 주는 `FormatMessage()` 함수를 호출한다. 소켓 함수들은 대부분의 윈도우 시스템 함수와 마찬가지로 마지막으로 호출된 함수의 에러 코드(정수)가 쓰레드 메모리에 보관되며 이 보관된 에러 코드를 쓰레드 마다 `WSAGetLastError()` 함수를 통해 반환 받을 수 있고 이 반환 받은 에러 코드(정수)를 문자열로 확인하는 함수가 `FormatMessage()` 다. 그러므로 소켓 함수를 호출한 후에는 꼭 에러 코드를 확인하기 위해 `WSAGetLastError()` 함수를 호출할 수 있도록 한다.

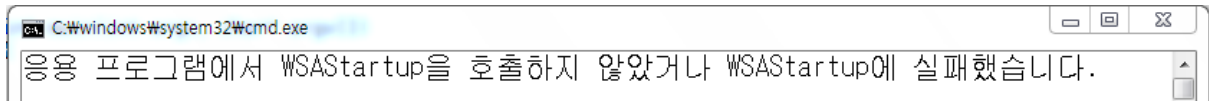
에러 코드를 확인하고 문자열로 출력하기 위해 다음을 호출한다.

```

listenSock = socket(
    AF_INET, SOCK_STREAM, 0 );
if(listenSock == INVALID_SOCKET)
{
    DisplayMessage();
    return -1;
}

```

만약 `WSAStartup(MAKEWORD(2, 2), &wsa)`을 호출하지 않고 `socket()` 함수를 호출한다면 아래와 같은 에러 문자열을 확인할 수 있다.



다음 코드는

```
SOCKADDR_IN serveraddr;  
ZeroMemory(&serveraddr, sizeof(serveraddr));  
serveraddr.sin_family = AF_INET; //주소체계  
serveraddr.sin_port = htons(40100); //지역포트번호  
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY); //지역IP 주소
```

서버의 대기 소켓을 위한 로컬 IP 주소와 로컬 포트 번호를 지정하기 위해 serveraddr 구조체 변수를 ZeroMemory() 함수로 0으로 모두 초기화한 후 주소체계는 AF_INET 으로 로컬 포트 번호는 40100으로 로컬 IP 는 INADDR_ANY 로 지정한다. 로컬 IP 를 INADDR_ANY 로 설정하면 특정 IP 주소가 아닌 어떤 컴퓨터의 IP 주소에서도 이 TCP 서버를 생성할 수 있다는 것을 의미하고 서버의 IP 주소는 현재 실행하는 시스템의 IP 주소가 된다. TCP 클라이언트는 이 IP 주소와 포트 번호로 접속한다. 주의할 점은 로컬 IP 와 로컬 포트 번호가 네트워크 바이트 정렬이 되도록 htons()와 htonl()을 호출해야 한다.

다음 코드는

```
retval = bind(listenSock, (SOCKADDR *)&serveraddr, sizeof(serveraddr));  
serveraddr 구조체 변수를 사용하여 대기 소켓(listenSock)을 설정한 값으로 설정한다.
```

다음 코드는

```
retval = listen(listenSock, SOMAXCONN );  
대기 소켓(listenSock)을 클라이언트 접속이 성공하도록 접속 대기 큐를 만들어 최대 개수(SOMAXCONN)의 클라이언트가 동시 접속할 수 있도록 설정한다.
```

```
다음 코드는  
while(1){  
    addrlen = sizeof(clientaddr);  
    // 접속 대기  
    clientSock = accept(  
        listenSock,  
        (SOCKADDR *)&clientaddr, //클라이언트의 정보 out param  
        &addrlen //주소구조체형식의크기, in(크기지정), out(초기화한크기반환) param  
    ); //통신소켓 생성: 원격 IP, 원격 포트 결정  
    if(clientSock == INVALID_SOCKET)  
    {  
        DisplayMessage();  
        continue;  
    }  
}
```

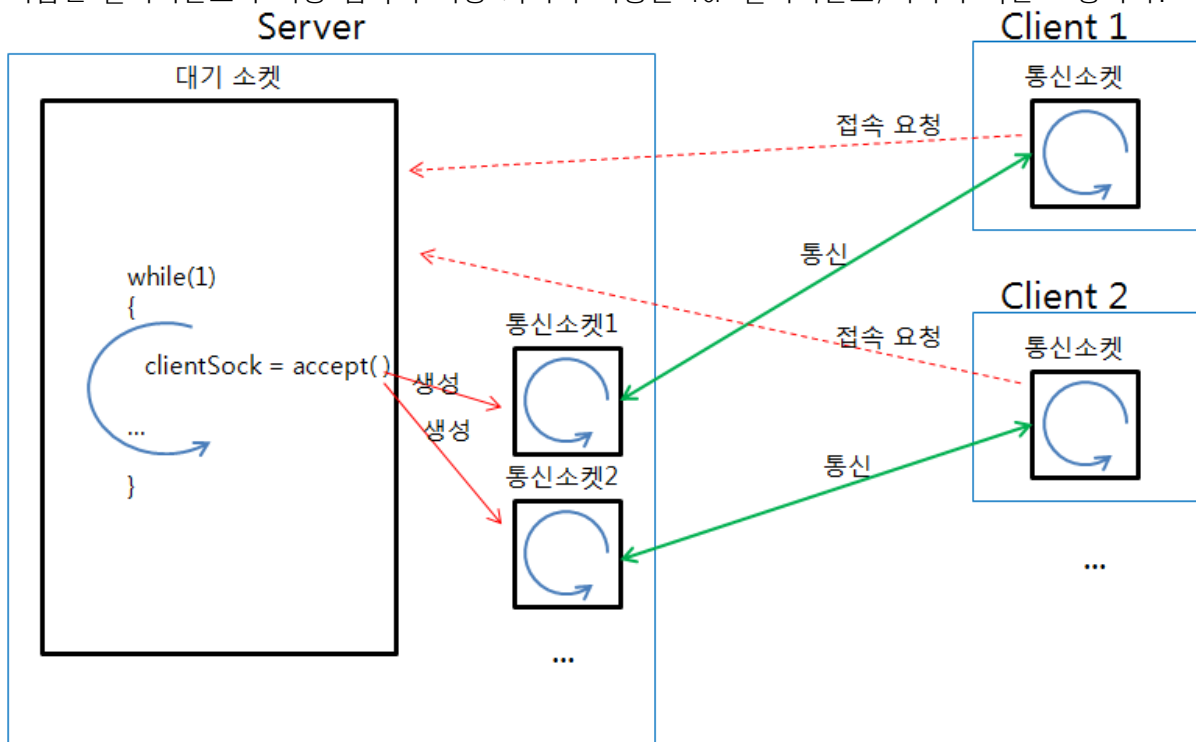

...
}

클라이언트의 접속이 요청 될 때까지 블로킹(대기)되고 클라이언트 접속이 요청되면 그 클라이언트와 통신할 수 있는 소켓을 생성하여 반환(clientSock)한다. clientaddr 은 접속한 클라이언트 정보를 반환받는다.

옵션 설정 없이 생성한 소켓의 기본 모드는 블로킹 모드이며 위 소켓을 블로킹 모드로 동작한다. 지금까지의 설명은 블로킹 소켓에 대한 설명이었으며 뒤쪽에서 블로킹 모드와 non블로킹 모드는 다시 공부한다.

여기서 대기 소켓(listenSock)은 다른 클라이언트의 접속을 처리해야 하므로 일반적으로(병렬 서버) 독립적인 루프를 실행하며 서버를 종료할 때까지 무한 반복한다. 지금은 간단한 TCP 서버 설명을 위한 코드이므로 하나의 클라이언트가 종료할 때까지 다른 클라이언트의 접속을 처리하지 않는다.

다음은 클라이언트의 다중 접속과 다중 처리가 가능한 TCP 클라이언트/서버의 기본 모형이다.



다음 코드는

```
printf("\n[TCP 서버] 클라이언트 접속: IP 주소=%s, 포트 번호=%d\n",  
      inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port) );  
접속한 클라이언트의 정보를 화면에 출력한다.
```

다음 코드는

```
while(1){  
    // 데이터 받기  
    retval = recv(
```

```

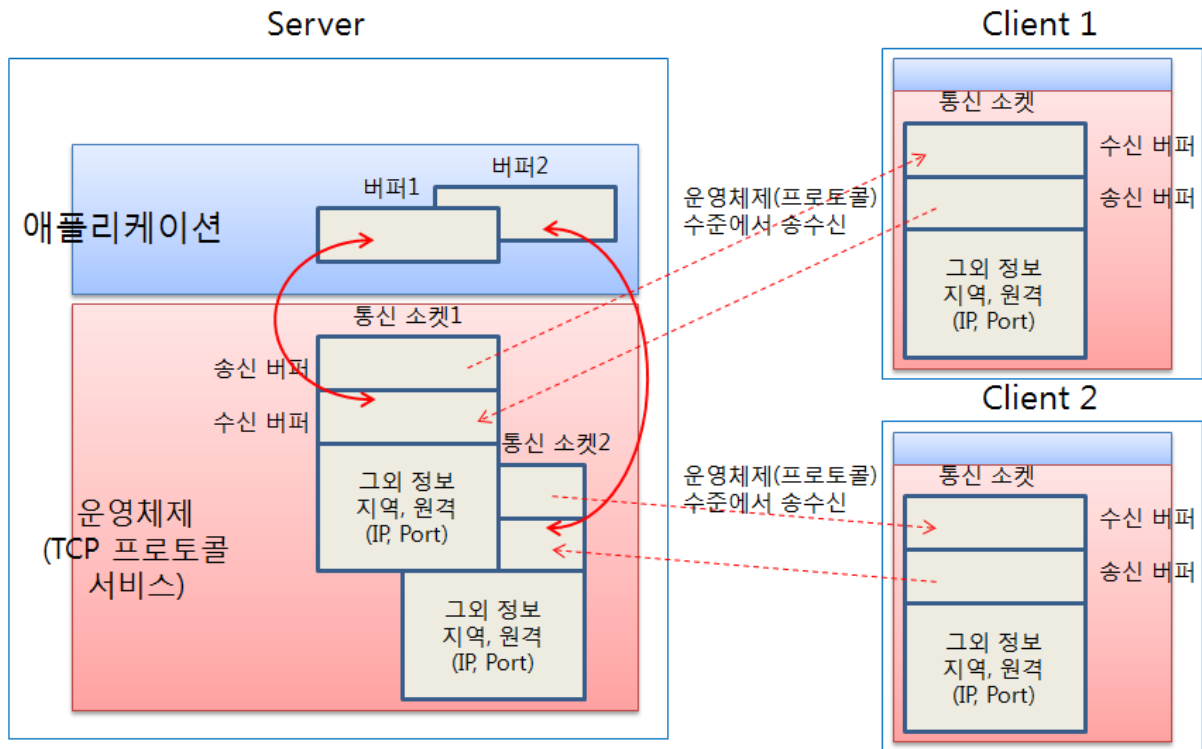
        clientSock, //통신소켓핸들
        buf, //받을 애플리케이션 버퍼
        BUFFER_SIZE, //수신 버퍼의 최대 크기
        0 //대부분 0 or MSG_PEEK와 MSG_OOB를 사용 가능
    );
    if(retval == SOCKET_ERROR)
    { //소켓 비정상 종료
        DisplayMessage();
        break;
    }
    else if(retval == 0)
    { //소켓 정상 종료
        DisplayMessage();
        break;
    }
    else{
        // 받은 데이터 출력
        buf[retval] = '\0';
        printf("[TCP 서버] IP 주소=%s, 포트 번호=%d의 받은 메시지:%s\n",
               inet_ntoa(clientaddr.sin_addr),
               ntohs(clientaddr.sin_port), buf );
    }
}

```

클라이언트가 보낸 데이터를 수신 버퍼에서 애플리케이션 버퍼로 복사한다. recv() 함수는 클라이언트가 보낸 데이터가 수신 버퍼에 도착할 때까지 블로킹되며 수신 버퍼에 데이터가 존재하면 애플리케이션 버퍼인 buf 에 저장하고(최대 BUFFER_SIZE) 반환한다. 이때 반환한 값은 애플리케이션 버퍼에 저장한 크기가 된다. recv() 함수도 블로킹 모드와 넌 블로킹 모드에 따라 조금 다르게 동작한다. 이 또한 뒤에서 공부한다.

printf("[TCP 서버] IP 주소=%s, 포트 번호=%d 의 받은 메시지:%s\n",
inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port), buf); 는
클라이언트가 전송한 문자열과 함께 클라이언트의 정보를 화면에 출력한다.

다음은 데이터를 송,수신할 때의 서버와 클라이언트의 TCP 버퍼와 애플리케이션 버퍼의 관계를 그림으로 표현한 것이다.



각각의 TCP 통신 소켓은 자신만의 송,수신 버퍼를 가지며 상대방과의 데이터 송, 수신은 운영체제(프로토콜) 수준에서 서비스 된다. recv() 함수는 운영체제로 관리되는 자신의 통신 소켓 수신 버퍼를 바라보고 있다가 데이터가 도착(자신과 연결된 클라이언트 소켓이 데이터를 보내면)하면 애플리케이션 버퍼로 데이터를 복사하는 역할을 담당할 뿐이다.

마지막으로 closesocket(clientSock)은 클라이언트와 접속을 종료하고 할당한 소켓 리소스를 운영체제에 반환한다.

다음은 TCP 클라이언트 코드다.

<코드>

[예제 02-03] TCPClient

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

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

```
#define BUFFERSIZE 1024
```

```
// 소켓 함수 오류 출력
```

```
void DisplayMessage()
```

```
{
```

```
    LPVOID pMsg;
```

```
    FormatMessage(
```

```
        FORMAT_MESSAGE_ALLOCATE_BUFFER | // 오류 메시지 저장 메모리를 내부에서 할당하라
```

```
        FORMAT_MESSAGE_FROM_SYSTEM, //운영체제로 부터 오류 메시지를 가져온다
```

```
        NULL,
```

```
        WSAGetLastError(), //오류 코드
```

```

    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), //언어(제어판 설정 언어)
    (LPTSTR)&pMsg, // 오류 메시지 outparam
    0, NULL);

    printf("%s\n", pMsg); // 오류 메시지 출력

    LocalFree(pMsg); // 오류 메시지 저장 메모리 반환
}

int main(int argc, char* argv[])
{
    int retval;

    WSADATA wsa;
    if(WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("윈도우 소켓 초기화 실패!\n");
        return -1;
    }

    // 클라이언트 소켓 생성
    SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
    if(sock == INVALID_SOCKET)
    {
        DisplayMessage();
        return -1;
    }

    SOCKADDR_IN serveraddr;
    ZeroMemory(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(40100);
    serveraddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    //서버에 접속 요청
    retval = connect(
        sock, //소켓핸들
        (SOCKADDR *)&serveraddr, //접속 서버 주소값
        sizeof(serveraddr) //주소값 크기
    ); // 통신 소켓(성공하면 자동으로 지역포트, 지역주소를 할당)
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return -1;
    }

    // 데이터 통신에 사용할 변수
    char buf[BUFFERSIZE];
    int len;

    // 서버와 데이터 통신
    while(1){
        // 데이터 입력
        ZeroMemory(buf, sizeof(buf));
        printf("[문자열 입력] ");
        if(fgets(buf, BUFFERSIZE, stdin) == NULL)
            break;

        // '\n' 문자 제거
        len = strlen(buf);
        if(buf[len-1] == '\n')

```

```

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

        // 서버에 데이터 보내기
        retval = send(
sock, //통신소켓핸들
buf, // 보낼 애플리케이션 버퍼
strlen(buf), //보낼 데이터 크기
0 //대부분 0 or MSG_DONTROUTE나 MSG_OOB를 사용 가능
);
        if(retval == SOCKET_ERROR)
        {
            DisplayMessage();
            break;
        }
        printf("[TCP 클라이언트] %d바이트를 전송\n", retval);
    }

    // 소켓 닫기
    closesocket(sock);

    WSACleanup();
    return 0;
}

```

</코드>

다음 코드는

```

SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
if(sock == INVALID_SOCKET)
{
    DisplayMessage();
    return -1;
}

```

클라이언트에서 사용한 소켓을 생성한다.

다음 코드는

```

SOCKADDR_IN serveraddr;
ZeroMemory(&serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(40100);
serveraddr.sin_addr.s_addr = inet_addr("127.0.0.1");

```

서버로 접속 요청하기 위한 구조체 초기화로 원격 IP 주소와 원격 포트 번호를 할당한다. 서버가 포트 번호를 40100으로 대기 소켓을 생성했으므로 같은 포트 번호를 할당하고 네트워크 바이트 정렬하기 위해 htons()를 호출한다. 서버의 IP 주소는 127.0.0.1로 같은 로컬 컴퓨터에 서버가 존재할 때 사용할 수 있으며 이 주소를 루프백 주소라 한다. 보통 네트워크 프로그래밍 테스트를 위해 사용되며 운영체제의 TCP 프로토콜 스택만을 사용하므로 네트워크 시스템이 마련되지 않은 컴퓨터에서도 이 주소를 사용하여 서버와 클라이언트 프로그램을 테스트할 수 있다. 우리도 루프백 주소를 사용할 것이다.

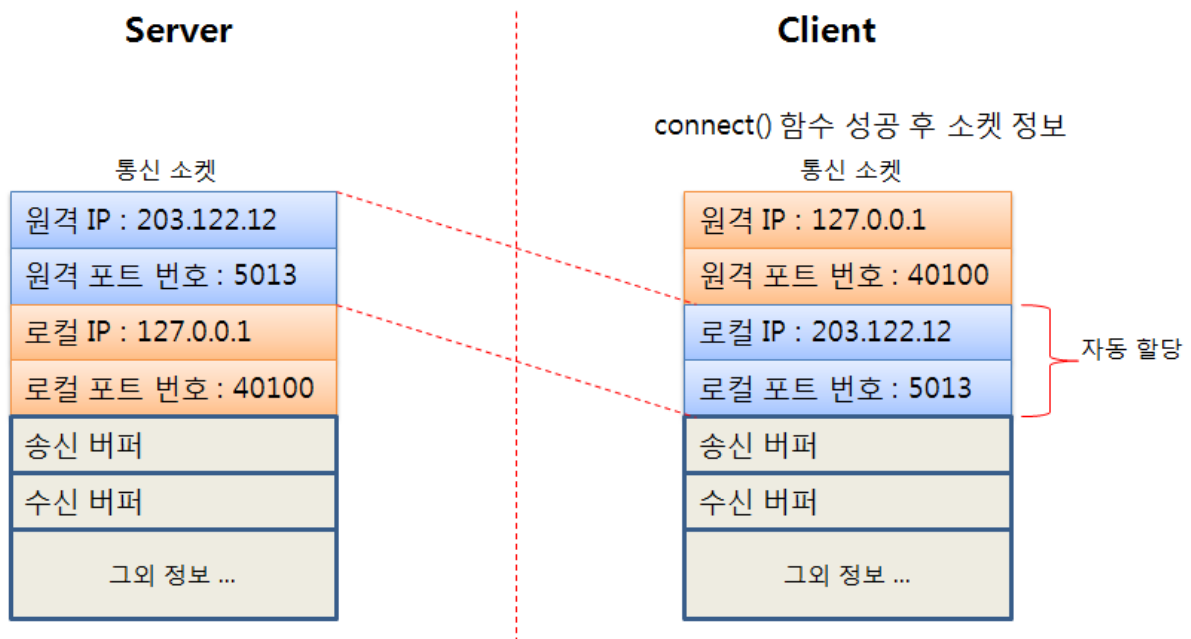
만약 서버의 실제 주소를 할당하려면 ipconfig 명령으로 IP 를 검색하여 설정하면 된다. inet_addr()은 문자열 주소를 정수 IP 주소로 변환하고 자동으로 네트워크 바이트 정렬을 수행한다.

다음 코드는

```
retval = connect(
    sock, //소켓핸들
    (SOCKADDR *)&serveraddr, //접속 서버 주소값
    sizeof(serveraddr) //주소값 크기
); // 통신 소켓(성공하면 자동으로 지역포트, 지역주소를 할당)
if(retval == SOCKET_ERROR)
{
    DisplayMessage();
    return -1;
}
```

TCP 클라이언트의 핵심 함수로 설정한 서버로 접속을 요청한다. 성공하면 자동으로 로컬 IP 와 로컬 Port 번호가 할당되며 이 소켓은 이제 서버와 데이터를 주고 받을 수 있다.

다음은 connect() 함수 성공 후의 서버와 클라이언트 소켓 정보를 나타낸 그림이다.



다음 코드는

```
while(1){
    // 데이터 입력
    ZeroMemory(buf, sizeof(buf));
    printf("[문자열 입력] ");
    if(fgets(buf, BUFFERSIZE, stdin) == NULL)
        break;
}
```

```

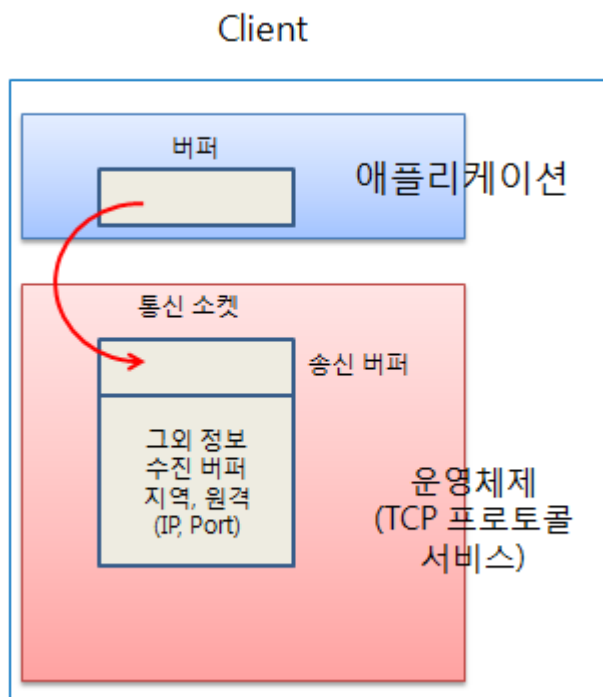
// 서버에 데이터 보내기
retval = send(
    sock, //통신소켓핸들
    buf, // 보낼 애플리케이션 버퍼
    strlen(buf), //보낼 데이터 크기
    0 //대부분 0 or MSG_DONTROUTE나 MSG_OOB를 사용 가능
);
if(retval == SOCKET_ERROR)
{
    DisplayMessage();
    break;
}
printf("[TCP 클라이언트] %d바이트를 전송\n", retval);
}

```

키보드에서 문자열을 입력받아 서버로 전송한다. send()는 애플리케이션 버퍼(buf)의 데이터를 송신 버퍼에 복사한다. send()는 strlen(buf) 크기의 모든 데이터를 송신 버퍼에 복사할 때까지 블로킹된다. 성공하면 복사한 크기를 반환한다.

마지막으로 문자열 입력이 없으면 closesocket(sock)을 호출하여 접속을 종료하고 소켓 리소스를 반환한다.

다음은 클라이언트 통신 소켓이 송신 버퍼로 애플리케이션 데이터를 복사하는 동작의 그림이다.



<장제목>

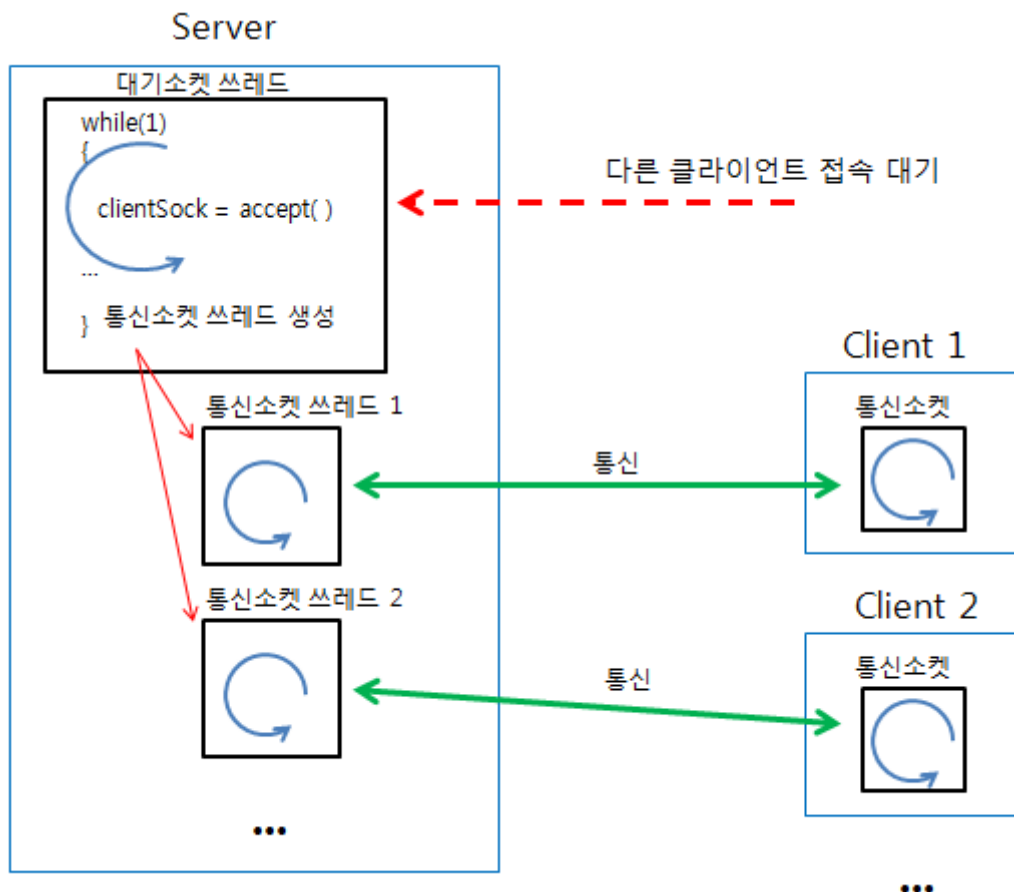
3. 스레드를 이용한 다중 에코 서버

</장제목>

서버는 일반적으로 여러 클라이언트와 통신(병렬 처리 서버라 한다)하므로 서버는 각각의 클라이언트와 통신하는 독립적인 처리 메커니즘을 구축해야 한다. 윈도우 소켓 라이브러리는 독립적인 여러 클라이언트를 처리할 수 있는 API 를 제공한다. 윈도우 소켓 라이브러리가 지원하는 API 를 사용하지 않고도 스레드를 사용하면 여러 클라이언트를 동시에 처리할 수 있는 서버를 구축할 수 있다.

이장에서는 스레드를 사용한 다중 클라이언트 에코 서버를 구축하고 여러 가지 고려 사항들을 살펴볼 것이다.

다음은 스레드를 사용한 에코 서버와 클라이언트를 표현한 그림이다.



서버는 여러 클라이언트의 접속을 대기하는 대기 스레드와 각각의 클라이언트와 통신을 담당하는 통신 스레드로 나뉜다.

<절제목>

01. 다중 에코 서버

</절제목>

다중 에코 서버는 크게 두 가지 스레드가 필요하다. 하나는 여러 클라이언트의 접속을 대기하고 통신 소켓을 생성하는 대기 소켓 스레드이고 또 하나는 클라이언트 통신을 담당하는 통신 소켓 스레드다.

다음은 클라이언트의 데이터를 에코하는 다중 스레드 서버 예제다.

<코드>

[예제 03-01] TCPThreadServer

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

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

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

```
#define BUFFERSIZE 1024
```

```
SOCKET listenSock;
```

```
// 소켓 함수 오류 출력
```

```
void DisplayMessage()
```

```
{
```

```
    LPVOID pMsg;
```

```
    FormatMessage(
```

```
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
```

```
        FORMAT_MESSAGE_FROM_SYSTEM,
```

```
        NULL,
```

```
        WSAGetLastError(),
```

```
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
```

```
        (LPTSTR)&pMsg, 0, NULL);
```

```
    printf("%s\n", pMsg);
```

```
    LocalFree(pMsg);
```

```
}
```

```
bool CreateListenSocket()
```

```
{
```

```
    int retval;
```

```
    // 대기 소켓 생성
```

```
    listenSock = socket(AF_INET, SOCK_STREAM, 0);
```

```
    if(listenSock == INVALID_SOCKET)
```

```
    {
```

```
        DisplayMessage();
```

```
        return false;
```

```
    }
```

```
    // 대기 소켓의 로컬 주소, 포트 설정
```

```
    SOCKADDR_IN serveraddr;
```

```
    ZeroMemory(&serveraddr, sizeof(serveraddr));
```

```
    serveraddr.sin_family = AF_INET;
```

```
    serveraddr.sin_port = htons(40100);
```

```
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
    retval = bind(listenSock, (SOCKADDR *)&serveraddr, sizeof(serveraddr));
```

```

    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }

    // 대기 소켓의 접속 대기 큐 생성 및 클라이언트 접속 대기
    retval = listen( listenSock, SOMAXCONN);
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }

    return true;
}
unsigned int WINAPI ComThread(void* pParam)
{
    SOCKET clientSock = (SOCKET) pParam;
    int recvByte;
    char buf[BUFFERSIZE];
    SOCKADDR_IN clientaddr;

    while(1)
    {
        // 데이터 받기
        recvByte = recv(clientSock, buf, BUFFERSIZE, 0);
        if(recvByte == SOCKET_ERROR)
        {
            //소켓 비정상 종료
            DisplayMessage();
            break;
        }
        else if(recvByte == 0)
        {
            //소켓 정상 종료
            DisplayMessage();
            break;
        }
        else
        {
            int addrlen = sizeof(clientaddr);
            int retval = getpeername(clientSock,
                (SOCKADDR *)&clientaddr, &addrlen);
            if(retval == SOCKET_ERROR)
            {
                DisplayMessage();
                continue;
            }

            // 받은 데이터 출력
            buf[recvByte] = '\0';
            printf("[TCP 서버] IP=%s, Port=%d의 메시지:%s\n",
                inet_ntoa(clientaddr.sin_addr),
                ntohs(clientaddr.sin_port),
                buf);

            // 클라이언트로 데이터 에코하기
            retval = send(clientSock, buf, recvByte, 0);
            if(retval == SOCKET_ERROR)
            {
                DisplayMessage();
                break;
            }
        }
    }

    // 통신 소켓 닫기
    closesocket(clientSock);
}

```

```

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

return 0;
}

unsigned int WINAPI ListenThread(void* pParam)
{
    while(1)
    {
        SOCKET clientSock;
        SOCKADDR_IN clientaddr;
        int addrlen;
        addrlen = sizeof(clientaddr);

        // 접속 대기
        clientSock = accept(listenSock, (SOCKADDR *)&clientaddr, &addrlen);
        if(clientSock == INVALID_SOCKET)
        {
            DisplayMessage();
            continue;
        }
        printf("\n[TCP 서버] 클라이언트 접속: IP 주소=%s, 포트 번호=%d\n",
            inet_ntoa(clientaddr.sin_addr),
            ntohs(clientaddr.sin_port));

        // 클라이언트와 독립적인 통신을 위한 Thread 생성
        unsigned int threadID;
        CloseHandle((HANDLE)_beginthreadex(0,0,ComThread, (void*) clientSock, 0,
            &threadID));
    }

    // 대기 소켓 닫기
    closesocket(listenSock);
}

int main(int argc, char* argv[])
{
    WSADATA wsa;

    if(WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("윈도우 소켓 초기화 실패!\n");
        return -1;
    }

    // 대기 소켓 초기화(socket()+bind()+listen())
    if( !CreateListenSocket( ) )
    {
        printf("대기 소켓 생성 실패!\n");
        return -1;
    }

    // 대기 쓰레드 종료를 기다림.
    unsigned int threadID;
    WaitForSingleObject((HANDLE)_beginthreadex(0,0, ListenThread, 0, 0,
        &threadID), INFINITE);

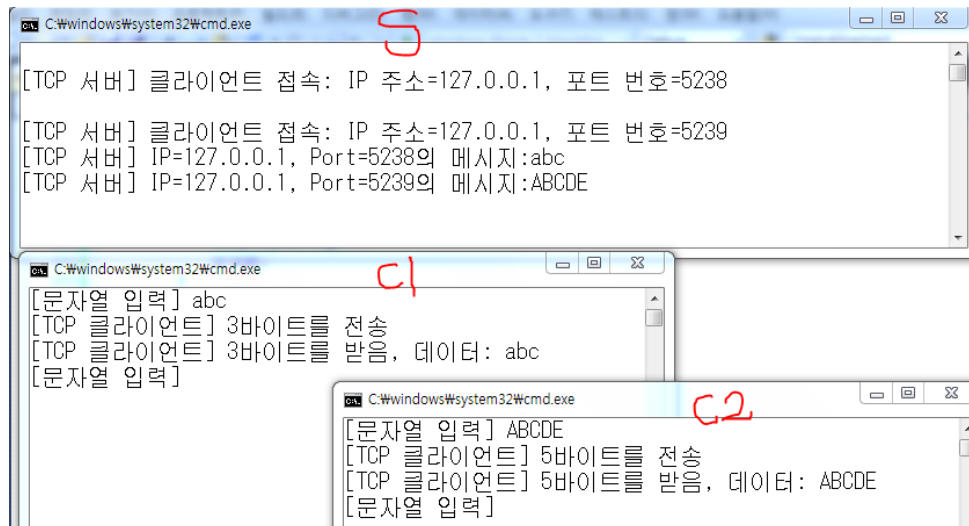
    WSACleanup();
    return 0;
}

```

</코드>

<결과>

[출력 결과]



</결과>

서버의 화면을 보면 포트 번호 5238과 5239의 두 클라이언트가 접속하여 각각 abc 문자열과 ABCDE 문자열을 주고받는다.

CreateListenSocket() 함수는 기존에 설명했던 내용처럼 대기 소켓을 생성하고 listen()까지 처리하는 함수다.

다음 코드는

```
recvByte = recv(clientSock, buf, BUFFERSIZE,0);
```

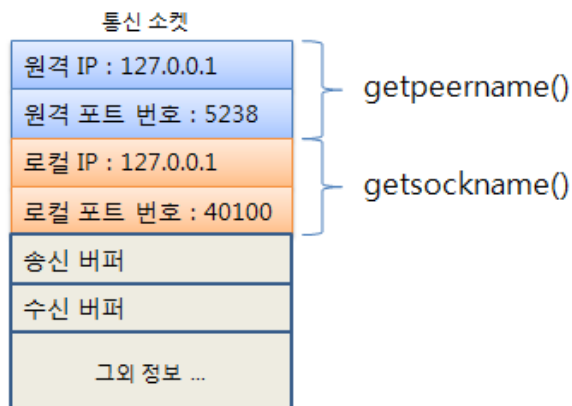
최대 BUFFERSIZE 까지 수신 버퍼에서 애플리케이션 버퍼(buf)로 데이터를 복사한다. 수신 버퍼에 클라이언트가 보낸 데이터가 없다면 recv() 함수는 블로킹되며 recv() 함수를 호출한 스레드는 Blocking 상태가 된다. 수신 버퍼에 데이터가 존재하면 BUFFERSIZE 이하의 데이터를 애플리케이션 버퍼로 복사한다. 이때 복사한 크기는 recvByte가 된다.

다음 코드는

```
int addrlen = sizeof(clientaddr);
int retval = getpeername(clientSock,
    (SOCKADDR *)&clientaddr, &addrlen);
if(retval == SOCKET_ERROR)
{
    DisplayMessage();
    continue;
}
```

소켓의 핸들로 클라이언트의 정보(IP, Port)를 반환하는 함수로 clientaddr 변수가 out parameter 로 사용되었다. 상대적인 함수로 getsockname()이 있으며 getpeername()이 소켓의 정보 중 원격 IP 와 원격 포트를 얻는데 반해 getsockname()은 소켓의 정보 중 지역 IP 와 지역 포트 번호를 얻는다. 접속할 때 클라이언트 정보를 보관한 후 사용해도 좋지만 소켓 정보에는 아래 그림과 같은 정보가 포함되어 있으므로 소켓 정보를 이용하는 방법도 자주 사용된다.

다음은 통신 소켓에서 정보를 얻는 함수를 표현한 그림이다.



다음 코드는

```
retval = send(clientSock, buf, recvByte, 0);
        if(retval == SOCKET_ERROR)
        {
            DisplayMessage();
            break;
        }
```

클라이언트에게서 받은 데이터를 다시 클라이언트로 에코하는 부분으로 받은 바이트 수 recvByte 만큼 전송한다. send()는 받은 데이터를 모두 전송할 때까지 블로킹된다.

다음 코드는

```
unsigned int threadID;
CloseHandle((HANDLE)_beginthreadex(0,0,ComThread, (void*) clientSock, 0,
&threadID));
```

클라이언트와 통신하기 위한 쓰레드를 생성하는 부분으로 _beginthreadex()를 사용한다. 인수로는 accept()가 반환한 통신 소켓을 전달하며 클라이언트와 통신에 사용할 수 있도록 한다. _beginthreadex()의 반환값은 쓰레드의 핸들이며 이 핸들을 사용하지 않기 때문에 생성 후 바로 핸들을 닫는다. 핸들을 닫는다고 쓰레드가 종료하는 것은 아니며 단지 Usage Count 를 감소할 뿐이다. 자세한 내용은 Window System Programming 부분을 참조하자.

다음 코드는

```
unsigned int threadID;
WaitForSingleObject((HANDLE)_beginthreadex(0,0, ListenThread, 0, 0,
&threadID), INFINITE);
```

클라이언트의 접속을 처리하기 위한 대기 소켓 쓰레드를 생성한다. 대기 소켓 쓰레드는 accept()를 호출하여 클라이언트의 접속이 이루어지면 통신 소켓 쓰레드를 생성한다.

여기서 WaitForSingleObject()는 대기 소켓 쓰레드가 종료할 때까지 주 쓰레드가 블로킹 상태에 놓이도록 한다. WaitForSingleObject()는 커널 오브젝트(쓰레드, 프로세스, 동기화 오브젝트 등)가 신호 상태가 되길 대기하는 함수다.

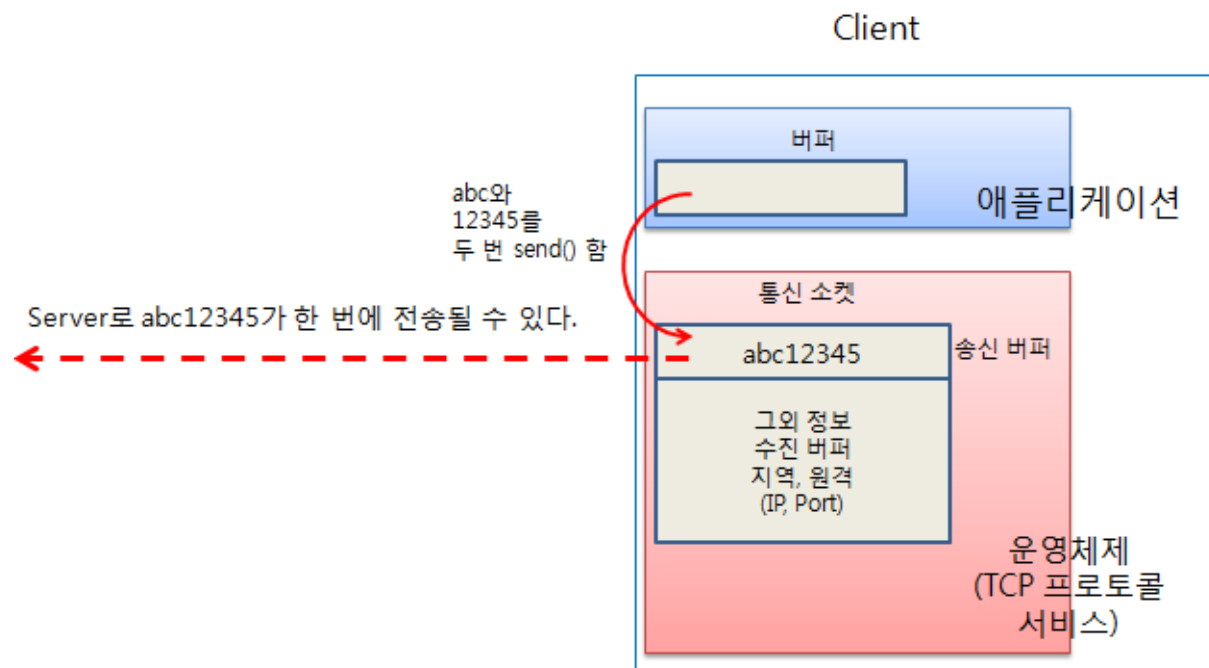
<절제목>

02. 다중 에코 클라이언트

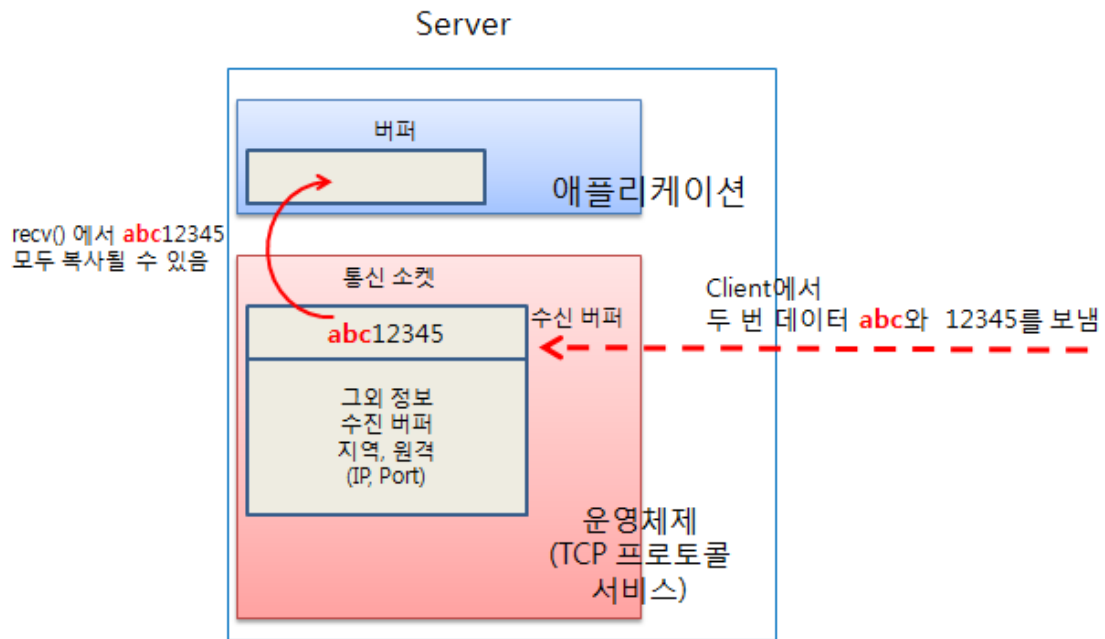
</절제목>

TCP 프로토콜 기반의 서버와 클라이언트의 구현 시 주의할 점은 TCP 는 ‘데이터의 경계가 없다’ 는 것이다. 데이터의 경계가 없다는 것은 공유 자원인 네트워크가 혼잡하여 TCP 송신 버퍼의 데이터가 보내지지 않고 있을 때 애플리케이션 데이터를 반복하여 TCP 송신 버퍼에 내려 보내면(복사) TCP 송신 버퍼에서 상대방 수신 버퍼로 송신 버퍼의 데이터를 한 번에 보내거나 TCP 수신 버퍼에 도착한 데이터를 애플리케이션 버퍼로 늦게 가져오는(복사) 경우 상대방에서 보낸 데이터가 한 번에 복사되어 올라올 수 있는 등의 상황이 발생할 수 있다는 것이다. 그래서 TCP 프로토콜 기반의 송,수신 데이터를 설계할 때는 위 사항을 고려하여 설계해야 한다.

다음은 네트워크가 혼잡하여 애플리케이션에서 send()가 두 번 이루어진 후 TCP 프로토콜 수준의 데이터 송신이 한 번 발생한 상황을 표현한 그림이다.



다음은 상대방에서 데이터를 두 번 이상 보낸 후 서버쪽 애플리케이션에서 recv()를 늦게 했을 때의 상황을 그림으로 표현한 것이다.

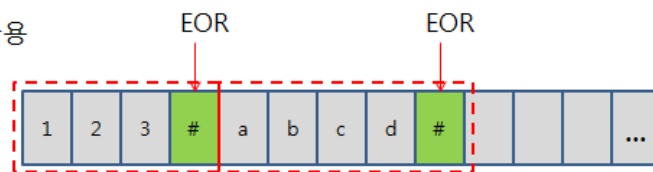


TCP 데이터를 설계하는 방법은 크게 세 가지가 있다.

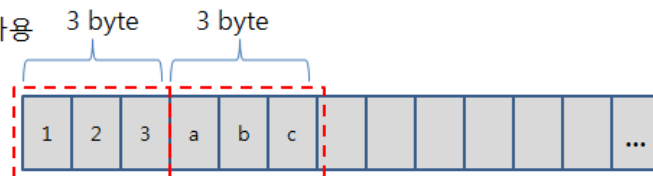
1. 경계 구분을 위한 특수한 표식(EOR: End Of Record)을 사용한다. 단순한 문자 전송 시스템이나 소형 기기 등의 통신에 사용한다.
2. 고정 길이 데이터를 송, 수신한다. 서버와 클라이언트는 항상 같은 길이의 데이터를 송수신한다. 사용하기 가장 간단하고 작은 데이터의 송, 수신 시스템에 사용된다.
3. 가변 길이 데이터를 송, 수신한다. 서버와 클라이언트는 헤더(실제 길이를 포함한 데이터 정보)를 고정길이로 송, 수신한 후 헤더에 포함된 실제 데이터의 길이를 확인하여 데이터의 송, 수신을 완료한다. 송, 수신 데이터가 크거나 가변적일 때 사용된다.

다음은 각 데이터 설계 방법을 그림으로 표현한 것이다.

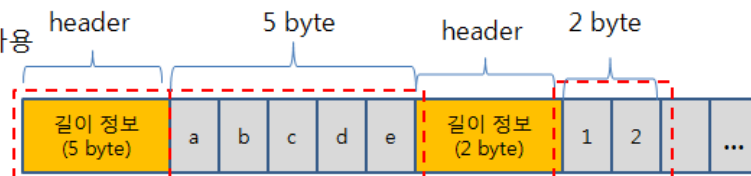
1. EOF 표식 사용



2. 고정 길이 사용 (3 byte)



3. 가변 길이 사용



다음은 에코 서버의 클라이언트 코드로 데이터를 고정 길이로 송, 수신하는 일반적인 코드로 구현한 예제다.

<코드>

[예제 03-01] TCPThreadClient

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

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

```
#define BUFFERSIZE 1024
```

```
// 소켓 함수 오류 출력
```

```
void DisplayMessage()
```

```
{
```

```
    LPVOID pMsg;
```

```
    FormatMessage(
```

```
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
```

```
        FORMAT_MESSAGE_FROM_SYSTEM,
```

```
        NULL,
```

```
        WSAGetLastError(),
```

```
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
```

```
        (LPTSTR)&pMsg, 0, NULL);
```

```
    printf("%s\n", pMsg);
```

```
    LocalFree(pMsg);
```

```
}
```

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

```
{
```

```
    int received;
```

```
    char *ptr = buf;
```

```
    int left = len;
```

```
    while(left > 0){
```

```
        received = recv(s, ptr, left, flags);
```

```
        if(received == SOCKET_ERROR)
```

```
            return SOCKET_ERROR;
```

```
        else if(received == 0)
```

```
            break;
```

```
        left -= received;
```



```

        ptr += received;
    }

    return (len - left);
}
int main(int argc, char* argv[])
{
    int retval;

    WSADATA wsa;
    if(WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("윈도우 소켓 초기화 실패!\n");
        return -1;
    }

    // 클라이언트 소켓 생성
    SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
    if(sock == INVALID_SOCKET)
    {
        DisplayMessage();
        return -1;
    }

    SOCKADDR_IN serveraddr;
    ZeroMemory(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(40100);
    serveraddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    //서버에 접속 요청
    retval = connect(
        sock, (SOCKADDR *)&serveraddr, sizeof(serveraddr));
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return -1;
    }

    // 데이터 통신에 사용할 변수
    char buf[BUFFERSIZE];
    int len;

    // 서버와 데이터 통신
    while(1)
    {
        // 데이터 입력
        ZeroMemory(buf, sizeof(buf));
        printf("[문자열 입력] ");
        if(fgets(buf, BUFFERSIZE-1, stdin) == NULL)
            break;

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

        // 서버에 데이터 보내기
        retval = send(sock, buf, strlen(buf), 0);
        if(retval == SOCKET_ERROR)
        {
            DisplayMessage();
            break;
        }
        printf("[TCP 클라이언트] %d바이트를 전송\n", retval);
    }
}

```

```

        // 데이터 받기
        retval = recvn(sock, buf, retval, 0);
        if(retval == SOCKET_ERROR || retval == 0)
        {
            DisplayMessage();
            break;
        }
        // 받은 데이터 출력
        buf[retval] = '\0';
        printf("[TCP 클라이언트] %d바이트를 받음, 데이터: %s\n", retval, buf);
    }

    // 소켓 닫기
    closesocket(sock);

    WSACleanup();
    return 0;
}
</코드>

```

다른 코드는 앞 장에서 공부했던 기본 TCP 코드와 비슷하고 중요 내용이 recvn() 함수다. recvn() 함수는 고정 크기의 데이터(지정한 크기의 데이터)를 완료할 때까지 recv() 함수를 계속 수행하는 함수다.

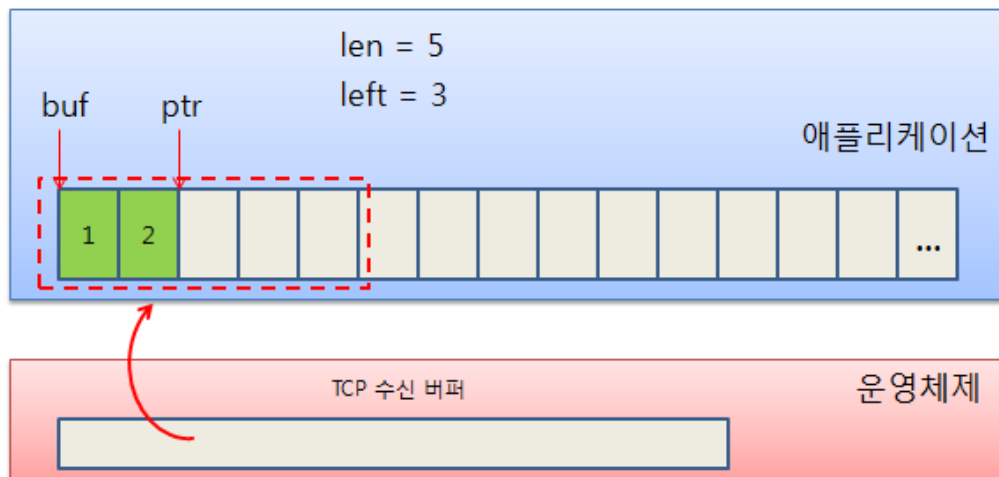
다음 코드의 s 는 통신 소켓의 핸들이며, buf 는 애플리케이션 버퍼의 시작 주소, len 은 고정 길이 크기이다. left 는 남아 있는 바이트 수이며 ptr 은 받을 애플리케이션 버퍼의 시작위치다. received 는 recv() 함수가 받은 바이트 수다.

```

int recvn(SOCKET s, char *buf, int len, int flags)
{
    int received;
    char *ptr = buf;
    int left = len;
    while(left > 0){
        received = recv(s, ptr, left, flags);
        if(received == SOCKET_ERROR)
            return SOCKET_ERROR;
        else if(received == 0)
            break;
        left -= received;
        ptr += received;
    }
    return (len - left);
}

```

다음 그림은 모두 5바이트(12345 데이터)를 받아야 하는 상황에서 첫 번째 recv() 함수로 2바이트(12 데이터)를 수신 버퍼에서 애플리케이션 버퍼로 복사하고 아직 3바이트(345 데이터)를 받기 위해 대기 중인 recvn() 함수를 표현한 것이다.



다음 코드는

```
retval = send(sock, buf, strlen(buf), 0 );
if(retval == SOCKET_ERROR)
{
    DisplayMessage();
    break;
}
printf("[TCP 클라이언트] %d바이트를 전송\n", retval);

// 데이터 받기
retval = recvn(sock, buf, retval, 0);
if(retval == SOCKET_ERROR || retval == 0)
{
    DisplayMessage();
    break;
}
```

클라이언트가 서버로 사용자가 입력한 문자열을 `send()` 함수로 보내고 보낸 바이트 수(`retval`)을 고정 길이 바이트로 다시 수신 하기 위해 `recvn()` 함수를 호출한다. `recvn()`은 클라이언트가 보낸 바이트 수를 정확히 받을 때까지 대기한다.

서버에서는 `recvn()`을 사용하지 않고 있는데 이것은 에코 서버로 설계되었기 때문에 몇 바이트가 도착하든 받은 바이트 수를 바로 바로 클라이언트로 에코하고 클라이언트가 보낸 바이트 수만큼 받기 위해 대기하는 방법으로 설계되었기 때문이다.

다른 사항은 특별이 전 코드와 다르지 않다.

<장제목>

4. Select 입출력 모델

</장제목>

윈도우 소켓은 효율적인 네트워크 통신을 위한 다양한 기능과 입출력 모델을 제공한다. 윈도우 소켓은 크게 두 가지 소켓 모드와 6 가지 입출력 모델을 지원한다.

소켓 모드는 두 가지 블로킹 소켓과 넌 블로킹 소켓 모드가 있으며 블로킹 소켓은 소켓 함수를 호출하면 소켓 함수의 목적을 완료할 때까지 반환하지 않고 대기(블로킹)하는 소켓으로 이때 해당(소켓 함수를 호출할 스레드) 스레드는 블로킹 상태(Blocked State)가 된다.

블로킹 소켓 함수의 목적과 반환 조건은

- `accept()` : 클라이언트가 접속할 때까지 블로킹되며 접속하면 반환한다.
- `send()`, `sendto()` : 송신 버퍼에 데이터를 모두 복사할 때까지 블로킹되며 모두 복사하면 반환한다.
- `recv()`, `recvfrom()` : 수신 버퍼에 데이터가 도착할 때까지 블로킹되며 데이터가 도착하면 데이터를 애플리케이션 버퍼로 복사하고 반환한다.

넌블로킹 소켓은 소켓 함수를 호출하면 소켓 함수의 목적이 완료되지 않아도 함수가 반환되며 해당 스레드는 블로킹되지 않고 계속 작업을 수행한다. 그래서 넌블로킹 소켓은 소켓 함수가 실제 목적을 완료했는지 아닌지를 구분할 수 있는 방법이 필요하다.

넌블로킹 소켓이 반환할 때 소켓 함수의 목적이 완료되지 않으면 소켓 함수는 오류를 반환하며 이때 소켓 함수가 목적을 완료하지 않아서 오류값을 반환했는지 실제 소켓 함수가 실패하여 오류값을 반환했는지를 판단하고 처리해야 한다. 이 오류값은 `WSAGetLastError()` 함수로 확인하며 소켓 함수가 자신의 목적을 완료하지 않고 반환하면 `WSAGetLastError()` 함수는 `WSAEWOULDBLOCK` 을 반환한다.

넌블로킹 소켓은 `ioctlsocket()` 함수를 호출하여 소켓 옵션을 변경하여 바꾼다.

최초로 소켓을 생성하면 블로킹 소켓이므로 `listenSock` 소켓을 먼저 생성하고 `ioctlsocket()` 을 사용하여 다음처럼 바꾼다.

```
// 대기 소켓 생성
listenSock = socket(AF_INET, SOCK_STREAM, 0);
if(listenSock == INVALID_SOCKET)
{
    DisplayMessage();
    return false;
}
```

```
// nonblocking socket으로 바꾼다.  
u_long on = 1;  
retval = ioctlsocket(listenSock, FIONBIO, &on);  
if(retval == SOCKET_ERROR)  
{  
    DisplayMessage();  
    return false;  
}
```

기본적으로 socket() 함수로 생성한 소켓은 초기에 블로킹 소켓으로 생성되며 생성한 대기 소켓으로 생성된(accept()) 통신 소켓도 블로킹 소켓으로 생성된다. 만약 대기 소켓이 nonblocking 소켓이라면 nonblocking 소켓으로 생성된 통신 소켓도 nonblocking 소켓으로 생성된다.

<절제목>

01. 년블로킹 에코 서버

</절제목>

다음은 기존 블로킹 에코 서버를 년블로킹 소켓으로 바꾸고 클라이언트와 통신하는 예제로 기본 코드와 다른 부분은 소켓을 생성하여 년블로킹 소켓으로 바꾸고 accept() 함수와 recv() 함수를 호출할 때 실패(소켓 함수의 목적을 완료하지 못한 경우)한 경우 소켓 함수를 재실행한다.

send() 함수도 송신 버퍼가 비어 있지 않은 경우 실패(소켓 함수의 목적을 완료하지 못한 경우)할 수 있지만 여기서는 그 처리를 생략하였다.

<코드>

[예제 04-01]

```
#include <winsock2.h>
#include <process.h>
#include <stdio.h>

#define BUFFERSIZE 1024

SOCKET listenSock;

// 소켓 함수 오류 출력
void DisplayMessage()
{
    LPVOID pMsg;
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        WSAGetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR)&pMsg,
        0, NULL);

    printf("%s\n", pMsg);

    LocalFree(pMsg);
}

bool CreateListenSocket()
{
    int retval;

    // 대기 소켓 생성
    listenSock = socket(AF_INET, SOCK_STREAM, 0);
    if(listenSock == INVALID_SOCKET)
    {
        DisplayMessage();
        return false;
    }

    // 년블로킹 소켓으로 바꾼다.
    u_long on = 1;
    retval = ioctlsocket(listenSock, FIONBIO, &on);
    if(retval == SOCKET_ERROR)
```

```

    {
        DisplayMessage();
        return false;
    }

    // 대기 소켓의 로컬 주소, 포트 설정
    SOCKADDR_IN serveraddr;
    ZeroMemory(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(40100);
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    retval = bind(listenSock, (SOCKADDR *)&serveraddr, sizeof(serveraddr));
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }

    // 대기 소켓의 접속 대기 큐 생성 및 클라이언트 접속 대기
    retval = listen(listenSock, SOMAXCONN);
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }

    return true;
}
unsigned int WINAPI ComThread(void* pParam)
{
    SOCKET clientSock = (SOCKET) pParam;
    int recvByte;
    char buf[BUFFERSIZE];
    SOCKADDR_IN clientaddr;

    while(1)
    {
        // 데이터 받기
        recvByte = recv(clientSock, buf, BUFFERSIZE, 0);
        if(recvByte == SOCKET_ERROR)
        {
            if(WSAGetLastError() != WSAEWOULDBLOCK)
            {
                //소켓 비정상 종료
                DisplayMessage();
                break;
            }

            // non블로킹 소켓 recv()의 목적이 완료되지 않음.
        }
        else if(recvByte == 0)
        {
            //소켓 정상 종료
            DisplayMessage();
            break;
        }
        else
        {
            int addrlen = sizeof(clientaddr);
            int retval = getpeername(clientSock,
                (SOCKADDR *)&clientaddr, &addrlen);
            if(retval == SOCKET_ERROR)
            {
                DisplayMessage();
                continue;
            }

            // 받은 데이터 출력
            buf[recvByte] = '\0';
            printf("[TCP 서버] IP=%s, Port=%d의 메시지:%s\n",

```

```

        inet_ntoa(clientaddr.sin_addr),
        ntohs(clientaddr.sin_port),
        buf);

    // 클라이언트로 데이터 에코하기
    retval = send(clientSock, buf, recvByte, 0);
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        break;
    }
}

// 통신 소켓 닫기
closesocket(clientSock);
printf("\n[TCP 서버] 클라이언트 종료: IP 주소=%s, 포트 번호=%d\n",
        inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port));

return 0;
}

unsigned int WINAPI ListenThread(void* pParam)
{
    while(1)
    {
        SOCKET clientSock;
        SOCKADDR_IN clientaddr;
        int addrlen;
        addrlen = sizeof(clientaddr);

        // 접속 대기
        clientSock = accept(listenSock, (SOCKADDR *)&clientaddr, &addrlen);
        if(clientSock == INVALID_SOCKET)
        {
            if(WSAGetLastError() != WSAEWOULDBLOCK)
                DisplayMessage();

            // nonblocking socket accept()의 목적이 완료되지 않음.
            continue;
        }

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

        // 클라이언트와 독립적인 통신을 위한 Thread 생성
        unsigned int threadID;
        CloseHandle((HANDLE)_beginthreadex(0, 0, ComThread, (void*) clientSock, 0,
                &threadID));
    }

    // 대기 소켓 닫기
    closesocket(listenSock);
}

int main(int argc, char* argv[])
{
    WSADATA wsa;

    if(WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("윈도우 소켓 초기화 실패!\n");
        return -1;
    }
}

```



```

// 대기 소켓 초기화(socket()+bind()+listen())
if( !CreateListenSocket( ) )
{
    printf("대기 소켓 생성 실패!\n");
    return -1;
}

// 대기 쓰레드 종료를 기다림.
unsigned int threadID;
waitForSingleObject((HANDLE)_beginthreadex(0,0, ListenThread, 0, 0,
    &threadID), INFINITE);

WSACleanup();
return 0;
}

```

</코드>

<결과>

[출력 결과]



</결과>

결과는 서버와 클라이언트가 데이터를 주고받는 방법은 블로킹 소켓과 같지만 소켓 함수(accept(), recv())가 목적을 완료하지 않으면 계속 루프를 실행하므로 쓰레드가 풀로 가동되며 CPU 사용률이 100%에 가깝다는 것을 알 수 있다. 년블록킹 소켓인 경우 목적이 완료하지 않더라도 쓰레드가 블로킹되지 않으므로 다른 작업을 진행할 수 있다는 장점이 있지만 소켓 함수의 작업을 재시도하기 위한 메커니즘이 필요하므로 코드가 복잡해질 수 있고 효율적으로 동작하도록 설계하는 것이 어렵다.

다음 코드는

```

u_long on = 1;
retval = ioctlsocket(listenSock, FIONBIO, &on);

```

```

if(retval == SOCKET_ERROR)
{
    DisplayMessage();
    return false;
}

```

대기 소켓을 년블로킹 소켓으로 바꾸며 이후 생성되는 모든 통신 소켓도 년 블로킹 소켓으로 생성된다.

다음 코드는

```

clientSock = accept(listenSock, (SOCKADDR *)&clientaddr, &addrLen);
if(clientSock == INVALID_SOCKET)
{
    if(WSAGetLastError() != WSAEWOULDBLOCK)
        DisplayMessage();
    // 년블로킹 소켓 accept()의 목적이 완료되지 않음.
    continue;
}

```

accept()가 목적을(클라이언트 접속) 완료하지 못하면 소켓 에러(INVALID_SOCKET)가 발생하고 이때 WSAGetLastError()의 값이 WSAEWOULDBLOCK 이 된다. 그리고 다시 accept() 함수를 실행하여 클라이언트 접속을 재확인한다. (이 부분의 작성을 효율적으로 하는 것이 년블로킹 서버의 핵심이다.)

다음 코드는

```

recvByte = recv(clientSock, buf, BUFFERSIZE, 0);
if(recvByte == SOCKET_ERROR)
{
    if(WSAGetLastError() != WSAEWOULDBLOCK)
    {
        //소켓 비정상 종료
        DisplayMessage();
        break;
    }
    // 년블로킹 소켓 recv()의 목적이 완료되지 않음.
}

```

recv() 함수가 목적을 완료(수신 버퍼에 데이터가 도착하면 애플리케이션 버퍼로 복사)하지 않으면 SOCKET_ERROR 가 반환되고 WSAGetLastError()의 값이 WSAEWOULDBLOCK 이 된다. 그리고 recv() 함수를 실행하여 수신 버퍼에 데이터 도착을 확인한다.

위 코드들은 년블로킹 소켓의 함수 동작을 설명할 뿐 상당히 비효율적으로 동작한다. 년블로킹 소켓에서 중요한 것은 쓰레드가 블록되지 않는다는 장점을 활용하면서 입출력 함수들이 효율적으로 설계되도록 구성하는 것이다.

<절제목>

02. Select 모델

</절제목>

Select 모델의 핵심 함수는 `select()` 함수다. 쓰레드를 사용한 예코 서버에서는 클라이언트와 통신하는 독립적인 작업을 각각의 쓰레드를 생성하여 수행했지만 Select 모델을 사용하면 하나의 쓰레드로 다중 클라이언트 예코 서버를 생성할 수 있으며 `select()` 함수를 사용하면 블로킹, 년블로킹 소켓에서 모두 사용할 수 있다.

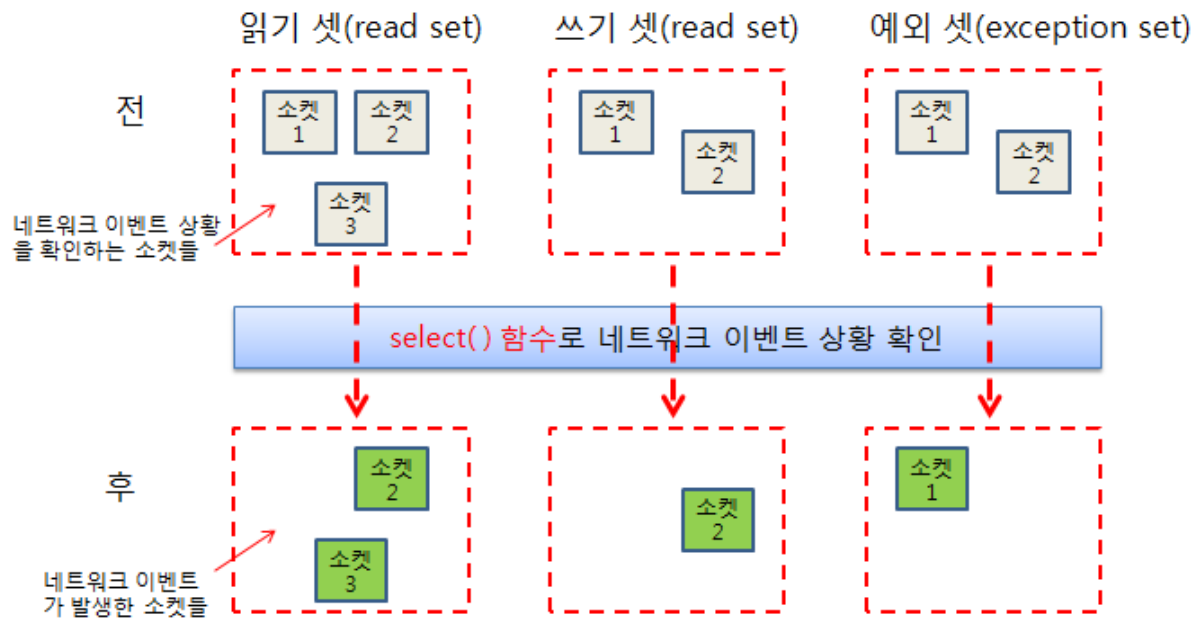
Select 모델은 `select()` 함수로 네트워크 이벤트 상황(클라이언트 접속, 클라이언트 종료, 데이터 수신, 데이터 송신 버퍼 빔 등)을 확인 할 수 있기 때문에 그 네트워크 이벤트 상황에 맞춰 소켓 함수를 적절하고 호출하고 동작할 수 있다. 한마디로 블로킹 소켓은 네트워크 이벤트 상황에 맞는 소켓 함수를 호출하여 소켓 함수가 블로킹되지 않는다. 네트워크 이벤트 상황에 맞는 소켓 함수를 정확히 호출하므로 년블로킹 소켓은 실패(목적 완료 실패)하지 않는다.

Select 모델은 소켓 셋(socket set)이라 부르는 세 개의 읽기 셋, 쓰기 셋, 예외 셋이 존재하며 `select()` 함수는 이 세 개의 셋을 통해 네트워크 이벤트 상황을 애플리케이션이 확인 할 수 있다.

다음은 Select 모델에서 각 셋에 소켓을 전달하여 네트워크 이벤트 상황을 확인하는 절차를 표현한 그림이다. 읽기 셋에는 소켓1, 소켓2, 소켓3을 확인하고자 하며 쓰기 셋은 소켓1, 소켓2를 확인하고자 하고 예외 셋은 소켓1, 소켓2를 확인하고자 한다.

이때 읽기 셋의 소켓2와 소켓3은 읽기 이벤트가 발생되었으며 쓰기 셋의 소켓2와 예외 셋의 소켓1에 이벤트가 발생한 상황이다.

이제 각 발생한 소켓의 이벤트 상황에 맞는 소켓 함수들을 호출하여 작업을 수행한다.



각 소켓 함수와 네트워크 상황은 아래와 같다.

- 읽기 셋은 클라이언트가 접속했을 때 accept() 호출, 데이터가 수신 버퍼에 도착했을 때 recv(), recvfrom() 호출, 연결 종료이면 recv(), recvfrom() 호출하여 0 반환된다.
- 쓰기 셋은 송신 버퍼가 여유 공간이 생겼으므로 send(), sendto()를 호출한다.
- 예외 셋은 00B 데이터가 도착했다. recv(), recvfrom()으로 00B 데이터를 수신한다. 잘 사용되지 않는다.

Select 모델은 버클리 소켓과 호환 가능한 모델이며 윈도우 소켓에서는 많이 사용되지 않는다.

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

- nfds 는 유닉스, 리눅스등에서 사용되며 윈도우 소켓은 사용되지 않는다.
- readfds, writefds, exceptfds 는 각각 읽기 셋, 쓰기 셋, 예외 셋을 나타내며 사용하지 않으면 NULL 을 사용한다.
- timeout 은 타임아웃을 설정한다.

```
struct timeval {
    long tv_sec; // 초
    long tv_usec; // 마이크로 초
```

}; 의 구조체 변수로 NULL 이면 네트워크 이벤트 상황이 발생할 때까지 무한정 대기한다.
변수가 0 으로 초기화되어 전달되면 네트워크 이벤트를 확인하고 바로 반환한다.

다음은 Select 모델을 기반으로 에코 서버를 구현한 것으로 select() 함수로 네트워크 이벤트 상황을 확인하고 상황에 맞게 클라이언트 접속처리의 accept()를 호출하고 데이터 송, 수신을 위한 send(), recv()를 호출한다.

(예제는 ITC00KB00K 윈도우 네트워크 프로그래밍 김선우님의 예제 코드를 참고하여 작성되었다.)

<코드>

[예제 04-02] SelectTCPServer

```
#include <winsock2.h>
#include <process.h>
#include <stdio.h>

#define BUFFERSIZE 1024
// 소켓 정보 저장을 위한 구조체

struct SOCKETINFO
{
    SOCKET sock;
    char buf[BUFFERSIZE];
    int recvbytes;
    int sendbytes;
};

SOCKET listenSock; //대기 소켓 핸들
int nTotalSockets = 0; // 통신 소켓의 개수
SOCKETINFO *SocketInfoArray[FD_SETSIZE];

// 소켓 함수 오류 출력
void DisplayMessage()
{
    LPVOID pMsg;
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        WSAGetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR)&pMsg,
        0, NULL);

    printf("%s\n", pMsg);

    LocalFree(pMsg);
}

bool CreateListenSocket()
{
    int retval;

    // 대기 소켓 생성
    listenSock = socket(AF_INET, SOCK_STREAM, 0);
    if(listenSock == INVALID_SOCKET)
    {
        DisplayMessage();
        return false;
    }

    // non블로킹 소켓으로 바꾼다.
    u_long on = 1;
    retval = ioctlsocket(listenSock, FIONBIO, &on);
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }
}
```

```

    }

    // 대기 소켓의 로컬 주소, 포트 설정
    SOCKADDR_IN serveraddr;
    ZeroMemory(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(40100);
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    retval = bind(listenSock, (SOCKADDR *)&serveraddr, sizeof(serveraddr));
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }

    // 대기 소켓의 접속 대기 큐 생성 및 클라이언트 접속 대기
    retval = listen(listenSock, SOMAXCONN);
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }

    return true;
}

// 소켓 정보를 추가한다.
BOOL AddSocketInfo(SOCKET sock)
{
    // FD_SETSIZE - 연결 대기 소켓
    if(nTotalSockets >= (FD_SETSIZE-1)){
        printf("[오류] 소켓 정보를 추가할 수 없습니다!\n");
        return FALSE;
    }

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

    ptr->sock = sock;
    ptr->recvbytes = 0;
    ptr->sendbytes = 0;
    SocketInfoArray[nTotalSockets++] = ptr;

    return TRUE;
}

// 소켓 정보를 삭제한다.
void RemoveSocketInfo(int nIndex)
{
    SOCKETINFO *ptr = SocketInfoArray[nIndex];

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

    closesocket(ptr->sock);
    delete ptr;

    for(int i=nIndex; i<nTotalSockets; i++){
        SocketInfoArray[i] = SocketInfoArray[i+1];
    }
}

```

```

        nTotalSockets--;
    }

    unsigned int WINAPI WorkerThread(void* pParam)
    {
        int retval;
        FD_SET rset;
        FD_SET wset;
        SOCKET clientSock;
        SOCKADDR_IN clientaddr;
        int addrlen;

        while(1)
        {
            // 소켓 셋 초기화
            FD_ZERO(&rset);
            FD_ZERO(&wset);
            FD_SET(listenSock, &rset); //대기 소켓을 rset에 등록
            for(int i=0; i<nTotalSockets; i++)
            {
                if(SocketInfoArray[i]->recvbytes >
                    SocketInfoArray[i]->sendbytes)
                    FD_SET(SocketInfoArray[i]->sock, &wset);
                else
                    FD_SET(SocketInfoArray[i]->sock, &rset); //

            }

            // select() 대기
            retval = select(0, &rset, &wset, NULL, NULL);
            if(retval == SOCKET_ERROR)
            {
                DisplayMessage();
                break;
            }

            // 대기 소켓에 클라이언트 접속이 요청되었나?
            if(FD_ISSET(listenSock, &rset)){
                addrlen = sizeof(clientaddr);
                clientSock = accept(listenSock, (SOCKADDR *)&clientaddr,
                                    &addrlen);
                if(clientSock == INVALID_SOCKET){
                    if(WSAGetLastError() != WSAEWOULDBLOCK)
                        DisplayMessage();
                }
                else{
                    printf("\n[TCP 서버] 클라이언트 접속: IP 주소=%s, 포트\n",
                           번호=%d\n",
                           inet_ntoa(clientaddr.sin_addr),
                           ntohs(clientaddr.sin_port));

                    // 소켓 정보 추가
                    if(AddSocketInfo(clientSock) == FALSE){
                        printf("[TCP 서버] 클라이언트 접속을\n",
                               해제합니다!\n");
                        closesocket(clientSock);
                    }
                }
            }

            // 데이터 송,수신 가능한가?
            for(int i=0; i<nTotalSockets; i++)
            {
                SOCKETINFO *ptr = SocketInfoArray[i];
                // 데이터가 수신 버퍼에 도착했다.
                if(FD_ISSET(ptr->sock, &rset)){

```

```

        retval = recv(ptr->sock, ptr->buf, BUFFERSIZE, 0);
        if(retval == SOCKET_ERROR){
            if(WSAGetLastError() != WSAEWOULDBLOCK){
                DisplayMessage();
                RemoveSocketInfo(i);
            }
            continue;
        }
        else if(retval == 0){
            RemoveSocketInfo(i);
            continue;
        }
        ptr->recvbytes = retval;
        // 받은 데이터 출력
        addrlen = sizeof(clientaddr);
        getpeername(ptr->sock, (SOCKADDR *)&clientaddr,
            &addrlen);
        ptr->buf[retval] = '\0';

        printf("[TCP 서버] IP=%s, Port=%d의 메시지:%s\n",
            inet_ntoa(clientaddr.sin_addr),
            ntohs(clientaddr.sin_port),
            ptr->buf);
    }
    // 송신 버퍼 여기 공간이 남아 데이터를 보낼 준비가 되었다.
    if(FD_ISSET(ptr->sock, &wset)){
        retval = send(ptr->sock, ptr->buf + ptr->sendbytes,
            ptr->recvbytes - ptr->sendbytes, 0);
        if(retval == SOCKET_ERROR){
            if(WSAGetLastError() != WSAEWOULDBLOCK){
                DisplayMessage();
                RemoveSocketInfo(i);
            }
            continue;
        }
        ptr->sendbytes += retval;
        if(ptr->recvbytes == ptr->sendbytes){
            ptr->recvbytes = ptr->sendbytes = 0;
        }
    }
}

// 대기 소켓 닫기
closesocket(listenSock);

return 0;
}

int main(int argc, char* argv[])
{
    WSADATA wsa;

    if(WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("윈도우 소켓 초기화 실패!\n");
        return -1;
    }

    // 대기 소켓 초기화(socket()+bind()+listen())
    if( !CreateListenSocket( ) )
    {
        printf("대기 소켓 생성 실패!\n");
        return -1;
    }
}

```

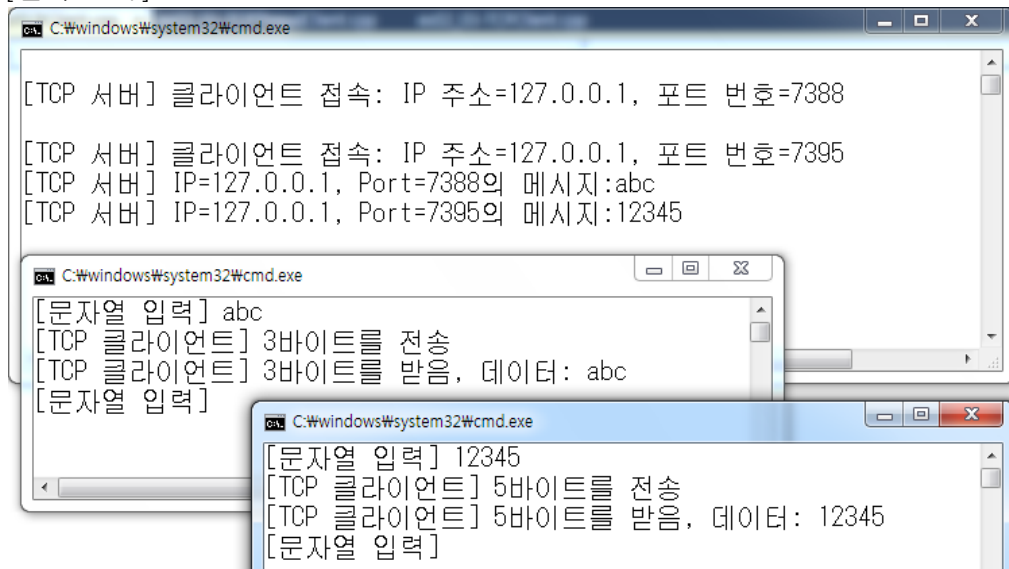


```

// 대기 쓰레드 종료를 기다림.
unsigned int threadID;
WaitForSingleObject((HANDLE)_beginthreadex(0,0, WorkerThread, 0, 0,
&threadID), INFINITE);

WSACleanup();
return 0;
}
</코드>
<결과>
[출력 결과]

```



</결과>

쓰레드를 사용한 에코 서버는 접속한 클라이언트의 개수만큼 쓰레드를 생성하지만 위 코드는 하나의 쓰레드(WorkerThread)를 사용하여 작업을 처리한다.

다음 코드는

```

struct SOCKETINFO
{
    SOCKET sock;
    char buf[BUFFERSIZE];
    int recvbytes;
    int sendbytes;
};

```

클라이언트의 정보를 보관하기 위한 구조체로 서버가 받은 데이터는 buf 에 저장하며 받은 바이트는 recvbytes 에 다시 클라이언트로 에코한 바이트는 sendbytes 를 사용한다.

다음 코드는

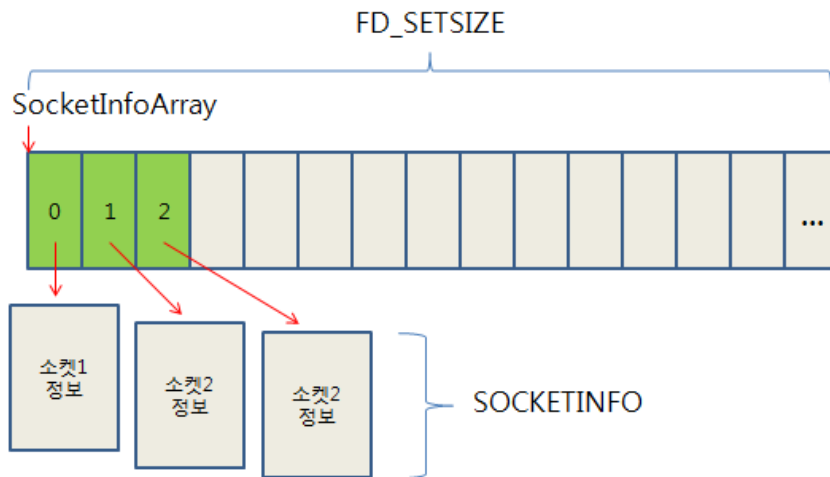
```

int nTotalSockets = 0; // 통신 소켓의 개수
SOCKETINFO *SocketInfoArray[FD_SETSIZE];

```

접속한 클라이언트의 정보를 보관하기 위한 배열로 최대 FD_SETSIZE 개수만큼 생성한다.

다음은 세 대의 클라이언트가 접속했을 때의 SocketInfoArray 의 배열을 표현한 그림이다.



다음 두 함수는

```

BOOL AddSocketInfo(SOCKET sock);
void RemoveSocketInfo(int nIndex);

```

SocketInfoArray 배열에 소켓 정보를 추가하고 제거하기 위한 함수다.

다음 코드는 블로킹 소켓을 넌블로킹 소켓으로 바꾼다.

```

u_long on = 1;
retval = ioctlsocket(listenSock, FIONBIO, &on);
if(retval == SOCKET_ERROR)
{
    DisplayMessage();
    return false;
}

```

다음 코드는

```

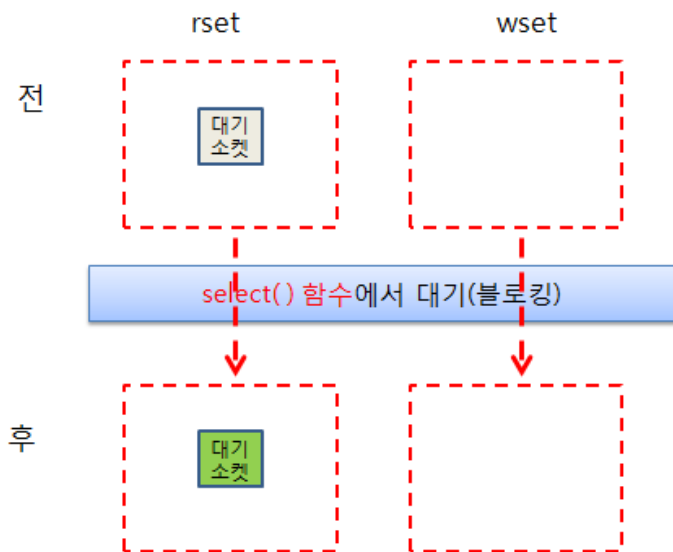
FD_ZERO(&rset);
FD_ZERO(&wset);

FD_SET(listenSock, &rset);

```

rset 과 wset 을 초기화하고 대기 소켓을 rset 에 등록한다. 클라이언트 접속이 아직 존재하지 않으면 아래 그림과 같이 rset 에 대기 소켓만 등록된다.

넌블로킹 소켓이더라도 소켓 함수 중 select() 함수만 블로킹되며 select() 함수는 현재 클라이언트 접속에서만 반환한다.



다음 코드는

```
for(int i=0; i<nTotalSockets; i++)
{
    if(SocketInfoArray[i]->recvbytes >
        SocketInfoArray[i]->sendbytes)
        FD_SET(SocketInfoArray[i]->sock, &wset);
    else
        FD_SET(SocketInfoArray[i]->sock, &rset); //
}
```

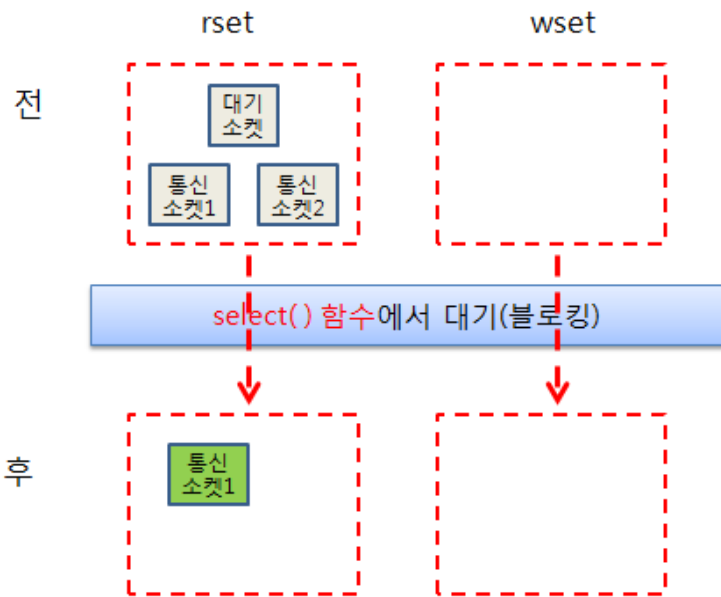
클라이언트 소켓이 존재하면 받은 데이터가 존재할 경우 데이터를 클라이언트로 보내기 위해 통신 소켓을 wset 에 등록하고 받은 데이터가 존재하지 않는다면 통신 소켓을 rset 에 등록한다.

다음 코드는

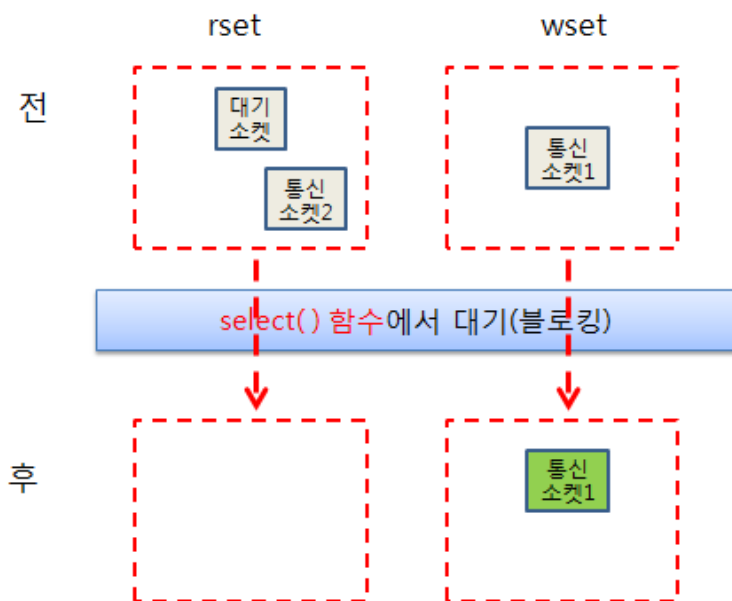
```
retval = select(0, &rset, &wset, NULL, NULL);
if(retval == SOCKET_ERROR)
{
    DisplayMessage();
    break;
}
```

Select 모델에서 가장 핵심이 되는 코드로 rset 과 wset 에 등록된 소켓의 이벤트 상황을 확인하기 위해 블로킹 상태에 놓인다. 예외 셋은 사용하지 않으며 타임아웃이 NULL 이므로 네트워크 이벤트 상황이 발생할 때까지 무한정 기다린다.

다음 그림은 클라이언트가 두 대 접속했을 때의 rset 과 wset 의 select() 모형을 표현한 것으로 통신 소켓 1 의 수신 버퍼의 데이터가 도착하면 select() 함수가 반환하고 통신 소켓 1 의 recv()를 호출하면 데이터 수신(애플리케이션 버퍼로 복사)이 가능하다.



다음 그림은 통신 소켓 1 이 데이터를 수신하고 다시 데이터를 보내기 위해 select()함수를 호출할 때를 표현한 그림으로 통신 소켓 1 은 데이터를 보내기 위해 wset 에 등록한다. 통신 소켓 1 은 최초의 상태로 송신 버퍼가 빈 상태이며 데이터를 보낼 수 있으므로 select() 함수는 바로 반환하여 그림과 같이 wset 에 통신 소켓 1 만 남게 된다.



다음 코드는

```
if(FD_ISSET(listenSock, &rset)){
    addrlen = sizeof(clientaddr);
    clientSock = accept(listenSock, (SOCKADDR *)&clientaddr,
        &addrlen);
    if(clientSock == INVALID_SOCKET){
        if(WSAGetLastError() != WSAEWOULDBLOCK)
            DisplayMessage();
    }
    else{
```

```

printf("\n[TCP 서버] 클라이언트 접속: IP 주소=%s, 포트
        번호=%d\n",
        inet_ntoa(clientaddr.sin_addr),
        ntohs(clientaddr.sin_port));
// 소켓 정보 추가
if(AddSocketInfo(clientSock) == FALSE){
    printf("[TCP 서버] 클라이언트 접속을
        해제합니다!\n");
    closesocket(clientSock);
}
}
}

```

클라이언트가 접속했는지 FD_ISSET (listenSock, &rset) 으로 rset 의 상태를 확인하여 클라이언트가 접속했다면 accept() 함수로 클라이언트 접속을 수락하고 통신 소켓을 SocketInfoArray 에 추가한다.

다음 코드는

```

if(FD_ISSET(ptr->sock, &rset)){
    retval = recv(ptr->sock, ptr->buf, BUFFERSIZE, 0);
    if(retval == SOCKET_ERROR){
        if(WSAGetLastError() != WSAEWOULDBLOCK){
            DisplayMessage();
            RemoveSocketInfo(i);
        }
        continue;
    }
    ...
    ptr->recvbytes = retval;
    ...
}

```

select() 함수가 네트워크 이벤트 상황이 발생하여 반환하면 어떤 클라이언트 소켓이 데이터를 수신할 준비가 되었는지 판단하고 그에 맞는 처리를 하는 코드다. 특히 데이터 수신부의 ptr->recvbytes = retval; 는 받은 데이터의 크기를 통신 소켓 구조체에 할당한다.

다음 코드는

```

if(FD_ISSET(ptr->sock, &wset)){
    retval = send(ptr->sock, ptr->buf + ptr->sendbytes,
        ptr->recvbytes - ptr->sendbytes, 0);
    if(retval == SOCKET_ERROR){
        if(WSAGetLastError() != WSAEWOULDBLOCK){
            DisplayMessage();
            RemoveSocketInfo(i);
        }
        continue;
    }
    ...
}

```

통신 소켓이 데이터를 보낼 준비가 되었다면 아직 덜 보낸 데이터부분을 보낼 수 있도록 하는 코드다.

특히, send(ptr->sock, ptr->buf + ptr->sendbytes, ptr->recvbytes - ptr->sendbytes, 0);

코드는 보낸 부분을 감안하여 보낼 수 있도록 구성한 코드다. 다음 코드에서 보낸 바이트 수를

다음 코드처럼 누적하고

```
ptr->sendbytes += retval;
```

아직 덜 보냈다면 보낼 준비가 될 때까지 select()에서 대기한다.

모두 보냈다면 다음 코드에서 초기화하여 다시 받을 준비를 한다.

```
if(ptr->recvbytes == ptr->sendbytes){  
    ptr->recvbytes = ptr->sendbytes = 0;  
}
```

클라이언트 코드는 쓰레드를 사용한 에코 서버/클라이언트 코드를 사용하여 테스트한다.

<장제목>

5. WSAAsyncSelect 입출력 모델

</장제목>

WSAAsyncSelect 모델은 소켓 프로그램이 윈도우 메시지 기반의 애플리케이션과 잘 어울려 동작하도록 설계된 입출력 모델이다. WSAAsyncSelect 모델을 사용하기 위해서는 하나 이상의 윈도우 객체가 필요하며 네트워크에서 발생하는 소켓 이벤트를 윈도우 메시지로 다룰 수 있어 윈도우 프로시저에서 일반 윈도우 메시지와 함께 네트워크 이벤트 처리가 가능하다.

<절제 목>

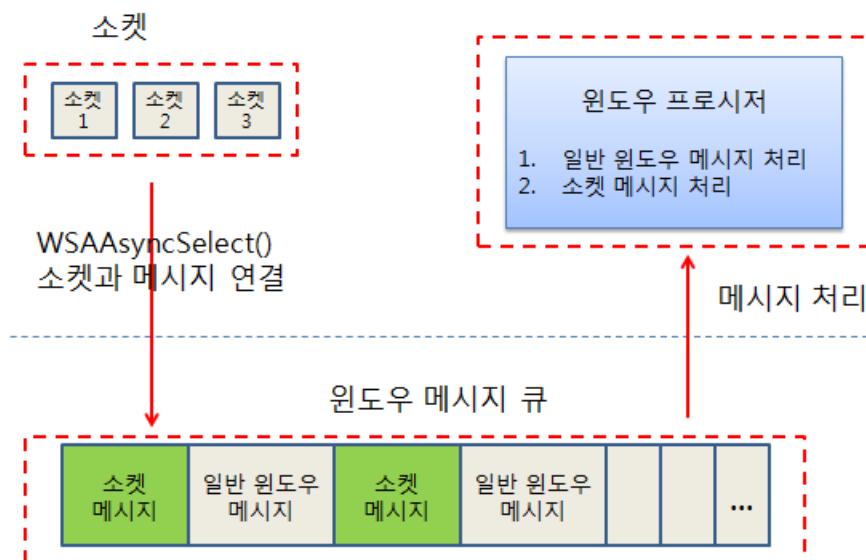
01. WSAAsyncSelect 모델의 이해

</절제 목>

WSAAsyncSelect 모델을 사용하는 방법은 간단한다. 크게 아래와 같은 단계로 진행된다.

1. 소켓을 윈도우 메시지와 연결한다.
2. 소켓에 네트워크 이벤트가 발생하면 연결된 윈도우 메시지 윈도우 프로시저에 전달된다.
3. 윈도우 프로시저에서 네트워크 이벤트를 확인하고 이벤트에 맞는 적절한 소켓 함수를 호출한다.

다음 그림은 이 세 단계를 그림으로 표현한 것이다.



이때, 소켓과 윈도우 메시지를 연결하기 위해 WSAAsyncSelect() 함수가 사용된다.

int WSAAsyncSelect(SOCKET s, HWND, hwnd, unsigned int wMsg, long lEvent);의

- s 는 연결하는 소켓의 핸들이다.
- hwnd 는 메시지를 처리할 프로시저의 윈도우 핸들이다.
- wMsg 는 연결할 사용자 정의 메시지이다.
- lEvent 는 소켓의 네트워크 이벤트다.

반환값은 성공 시 0 이며 실패는 SOCKET_ERROR 를 반환한다.

다음 코드는

```
WSAAsyncSelect(listenSock, hwnd, WM_SOCKET, FD_ACCEPT|FD_CLOSE);
```


소켓 `listenSock` 에 네트워크 이벤트 `FD_ACCEPT`(클라이언트 접속)나 `FD_CLOSE`(종료)가 발생하면 `hWnd` 윈도우의 `WM_SOCKET` 사용자 메시지를 발생시킨다. 그러면 `WM_SOCKET` 사용자 메시지는 메시지 큐에 저장되며 윈도우 프로시저에 전달된다.

네트워크 이벤트는 아래와 같다.

- `FD_ACCEPT` : 클라이언트가 접속하였다.
- `FD_READ` : 수신 버퍼에 데이터가 도착했다.
- `FD_WRITE` : 데이터 송신이 가능하다.(송신 버퍼에 여유 공간이 생겼다.)
- `FD_CLOSE` : 접속을 종료했다.
- `FD_CONNECT` : 클라이언트 쪽 이벤트로 접속이 완료되었다.
- `FD_00B` : `00B`(긴급 메시지)가 도착했다.

`WSAAsyncSelect()` 함수의 특징 중 하나는 함수 호출 시 자동으로 소켓을 논블로킹 소켓으로 변경한다. 대기 소켓은 클라이언트의 접속 이벤트를 확인해야 하므로 `FD_ACCEPT` 이벤트가 핵심 이벤트이며 통신 소켓은 데이터 송, 수신 이벤트를 확인해야 하므로 `FD_READ`, `FD_WRITE`, `FD_CLOSE`가 핵심 이벤트다.

네트워크 이벤트가 발생하여 윈도우 프로시저가 호출되면
`LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)`
의 `wParam` 은 이벤트를 발생시킨 소켓 핸들이며 `lParam` 의 하위 바이트는 네트워크 이벤트이고 상위 바이트는 오류 코드값이다.

`WSAGETSELECTERROR(lParam)`은 `HIWORD(lParam)`로 정의되어 있으며 `WSAGETSELECTEVENT(lParam)`은 `LOWORD(lParam)`로 정의되어 있으므로 이 매크로를 호출하여 확인한다.

<절제목>

02. WSAAsyncSelect 모델 에코 서버

</절제목>

WSAAsyncSelect 모델은 기본적으로 윈도우가 필요하므로 소켓과 연결할 임의의 윈도우를 생성한다. 일반적으로 WSAAsyncSelect 모델은 윈도우 기반의 애플리케이션 개발에 사용되지만 여기서는 테스트를 위한 임의의 윈도우를 생성하여 소켓에 네트워크 이벤트가 발생하면 WM_SOCKET 메시지가 윈도우 프로시저에 전달되도록 한다.

다음은 WSAAsyncSelect 모델을 사용한 TCP 에코 서버 예제다.

<코드>

```
[예제 05-01] WSAAsyncSelectTCPServer
#include <winsock2.h>
#include <stdio.h>

#define BUFSIZE 1024
#define WM_SOCKET (WM_USER+1)

// 소켓 정보 저장을 위한 구조체
struct SOCKETINFO
{
    SOCKET sock;
    char buf[BUFSIZE];
    int recvbytes;
    int sendbytes;
    BOOL recvdelayed;
};

HWND hwnd; //윈도우의 핸들
SOCKET listenSock; //대기 소켓 핸들
int nTotalSockets = 0; // 통신 소켓의 개수
SOCKETINFO *SocketInfoArray[FD_SETSIZE];

// 윈도우 프로시저
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
// 소켓 함수 오류 출력
void DisplayMessage()
{
    LPVOID pMsg;
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        WSAGetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR)&pMsg,
        0, NULL);

    printf("%s\n", pMsg);

    LocalFree(pMsg);
}
```

```

}
bool CreateListenSocket()
{
    int retval;

    // 대기 소켓 생성
    listenSock = socket(AF_INET, SOCK_STREAM, 0);
    if(listenSock == INVALID_SOCKET)
    {
        DisplayMessage();
        return false;
    }

    //대기 소켓과 사용자 정의 메시지 WM_SOCKET을 연결한다.
    //년블로킹 소켓으로 변경하는 일도 같이 한다.
    retval = WSAAsyncSelect(listenSock, hwnd,
        WM_SOCKET, FD_ACCEPT|FD_CLOSE);
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }

    // 대기 소켓의 로컬 주소, 포트 설정
    SOCKADDR_IN serveraddr;
    ZeroMemory(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(40100);
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    retval = bind(listenSock, (SOCKADDR *)&serveraddr, sizeof(serveraddr));
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }

    // 대기 소켓의 접속 대기 큐 생성 및 클라이언트 접속 대기
    retval = listen(listenSock, SOMAXCONN);
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }

    return true;
}

// 소켓 정보를 추가한다.
BOOL AddSocketInfo(SOCKET sock)
{
    // FD_SETSIZE - 연결 대기 소켓
    if(nTotalSockets >= (FD_SETSIZE-1)){
        printf("[오류] 소켓 정보를 추가할 수 없습니다!\n");
        return FALSE;
    }

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

    ptr->sock = sock;
    ptr->recvbytes = 0;
    ptr->sendbytes = 0;
    SocketInfoArray[nTotalSockets++] = ptr;
}

```

```

        return TRUE;
    }
    // 소켓 정보를 삭제한다.
    void RemoveSocketInfo(int nIndex)
    {
        SOCKETINFO *ptr = SocketInfoArray[nIndex];

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

        closesocket(ptr->sock);
        delete ptr;

        for(int i=nIndex; i<nTotalSockets; i++){
            SocketInfoArray[i] = SocketInfoArray[i+1];
        }
        nTotalSockets--;
    }

    int GetSocketInfo(SOCKET sock)
    {
        for(int i = 0 ; i < nTotalSockets ; ++i)
        {
            if(SocketInfoArray[i]->sock == sock)
                return i;
        }

        return -1; //클라이언트에서 오류 처리 해야 함.
    }

    int main(int argc, char* argv[])
    {
        // 윈도우 클래스 등록
        WNDCLASS wndclass;
        wndclass.cbClsExtra = 0;
        wndclass.cbWndExtra = 0;
        wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
        wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
        wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
        wndclass.hInstance = NULL;
        wndclass.lpfnWndProc = (WNDPROC)wndProc;
        wndclass.lpszClassName = "MywindowClass";
        wndclass.lpszMenuName = NULL;
        wndclass.style = CS_HREDRAW | CS_VREDRAW;
        if(!RegisterClass(&wndclass)) return -1;

        // 윈도우 생성
        hwnd = CreateWindow("MywindowClass", "TCP 서버",
            WS_OVERLAPPEDWINDOW, 0, 0, 600, 300,
            NULL, (HMENU)NULL, NULL, NULL);
        if(hwnd == NULL) return -1;
        ShowWindow(hwnd, SW_SHOWNORMAL);
        UpdateWindow(hwnd);

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

        // 대기 소켓 초기화(socket()+bind()+listen())
    }

```

```

if( !CreateListenSocket( ) )
{
    printf("대기 소켓 생성 실패!\n");
    return -1;
}

// 메시지 루프
MSG msg;
while(GetMessage(&msg, 0, 0, 0) > 0){
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

// 윈속 종료
WSACleanup();
return msg.wParam;
}

// 윈도우 메시지 처리
LRESULT CALLBACK WndProc(HWND hwnd, UINT uMsg,
    WPARAM wParam, LPARAM lParam)
{
    switch(uMsg){
    case WM_SOCKET: // 소켓 관련 윈도우 메시지
    {
        // 데이터 통신에 사용할 변수
        SOCKETINFO *ptr;
        SOCKET client_sock;
        SOCKADDR_IN clientaddr;
        int addrlen;
        int retval;

        // 오류 발생 여부 확인
        if(WSAGETSELECTERROR(lParam)){
            RemoveSocketInfo(GetSocketInfo(wParam));
            break;
        }

        // 메시지 처리
        switch(WSAGETSECTEVEVENT(lParam)){
        case FD_ACCEPT:
            addrlen = sizeof(clientaddr);
            client_sock = accept(wParam, (SOCKADDR *)&clientaddr,
                &addrlen);
            if(client_sock == INVALID_SOCKET){
                if(WSAGetLastError() != WSAEWOULDBLOCK)
                    DisplayMessage();
                break;
            }
            printf("[TCP 서버] 클라이언트 접속: IP 주소=%s, 포트 번호=%d\n",
                inet_ntoa(clientaddr.sin_addr),
                ntohs(clientaddr.sin_port));
            AddSocketInfo(client_sock);
            retval = WSAAsyncSelect(client_sock, hwnd,
                WM_SOCKET, FD_READ|FD_WRITE|FD_CLOSE);
            if(retval == SOCKET_ERROR){
                DisplayMessage();
                RemoveSocketInfo(GetSocketInfo(client_sock));
            }
            break;
        case FD_READ:
            ptr = SocketInfoArray[GetSocketInfo(wParam)];
            if(ptr->recvbytes > 0){
                ptr->recvdelayed = TRUE;
                break;
            }
        }
    }
}

```

```

    }
    // 데이터 받기
    retval = recv(ptr->sock, ptr->buf, BUFSIZE, 0);
    if(retval == SOCKET_ERROR){
        if(WSAGetLastError() != WSAEWOULDBLOCK){
            DisplayMessage();
            RemoveSocketInfo(GetSocketInfo(wParam));
        }
        break;
    }
    ptr->recvbytes = retval;
    // 받은 데이터 출력
    ptr->buf[retval] = '\0';
    addrlen = sizeof(clientaddr);
    getpeername(wParam, (SOCKADDR *)&clientaddr, &addrlen);
    printf("[TCP/%s:%d] %s\n", inet_ntoa(clientaddr.sin_addr),
        ntohs(clientaddr.sin_port), ptr->buf);
case FD_WRITE:
    ptr = SocketInfoArray[GetSocketInfo(wParam)];
    if(ptr->recvbytes <= ptr->sendbytes)
        break;
    // 데이터 보내기
    retval = send(ptr->sock, ptr->buf + ptr->sendbytes,
        ptr->recvbytes - ptr->sendbytes, 0);
    if(retval == SOCKET_ERROR){
        if(WSAGetLastError() != WSAEWOULDBLOCK){
            DisplayMessage();
            RemoveSocketInfo(wParam);
        }
        break;
    }
    ptr->sendbytes += retval;
    // 받은 데이터를 모두 보냈는지 체크
    if(ptr->recvbytes == ptr->sendbytes){
        ptr->recvbytes = ptr->sendbytes = 0;
        if(ptr->recvdelayed){
            ptr->recvdelayed = FALSE;
            PostMessage(hwnd, WM_SOCKET, wParam,
                FD_READ);
        }
    }
    break;
case FD_CLOSE:
    RemoveSocketInfo(GetSocketInfo(wParam));
    break;
}
return 0;

case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
}

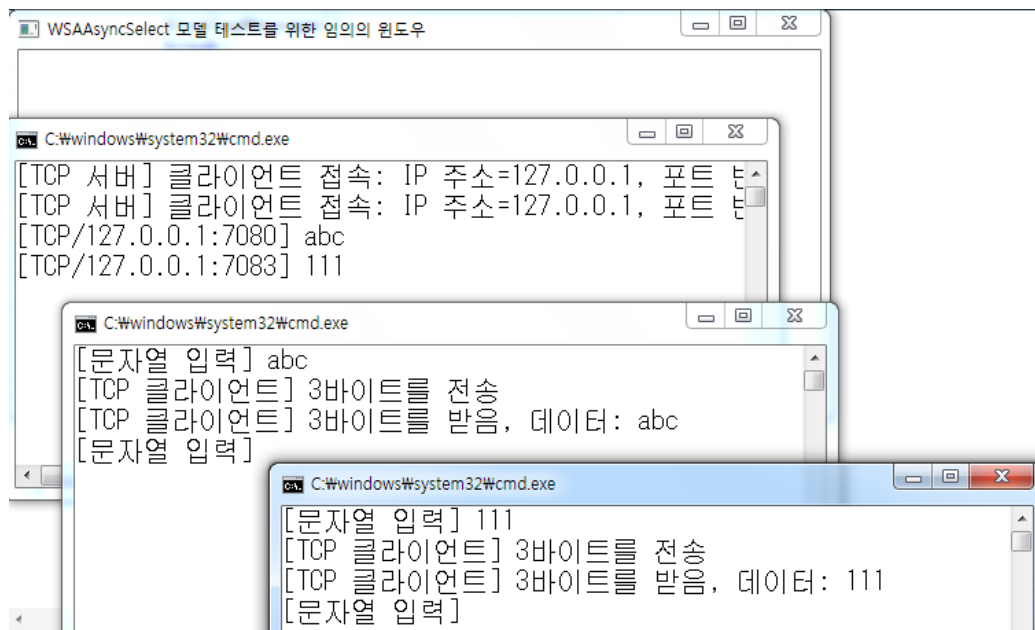
return DefWindowProc(hwnd, uMsg, wParam, lParam);
}

```

</코드>

<결과>

[출력 결과]



</결과>

위 예제는 테스트를 위한 콘솔 기반의 애플리케이션이지만 WSAAsyncSelect 모델을 위하여 임의의 윈도우를 생성한다.

다음 코드는

```
struct SOCKETINFO
{
    SOCKET sock;
    char buf[BUFSIZE];
    int recvbytes;
    int sendbytes;
    BOOL recvdelayed;
};
```

통신 소켓의 정보를 저장하기 위한 구조체로 recvdelayed 멤버는 클라이언트가 보내준 데이터를 서버가 아직 덜 보낸 상태에서 FD_READ 네트워크 이벤트가 발생하면 기존 받은 데이터를 아직 보내지 않았기 때문에 기존 데이터를 모두 에코(보낸 후)한 후 수동으로 다시 FD_READ 네트워크 이벤트를 발생시키기 위해 필요하다.

Select 모델의 select() 함수는 수신 버퍼에 데이터가 존재하여 select() 함수가 반환(수신 버퍼에 데이터가 도착했다는 것을 확인)하더라도 만약 수신 버퍼의 데이터를 읽지 않으면 다시 select() 함수를 호출하더라도 수신 버퍼에 데이터가 존재하는 동안 계속해서 select() 함수는 수신 버퍼의 데이터를 확인하고 계속해서 반환한다.(네트워크의 이벤트 상황을 확인한다.) 하지만 WSAAsyncSelect 모델의 FD_READ 네트워크 이벤트는 수신 버퍼에 데이터가 존재하여 이벤트가 발생하고 만약 데이터를 읽지 않는다면 계속 수신 버퍼에 데이터가 존재하더라도 다시 FD_READ 네트워크 이벤트가 발생되지 않는다. 그러므로 이것을 기억해놓고 데이터를 읽을 수 있는 준비가 되었을 때 수동으로 FD_READ 를 발생시켜 주는 것이 중요하다.

다음 코드는

```
retval = WSAAsyncSelect(listenSock, hwnd,
                        WM_SOCKET, FD_ACCEPT|FD_CLOSE);
if(retval == SOCKET_ERROR)
{
    DisplayMessage();
    return false;
}
```

대기 소켓을 윈도우와 연결하여 FD_ACCEPT 나 FD_CLOSE 이벤트가 발생하면 WM_SOCKET 메시지를 메시지 큐에 저장한다.(윈도우 프로시저가 호출된다.)

다음 코드는

```
hwnd = CreateWindow("MywindowClass", "TCP 서버",
                    WS_OVERLAPPEDWINDOW, 0, 0, 600, 300,
                    NULL, (HMENU)NULL, NULL, NULL);
if(hwnd == NULL) return -1;
ShowWindow(hwnd, SW_SHOWNORMAL);

UpdateWindow(hwnd);
```

소켓과 연결할 윈도우를 생성한다.

다음 코드는

```
case WM_SOCKET: // 소켓 관련 윈도우 메시지
{
    ...

    // 오류 발생 여부 확인
    if(WSAGETSELECTERROR(lParam)){
        RemoveSocketInfo(GetSocketInfo(wParam));
        break;
    }
    ...
}
```

소켓의 네트워크 이벤트가 발생하면 WM_SOCKET 메시지가 발생하며 WSAGETSELECTERROR(lParam)을 사용하여 소켓 에러가 발생한지 확인한다.

다음 코드는

```
switch(WSAGETSELECTEVENT(lParam)){
case FD_ACCEPT:
    addrlen = sizeof(clientaddr);
    client_sock = accept(wParam, (SOCKADDR *)&clientaddr,
                        &addrlen);
    if(client_sock == INVALID_SOCKET){
        if(WSAGetLastError() != WSAEWOULDBLOCK)
            DisplayMessage();
        break;
    }
    ...
}
```

네트워크 이벤트가 FD_ACCEPT 라면 클라이언트 접속을 수락한다.

다음 코드는

```
AddSocketInfo(client_sock);
```



```

        retval = WSAAsyncSelect(client_sock, hwnd,
                                WM_SOCKET, FD_READ|FD_WRITE|FD_CLOSE);
    if(retval == SOCKET_ERROR){
        DisplayMessage();
        RemoveSocketInfo(GetSocketInfo(client_sock));
    }

```

생성된 통신 소켓을 SocketInfoArray 에 추가하고 통신 소켓과 윈도우를 연결한다. 통신 소켓에 FD_READ 나 FD_WRITE 나 FD_CLOSE 가 발생하면 WM_SOCKET 메시지 발생시킨다.

다음 코드는

```

    case FD_READ:
        ptr = SocketInfoArray[GetSocketInfo(wParam)];
        if(ptr->recvbytes > 0){
            ptr->recvdelayed = TRUE;
            break;
        }

```

FD_READ 이벤트가 발생한 통신 소켓을 찾아 아직 받은 데이터를 클라이언트로 에코하지 않았다면 모두 에코한 후 데이터를 다시 받을 준비가 되면 FD_READ 이벤트를 수동으로 발생시키기 위해 recvdelayed 를 TRUE 로 설정한다.

다음 코드는

```

    case FD_WRITE:
        ptr = SocketInfoArray[GetSocketInfo(wParam)];
        if(ptr->recvbytes <= ptr->sendbytes)
            break;

```

FD_WRITE 네트워크 이벤트가 발생하거나 데이터를 수신한 후 break; 없이 바로 실행하여 이벤트가 발생했을 때의 소켓 정보를 찾고 받은 데이터가 보낸 데이터와 같거나 작다면 문장 수행을 건너 뛴다. FD_WRITE 이벤트는 통신 소켓을 생성한 후 최초로 발생하며(소켓이 생성된 후 송신 버퍼가 빈 상태로 데이터를 송신할 수 있는 상태) 이때 문자를 건너뛴다.

다음 코드는

```

        retval = send(ptr->sock, ptr->buf + ptr->sendbytes,
                      ptr->recvbytes - ptr->sendbytes, 0);
    if(retval == SOCKET_ERROR){
        if(WSAGetLastError() != WSAEWOULDBLOCK){
            DisplayMessage();
            RemoveSocketInfo(wParam);
        }
        break;
    }
    ptr->sendbytes += retval;

```

데이터를 에코한 크기와 위치를 계산하여 데이터를 클라이언트로 에코한다. 보낸 데이터는 소켓 정보 구조체 sendbytes 멤버에 누적한다.

다음 코드는

```

    if(ptr->recvbytes == ptr->sendbytes){
        ptr->recvbytes = ptr->sendbytes = 0;
        if(ptr->recvdelayed){
            ptr->recvdelayed = FALSE;
            PostMessage(hwnd, WM_SOCKET, wParam,
                        FD_READ);
        }
    }

```

받은 데이터를 클라이언트로 모두 에코(보냈다면)했다면 소켓 정보 구조체의 `recvbytes` 멤버와 `sendbytes` 멤버를 0 으로 초기화하고 기본에 `FD_READ` 이벤트가 발생하여 수신 버퍼에 데이터가 도착해 있는데도 아직 처리하지 않았다면 `PostMessage()`를 사용하여 수동으로 `FD_READ` 이벤트를 발생시킨다.

다음 코드는

```
case FD_CLOSE:
    RemoveSocketInfo(GetSocketInfo(wParam));
    break;
```

클라이언트가 접속을 종료하면 소켓 정보를 삭제한다.

클라이언트는 쓰레드 버전을 사용하여 테스트할 수 있다.

<장제목>

6. WSAEventSelect 입출력 모델

</장제목>

WSAEventSelect 모델은 소켓과 커널 오브젝트 이벤트 객체를 연결하여 소켓의 네트워크 이벤트가 발생하면 커널 오브젝트 이벤트 객체(이하 이벤트)가 신호 상태가 되어 네트워크 이벤트에 맞는 적절한 소켓 함수를 호출하므로써 작업을 처리하는 입출력 모델이다. WSAEventSelect 모델에서 소켓과 연결되는 커널 오브젝트는 '수동 이벤트'를 사용한다. 그래서 CreateEvent() 함수로 수동 이벤트를 만들어 사용할 수 있지만 소켓 전용 커널 이벤트 생성 함수인 WSACreateEvent()를 사용하여 이벤트를 생성한다.

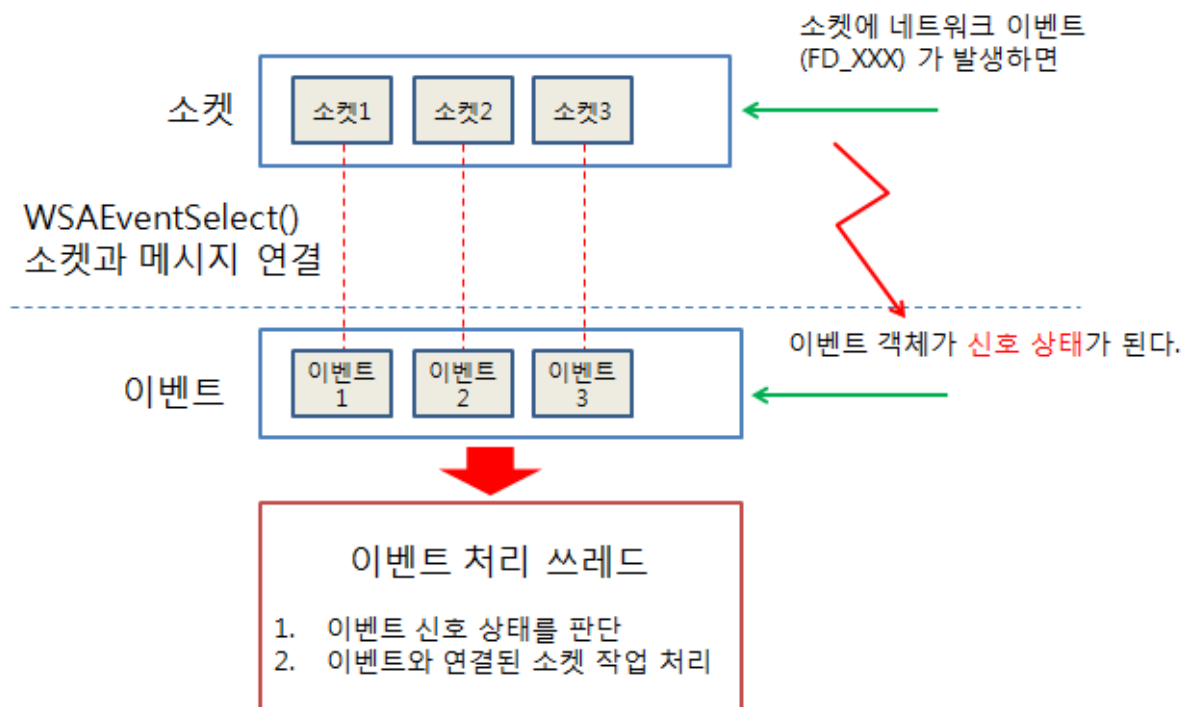
<절제 목>

01. WSAEventSelect 모델의 이해

</절제 목>

WSAEventSelect 모델은 WSAEventSelect() 함수는 네트워크 이벤트가 발생하면 소켓의 이벤트를 신호 상태로 만들 수 있도록 소켓과 커널 이벤트를 연결한다. 애플리케이션에서는 커널 이벤트가 신호 상태가 되는지 대기하고 신호 상태가 되면 어떤 소켓과 연결된 이벤트가 신호 상태인지 확인하여 네트워크 이벤트에 맞는 적절한 작업(소켓 함수 호출)을 수행한다.

다음은 WSAEventSelect() 함수의 동작을 그림으로 표현한 것이다.

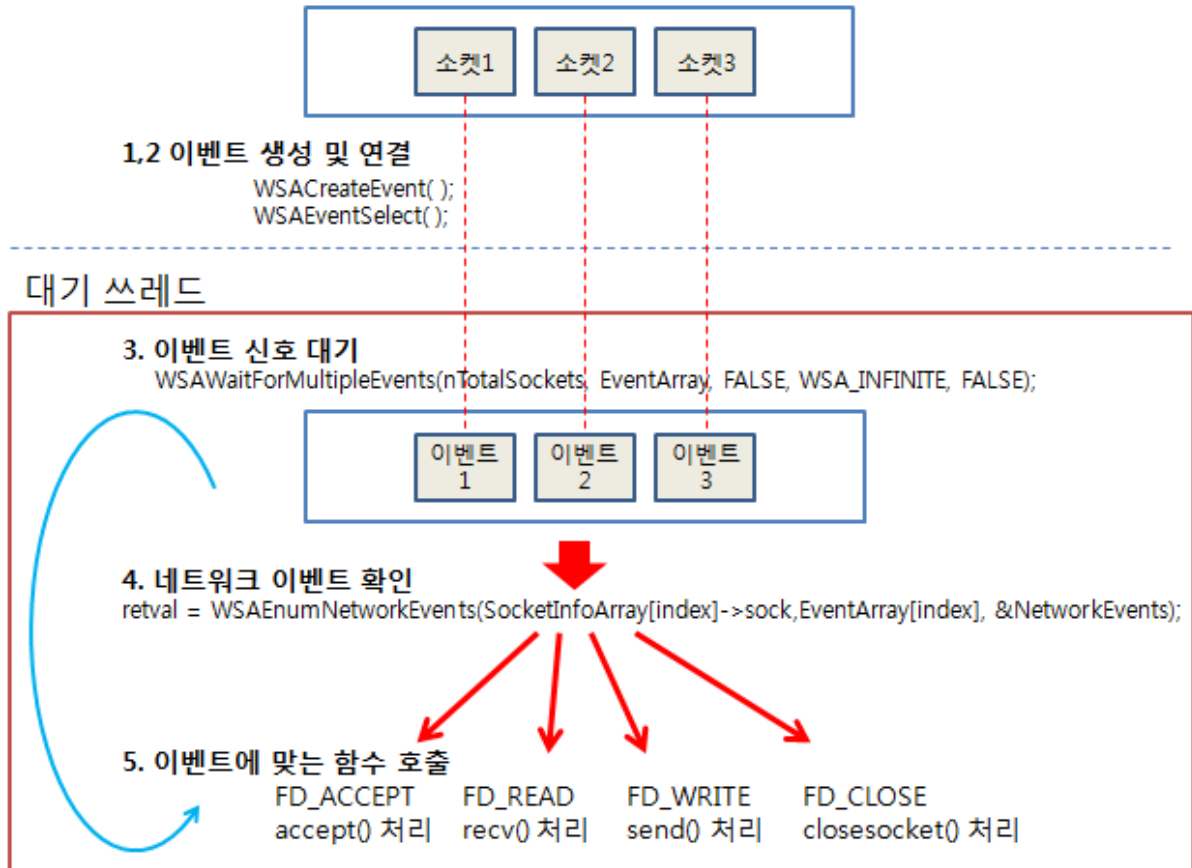


WSAEventSelect() 모델의 절차는 다음과 같다.

1. 커널 이벤트를 생성한다. (WSACreateEvent() 사용)
2. 커널 이벤트를 소켓과 연결하여 네트워크 이벤트 발생 시 커널 이벤트가 신호 상태가 되도록 한다. (WSAEventSelect() 사용)
3. 커널 이벤트가 신호 상태가 되는지 대기한다. (WSAWaitForMultipleEvents() 사용)
4. 커널 이벤트와 연결된 어떤 소켓의 어떤 네트워크 이벤트가 발생되었는지 확인한다. (WSAEnumNetworkEvents() 사용)

5. 네트워크 이벤트에 맞는 적절한 작업을 수행한다. (accept(), recv(), send() 등 사용)

다음은 WSAEventSelect() 모델의 절차를 표현한 그림이다.



이제 WSAEventSelect 모델의 절차를 확인했으니 기능을 수행하기 위해 필요한 소켓 함수들을 하나하나 살펴보도록 하자.

커널 이벤트를 생성하는 함수는

```
WSAEVENT WSACreateEvent();
```

이고 성공 시 객체의 핸들을 반환하고 실패하면 WSA_INVALID_EVENT 를 반환한다. WSACreateEvent() 는 수동 이벤트를 생성하며 네트워크 이벤트에 의해 이 이벤트가 신호 상태가 되면 네트워크 정보를 확인하기 위한 함수 WSAEnumNetworkEvents()에 의해 다시 비신호 상태로 변경된다. 만약 WSAEnumNetworkEvents()를 호출하지 않으면 계속 신호 상태로 남게 된다. 그래서 WSAAsyncSelect() 모델은 네트워크 이벤트가 발생할 때 적절한 작업을 처리하지 못하면 다시 네트워크 이벤트가 발생되지 않으므로 이 네트워크 이벤트와 연결된 메시지를 수동으로 메시지

발생시켜야 하지만 `WSAEnumNetworkEvents()`는 소켓의 네트워크 이벤트와 연결된 커널 이벤트가 계속 신호 상태로 남아 있으므로 `WSAWaitForMultipleEvents()`를 호출하여 확인할 수 있다.

커널 이벤트를 제거하는 함수는

```
BOOL WSACloseEvent(WSAEVENT hEvent);
```

이며 이벤트를 제거(운영체제로 리소스 반환) 한다.

소켓과 커널 이벤트를 연결하기 위해

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

- `s`는 소켓의 핸들이다.
- `hEventObject`는 소켓과 연결할 커널 이벤트 핸들이다.
- `lNetworkEvents`는 설정할 네트워크 이벤트다.
네트워크 이벤트는 `WSAAsyncSelect` 모델에서 공부한 내용과 같다.

- `FD_ACCEPT` : 클라이언트가 접속하였다.
- `FD_READ` : 수신 버퍼에 데이터가 도착했다.
- `FD_WRITE` : 데이터 송신이 가능하다.(송신 버퍼에 여유 공간이 생겼다.)
- `FD_CLOSE` : 접속을 종료했다.
- `FD_CONNECT` : 클라이언트 쪽 이벤트로 접속이 완료되었다.
- `FD_00B` : 00B(긴급 메시지)가 도착했다.

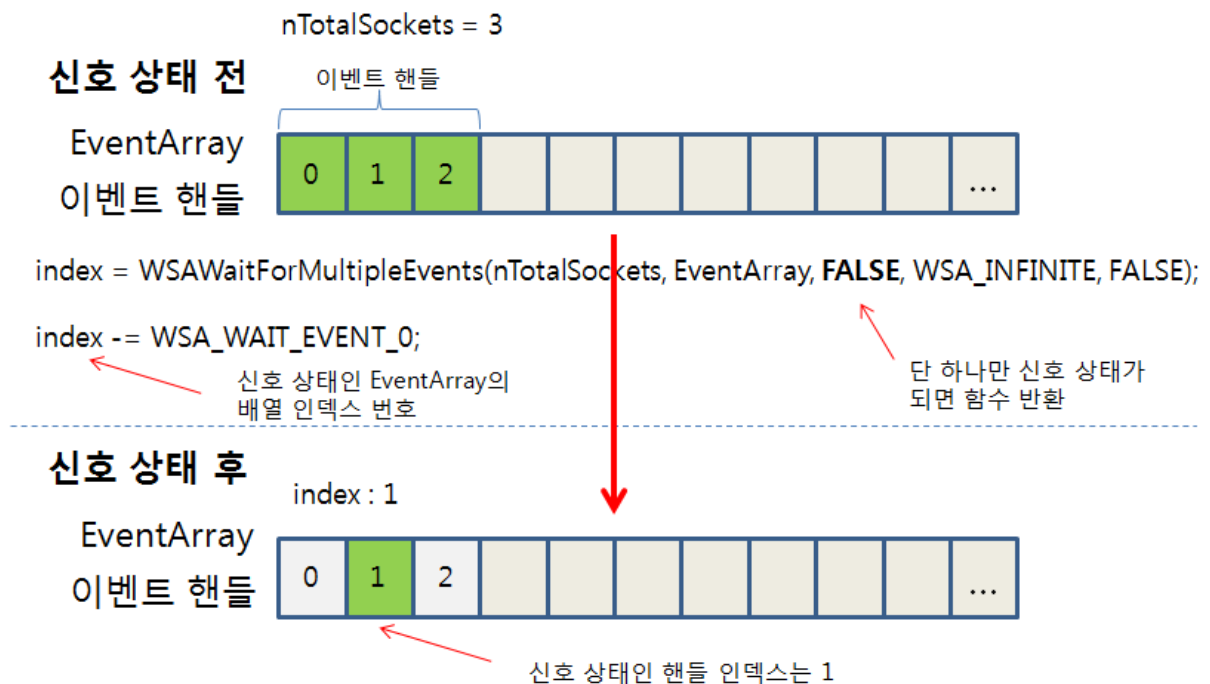
`WSAWaitForMultipleEvents()` 함수는 이벤트 신호를 대기하고 확인하는 함수로

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

- `cEvents`는 대기하는 커널 이벤트 개수다.
- `lphEvents`는 대기하는 커널 이벤트 핸들 테이블 주소다.
- `fWaitAll`은 대기하는 커널 이벤트가 모두 신호 상태가 되었을 때 대기 함수가 반환할 것인지의 인수로 `false`이면 대기하는 커널 이벤트 중 하나만 신호 상태가 되어도 함수가 반환 한다.
- `dwTimeout`은 대기 시간을 밀리초 단위로 설정한다. `WSA_INFINITE` 값이면 신호 상태가 될 때까지 무한정 대기한다.
- `fAlertable`은 입출력 완료 루틴에서 사용하며 `WSAEventSelect` 모델은 `false`로 설정한다.
-

이때 성공 시 반환값에 WSA_WAIT_EVENT_0를 빼면 신호 상태가 발생한 커널 이벤트 테이블의 인덱스 번호가 된다. 대기 시간을 설정하여 만료되면 WSA_WAIT_TIMEOUT 을 반환하며 실패 시 WSA_WAIT_FAILED 를 반환한다.

다음 그림은 커널 이벤트를 대기하고 신호 상태가 되었을 때의 상황을 그림으로 표현한 것으로 1번 커널 이벤트가 신호 상태가 되었을 때 WSAWaitForMultipleEvents() 함수는 반환하며 index-WSA_WAIT_EVENT_0의 값은 신호 상태인 배열의 인덱스 1이다. WSA_WAIT_EVENT_0은 내부적으로 0으로 정의되어 있으며 지금은 빼지 않아도 된다. 하지만 라이브러리 스펙의 정의에 따라 빼주는 것이 좋다.



네트워크 이벤트 정보를 확인하기 위해서는

```
int WSAEnumNetworkEvents(SOCKET s, WSAEVENT hEvnetObject, LPWSANETWORKEVENTS lpNetworkEvents);
```

를 사용한다.

성공 시 0이며 실패는 SOCKET_ERROR 를 반환한다.

- s 는 정보를 확인할 소켓의 핸들이다.
- hEventObject 는 소켓과 연결된 이벤트의 핸들로 커널 이벤트를 다시 비신호 상태로 변경하기 위해 사용된다.
- lpNetworkEvents 는 WSANETWORKEVENTS 구조체 변수로 네트워크 이벤트와 오류 정보를 반환한다.

WSANETWORKEVENTS 구조체는

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

로 lNetworkEvents 멤버는 발생한 네트워크 이벤트다. iErrorCode 는 배열로 각각의 네트워크 이벤트와 매핑된 오류 코드값이 보관된다.

네트워크 이벤트 오류값을 확인하기 위한 배열의 인덱스 값은 아래와 같다.

- FD_ACCEPT 는 FD_ACCEPT_BIT 를 사용
- FD_READ 는 FD_READ_BIT 를 사용
- FD_WRITE 는 FD_WRITE_BIT 를 사용
- FD_CLOSE 는 FD_CLOSE_BIT 를 사용
- FD_CONNECT 는 FD_CONNECT_BIT 를 사용
- FD_OOB 는 FD_OOB_BIT 를 사용

이처럼 각 네트워크 이벤트에 XXX_BIT 를 붙이면 된다.

<절제목>

02. WSAEventSelect 모델 에코 서버

</절제목>

WSAEventSelect 모델은 윈도우 기반의 애플리케이션이 아니거나 윈도우 없이도 커널 이벤트를 사용하여 소켓 네트워크의 이벤트 상황을 애플리케이션이 확인할 수 있는 소켓 입출력 모델이다.

다음은 WSAEventSelect 모델을 사용한 에코 서버 예제다.

<코드>

```
[예제 06-01] WSAEventSelectTCPServer
#include <winsock2.h>
#include <process.h>
#include <stdio.h>

#define BUFSIZE 1024

// 소켓 정보 저장을 위한 구조체
struct SOCKETINFO
{
    SOCKET sock;
    char buf[BUFSIZE];
    int recvbytes;
    int sendbytes;
};

SOCKET listenSock; //대기 소켓 핸들
int nTotalSockets = 0; //커널 이벤트를 위한 소켓 개수
// 소켓 정보 저장 배열
SOCKETINFO *SocketInfoArray[WSA_MAXIMUM_WAIT_EVENTS];
// 소켓 정보와 연결되는 커널 이벤트 배열
WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];

void DisplayMessage()
{
    LPVOID pMsg;
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        WSAGetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR)&pMsg,
        0, NULL);
    printf("%s\n", pMsg);
    LocalFree(pMsg);
}

// 소켓 정보를 추가한다.
BOOL AddSocketInfo(SOCKET sock)
{
    if(nTotalSockets >= WSA_MAXIMUM_WAIT_EVENTS){
        printf("[오류] 소켓 정보를 추가할 수 없습니다!\n");
    }
}
```

```

        return FALSE;
    }

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

    WSAEVENT hEvent = WSACreateEvent();
    if(hEvent == WSA_INVALID_EVENT){
        DisplayMessage();
        return FALSE;
    }

    ptr->sock = sock;
    ptr->recvbytes = 0;
    ptr->sendbytes = 0;
    SocketInfoArray[nTotalSockets] = ptr;
    EventArray[nTotalSockets] = hEvent;
    nTotalSockets++;

    return TRUE;
}

// 소켓 정보 삭제
void RemoveSocketInfo(int nIndex)
{
    SOCKETINFO *ptr = SocketInfoArray[nIndex];

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

    closesocket(ptr->sock);
    delete ptr;
    WSACloseEvent(EventArray[nIndex]);

    for(int i=nIndex; i<nTotalSockets; i++){
        SocketInfoArray[i] = SocketInfoArray[i+1];
        EventArray[i] = EventArray[i+1];
    }
    nTotalSockets--;
}

bool CreateListenSocket()
{
    int retval;

    // 대기 소켓 생성
    listenSock = socket(AF_INET, SOCK_STREAM, 0);
    if(listenSock == INVALID_SOCKET)
    {
        DisplayMessage();
        return false;
    }

    // 대기 소켓을 SocketInfoArray에 추가한다.
    if(AddSocketInfo(listenSock) == FALSE)
        return false;

    // 대기 소켓과 커널 이벤트 객체를 연결한다.
    // FD_ACCEPT나 FD_CLOSE 네트워크 이벤트가 발생하면 커널 이벤트 객체가 신호상태가 된다.
    retval = WSAEventSelect(listenSock, EventArray[nTotalSockets-1],

```

```

        FD_ACCEPT|FD_CLOSE);
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }
    // 대기 소켓의 로컬 주소, 포트 설정
    SOCKADDR_IN serveraddr;
    ZeroMemory(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(40100);
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    retval = bind(listenSock, (SOCKADDR *)&serveraddr, sizeof(serveraddr));
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }

    // 대기 소켓의 접속 대기 큐 생성 및 클라이언트 접속 대기
    retval = listen(listenSock, SOMAXCONN);
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }

    return true;
}
unsigned int WINAPI WorkerThread(void* pParam)
{
    int retval;
    int index;
    WSANETWORKEVENTS NetworkEvents;
    SOCKET client_sock;
    SOCKADDR_IN clientaddr;
    int addrlen;

    while(1){
        // 이벤트 객체가 신호상태가 될 때까지 대기한다.
        index = WSAWaitForMultipleEvents(nTotalSockets, EventArray,
            FALSE, WSA_INFINITE, FALSE);
        if(index == WSA_WAIT_FAILED){
            DisplayMessage();
            continue;
        }
        index -= WSA_WAIT_EVENT_0;

        // 네트워크 이벤트가 발생한 소켓의 정보를 확인한다.
        // 신호 상태의 수동 이벤트를 비신호 상태로 변경한다.
        retval = WSAEnumNetworkEvents(SocketInfoArray[index]->sock,
            EventArray[index], &NetworkEvents);
        if(retval == SOCKET_ERROR){
            DisplayMessage();
            continue;
        }

        // FD_ACCEPT 이벤트를 처리한다.
        if(NetworkEvents.lNetworkEvents & FD_ACCEPT){
            if(NetworkEvents.iErrorCode[FD_ACCEPT_BIT] != 0){
                DisplayMessage();
                continue;
            }

            addrlen = sizeof(clientaddr);
            client_sock = accept(SocketInfoArray[index]->sock,
                (SOCKADDR *)&clientaddr, &addrlen);

```

```

        if(client_sock == INVALID_SOCKET){
            DisplayMessage();
            continue;
        }
        printf("[TCP 서버] 클라이언트 접속: IP 주소=%s, 포트 번호=%d\n",
            inet_ntoa(clientaddr.sin_addr),
            ntohs(clientaddr.sin_port));

        if(nTotalSockets >= WSA_MAXIMUM_WAIT_EVENTS){
            printf("[오류] 더 이상 접속을 받아들일 수 없습니다!\n");
            closesocket(client_sock);
            continue;
        }

        if(AddSocketInfo(client_sock) == FALSE)
            continue;

        retval = WSAEventSelect(client_sock,
            EventArray[nTotalSockets-1],
            FD_READ|FD_WRITE|FD_CLOSE);
        if(retval == SOCKET_ERROR)
        {
            DisplayMessage();
            break;
        }
    }

    // FD_READ, FD_WRITE 이벤트 처리를 처리한다.
    if(NetworkEvents.lNetworkEvents & FD_READ
        || NetworkEvents.lNetworkEvents & FD_WRITE)
    {
        if(NetworkEvents.lNetworkEvents & FD_READ
            && NetworkEvents.iErrorCode[FD_READ_BIT] != 0)
        {
            DisplayMessage();
            continue;
        }
        if(NetworkEvents.lNetworkEvents & FD_WRITE
            && NetworkEvents.iErrorCode[FD_WRITE_BIT] != 0)
        {
            DisplayMessage();
            continue;
        }
        SOCKETINFO *ptr = SocketInfoArray[index];
        if(ptr->recvbytes == 0){
            // 데이터 받기
            retval = recv(ptr->sock, ptr->buf, BUFSIZE, 0);
            if(retval == SOCKET_ERROR){
                if(WSAGetLastError() != WSAEWOULDBLOCK){
                    DisplayMessage();
                    RemoveSocketInfo(index);
                }
                continue;
            }
            ptr->recvbytes = retval;
            // 받은 데이터 출력
            ptr->buf[retval] = '\0';
            addrlen = sizeof(clientaddr);
            getpeername(ptr->sock, (SOCKADDR *)&clientaddr,
                &addrlen);
            printf("[TCP/%s:%d] %s\n",
                inet_ntoa(clientaddr.sin_addr),
                ntohs(clientaddr.sin_port), ptr->buf);
        }

        if(ptr->recvbytes > ptr->sendbytes){

```

```

        // 데이터 보내기
        retval = send(ptr->sock, ptr->buf + ptr->sendbytes,
            ptr->recvbytes - ptr->sendbytes, 0);
        if(retval == SOCKET_ERROR){
            if(WSAGetLastError() != WSAEWOULDBLOCK){
                DisplayMessage();
                RemoveSocketInfo(index);
            }
            continue;
        }
        ptr->sendbytes += retval;
        // 받은 데이터를 모두 보냈는지 체크
        if(ptr->recvbytes == ptr->sendbytes)
            ptr->recvbytes = ptr->sendbytes = 0;
    }

    // FD_CLOSE 이벤트를 처리한다.
    if(NetworkEvents.lNetworkEvents & FD_CLOSE){
        if(NetworkEvents.iErrorCode[FD_CLOSE_BIT] != 0)
            DisplayMessage();
        RemoveSocketInfo(index);
    }
}

return 0;
}

int main(int argc, char* argv[])
{
    WSADATA wsa;

    if(WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("윈도우 소켓 초기화 실패!\n");
        return -1;
    }

    // 대기 소켓 초기화(socket()+bind()+listen())
    if( !CreateListenSocket( ) )
    {
        printf("대기 소켓 생성 실패!\n");
        return -1;
    }

    // 대기 스레드 종료를 기다림.
    unsigned int threadID;
    WaitForSingleObject((HANDLE)_beginthreadex(0,0, workerThread, 0, 0,
        &threadID), INFINITE);

    WSACleanup();
    return 0;
}

```

</코드>

<결과>

[출력 결과]

```

C:\windows\system32\cmd.exe
[TCP 서버] 클라이언트 접속: IP 주소=127.0.0.1, 포트 번호=1615
[TCP 서버] 클라이언트 접속: IP 주소=127.0.0.1, 포트 번호=1618
[TCP/127.0.0.1:1618] abc
[TCP/127.0.0.1:1615] 333

C:\windows\system32\cmd.exe
[문자열 입력] abc
[TCP 클라이언트] 3바이트를 전송
[TCP 클라이언트] 3바이트를 받음, 데이터: abc
[문자열 입력]

C:\windows\system32\cmd.exe
[문자열 입력] 333
[TCP 클라이언트] 3바이트를 전송
[TCP 클라이언트] 3바이트를 받음, 데이터: 333
[문자열 입력]

```

</결과>

다음 코드는

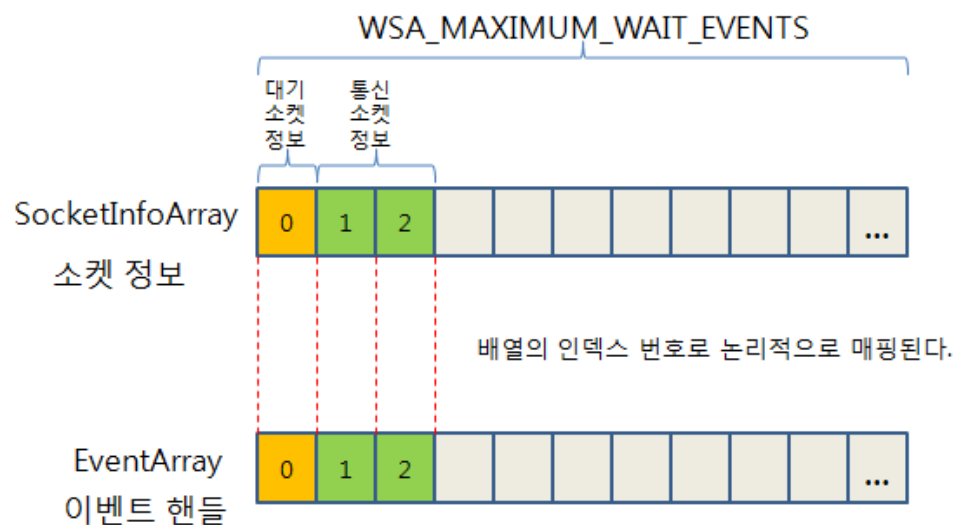
```

int nTotalSockets = 0
SOCKETINFO *SocketInfoArray[WSA_MAXIMUM_WAIT_EVENTS];
WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];

```

소켓 정보와 커널 이벤트 정보를 보관하기 위한 배열로 ‘테이블 매핑’ 이라는 방법을 사용하여 서로의 관계를 확인한다. 테이블 매핑은 보통 하나의 구조체에 묶일 수 있는 정보지만 각각 독립적으로 멤버 정보를 유지할 때 사용되는 방법으로 배열의 인덱스 번호를 사용하여 같은 위치의 인덱스는 관련성 있는 정보를 나타낸다. EventArray 배열은 WSAWaitForMultipleEvents()를 사용하기 위해 이처럼 구분되었다.

다음은 인덱스 번호로 두 배열의 매핑을 표현한 그림이다.



다음 코드는

```

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

WSAEVENT hEvent = WSACreateEvent();
if(hEvent == WSA_INVALID_EVENT){
    DisplayMessage();
    return FALSE;
}

ptr->sock = sock;
ptr->recvbytes = 0;
ptr->sendbytes = 0;
SocketInfoArray[nTotalSockets] = ptr;
EventArray[nTotalSockets] = hEvent;
nTotalSockets++;

```

AddSocketInfo() 함수의 일부로 소켓 정보 객체와 커널 이벤트 객체를 생성하고 각 정보 배열의 같은 인덱스 위치에(nTotalSockets)에 각각의 정보를 테이블 매핑하여 보관한다. 대기 소켓 같은 경우 소켓 정보 구조체는(SOCKETINFO) 단지 소켓 핸들만 저장하는 용도로 사용된다.

다음 코드는

```

closesocket(ptr->sock);
delete ptr;
WSACloseEvent(EventArray[nIndex]);

for(int i=nIndex; i<nTotalSockets; i++){
    SocketInfoArray[i] = SocketInfoArray[i+1];
    EventArray[i] = EventArray[i+1];
}
nTotalSockets--;

```

RemoveSocketInfo() 함수의 일부 코드로 소켓을 닫고 소켓 정보 객체를 삭제하고 커널 이벤트를 닫는다. 마지막으로 닫은 소켓에 해당하는 소켓 정보 배열과 커널 이벤트 배열의 원소를 제거한다.

다음 코드는

```

if(AddSocketInfo(listenSock) == FALSE)
    return false;

```

대기 소켓 정보를 소켓 정보 배열에 저장하고 커널 이벤트 배열에 이벤트를 생성하여 보관한 후에

```

retval = WSAEventSelect(listenSock, EventArray[nTotalSockets-1],
    FD_ACCEPT|FD_CLOSE);
if(retval == SOCKET_ERROR)
{
    DisplayMessage();
    return false;
}

```

WSAEventSelect() 함수를 사용하여 대기 소켓과 커널 이벤트를 연결하여 네트워크 이벤트 FD_ACCEPT 와 FD_CLOSE 가 발생하면 커널 이벤트를 신호 상태로 변경하도록 구성하는 코드다.

다음 코드는

```

index = WSAAwaitForMultipleEvents(nTotalSockets, EventArray,
    FALSE, WSA_INFINITE, FALSE);
if(index == WSA_WAIT_FAILED){
    DisplayMessage();
    continue;
}

```

```

    }
    index -= WSA_WAIT_EVENT_0;

```

WorkerThread 에서 커널 이벤트가 신호 상태가 되기를 대기하는 코드로 EventArray 배열의 이벤트를 nTotalSockets 개수만큼 대기하며 단 하나의 이벤트가 신호상태가 되어도 함수는 반환한다. 반환값에 WSA_WAIT_EVENT_0 을 빼서 이벤트가 발생한 배열의 인덱스 번호를 찾아 낸다.

다음 코드는

```

    retval = WSAEnumNetworkEvents(SocketInfoArray[index]->sock,
        EventArray[index], &NetworkEvents);
    if(retval == SOCKET_ERROR){
        DisplayMessage();
        continue;
    }

```

네트워크 이벤트 정보를 반환받는 중요 코드로 NetworkEvents 에 out parameter 를 이용하여 이벤트 정보를 반환한다. 또 WSAEnumNetworkEvents() 함수는 두 번째 인수로 전달한 커널 이벤트를 비신호 상태로 되돌린다.

다음 코드는

```

if(NetworkEvents.lNetworkEvents & FD_ACCEPT)

```

FD_ACCEPT 네트워크 이벤트가 반환되었는지 검사하는 코드로 NetworkEvent.lNetworkEvents 가 각각의 네트워크 이벤트에 따라 비트를 셋트하므로 비트 논리 연산(AND)을 사용하여 검사한다

다음 코드는

```

    if(NetworkEvents.iErrorCode[FD_ACCEPT_BIT] != 0){
        DisplayMessage();
        continue;
    }

```

NetworkEvents.iErrorCode[]배열의 FD_ACCEPT_BIT 를 검사하여 FD_ACCEPT 네트워크 이벤트가 반환될 때 오류가 발생했는지 검사하는 코드다.

다음 코드는

```

    if(AddSocketInfo(client_sock) == FALSE)
        continue;

    retval = WSAEventSelect(client_sock,
        EventArray[nTotalSockets-1],
        FD_READ|FD_WRITE|FD_CLOSE);
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        break;
    }

```

accept() 함수로 통신 소켓이 생성되면 소켓 정보와 커널 이벤트를 생성하여 소켓 정보 배열과 커널 이벤트 배열에 각각 저장하고 통신 소켓을 커널 이벤트와 네트워크 이벤트 FD_READ, FD_WRITE, FD_CLOSE 가 발생되면 커널 이벤트를 신호 상태로 변경하도록 연결한다.

다음 코드는

```

    if(NetworkEvents.lNetworkEvents & FD_READ
        || NetworkEvents.lNetworkEvents & FD_WRITE)

```


FD_READ 나 FD_WRITE 네트워크 이벤트가 반환되었는지 검사하는 코드로 NetworkEvent.lNetworkEvents 가 각각의 네트워크 이벤트에 따라 비트를 셋트하므로 비트 논리 연산(AND)을 사용하여 검사한다.

다음 코드는

```
if(NetworkEvents.lNetworkEvents & FD_READ
    && NetworkEvents.iErrorCode[FD_READ_BIT] != 0)
{
    DisplayMessage();
    continue;
}
if(NetworkEvents.lNetworkEvents & FD_WRITE
    && NetworkEvents.iErrorCode[FD_WRITE_BIT] != 0)
{
    DisplayMessage();
    continue;
}
```

NetworkEvents.iErrorCode[] 배열의 FD_READ_BIT 와 FD_WRITE_BIT 를 각각 검사하여 네트워크 이벤트가 반환될 때 오류가 발생했는지 검사하는 코드다.

다음 코드는

```
if(ptr->recvbytes == 0){
    retval = recv(ptr->sock, ptr->buf, BUFSIZE, 0);
    if(retval == SOCKET_ERROR){
        if(WSAGetLastError() != WSAEWOULDBLOCK){
            DisplayMessage();
            RemoveSocketInfo(index);
        }
        continue;
    }
    ptr->recvbytes = retval;
    ...
}
```

아직 받은 데이터가 없다면(혹은 받은 데이터를 모두 보냈다면 항상 소켓 정보의 recvbytes 멤버는 0 이므로) 수신 버퍼에서 애플리케이션 버퍼로 데이터를 복사한다. 만약 아직 클라이언트로 보내지 않은 데이터가 있다면 다음 턴에 데이터를 수신하게 된다.

다음 코드는

```
if(ptr->recvbytes > ptr->sendbytes){
    // 데이터 보내기
    retval = send(ptr->sock, ptr->buf + ptr->sendbytes,
        ptr->recvbytes - ptr->sendbytes, 0);
    if(retval == SOCKET_ERROR){
        if(WSAGetLastError() != WSAEWOULDBLOCK){
            DisplayMessage();
            RemoveSocketInfo(index);
        }
        continue;
    }
    ptr->sendbytes += retval;
    // 받은 데이터를 모두 보냈는지 체크
    if(ptr->recvbytes == ptr->sendbytes)
        ptr->recvbytes = ptr->sendbytes = 0;
}
```

아직 받은 데이터를 클라이언트로 보내지 않은(송신 버퍼로 복사하지 않은) 데이터가 있다면 남은 데이터를 송신 퍼버로 복사하고 모두 송신 버퍼로 받은 데이터를 복사하면(ptr->recvbytes == ptr->sendbytes) 두 멤버 변수를 모두 0 으로 초기화한다.

다음 코드는

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

클라이언트가 접속 종료했다면 서버가 마무리 작업을 처리하는 코드다.

<장제목>

7. Over lapped 모델

</장제목>

Over lapped 모델은 윈도우 소켓이 제공하는 비동기 입출력 모델로 성능이 뛰어나며 비동기 입출력과 관련한 여러가지 기능과 API 가 제공된다. Over lapped 모델이 기존 입출력 모델과 크게 다른점은 비동기 입출력 함수를 사용하여 선 동작(먼저 소켓 함수를 호출), 후 완료 처리 순으로 입출력 작업을 수행한다.

그래서 Over lapped 모델은 비동기 입출력 함수의 완료 처리가 핵심이며 이 완료 처리 방법에 따라 크게 두 가지로 나뉜다. 하나는 커널 이벤트를 사용한 Over lapped 모델이며 또 하나는 완료 루틴(completion routine)이라고 부르는 콜백 함수(callback function)를 사용하는 방법이다.

<절제목>

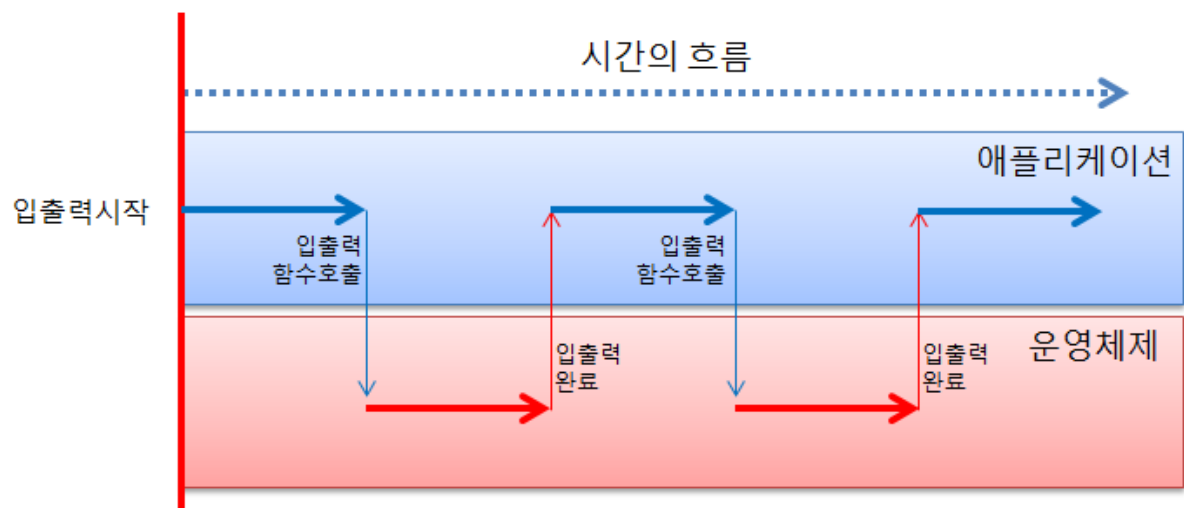
01. Overlapped 모델과 비동기 입출력 함수

</절제목>

Overlapped 모델은 기본적으로 비동기 입출력으로 동작하며 여러가지 비동기 입출력과 관련한 함수를 제공한다. Overlapped 모델은 윈도우 소켓뿐만 아니라 윈도우 운영체제에서 다른 비동기 입출력(파일 입출력, 동기화 오브젝트 입출력 등)을 위한 모델로 사용된다. 한마디로 Overlapped 모델은 윈도우 운영체제가 제공하는 비동기 입출력 모델이다.

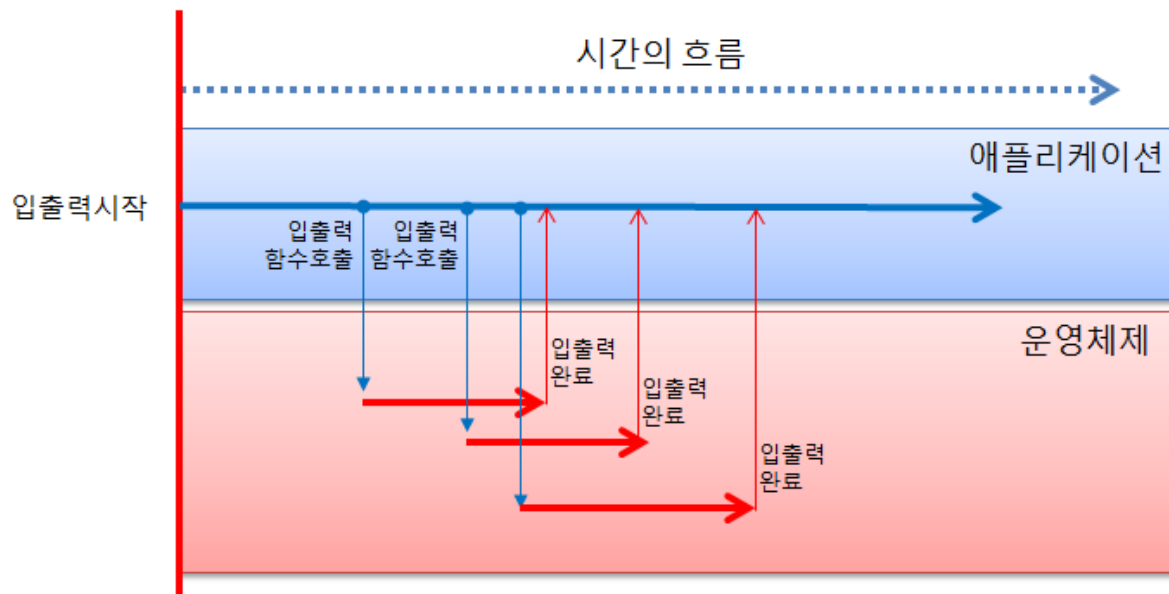
그렇다면 비동기 입출력이 무엇인지 동기 입출력과 비교하여 알아 보도록하자.

다음 그림은 동기 입출력 함수를 그림으로 표현한 것으로 동기 입출력 함수는 시간 흐름에 따라 입출력 동작이 모두 완료된 후 애플리케이션 코드를 실행할 수 있다. 동기 입출력 함수의 장점은 입출력 완료가 발생하는 시점을 정확히 애플리케이션에서 확인하므로 작업의 흐름이 단순하다는 것이다. 하지만 단점은 입출력 함수가 수행되는 동안 애플리케이션 코드는 다른 작업을 수행하지 못하며 입출력이 완료할 때까지 대기해야 한다. 입출력 작업이 긴 시간을 요구한다면 애플리케이션은 다른 작업을 수행하지 못하고 많은 시간을 입출력 작업 완료 대기에서 소모해야 한다.



많은 작업의 입출력을 수행할 때 발생하는 동기 입출력의 비효율성을 해결할 수 있는 방법으로 비동기 입출력을 사용할 수 있다.

다음 그림은 비동기 입출력을 표현한 그림으로 애플리케이션은 입출력 함수를 호출하고 운영체제가 실제 입출력을 담당한다. 운영체제는 입출력이 완료하면 애플리케이션에 보고하며(보고하는 방법이 존재하며 이 방법에 따라 비동기 입출력 모델이 달라진다.) 이때 애플리케이션은 입출력 완료에 대한 처리를 수행한다. 비동기 입출력 모델은 운영체제가 보고하는 이 입출력 완료 처리가 핵심이며 상대적으로 동기 입출력 모델에 비해 코드가 복잡해진다.



운영체제에서 동작하는 비동기 입출력 작업은 중첩(Overlapped)되어 진행되며 그래서 윈도우 운영체제는 이 비동기 입출력 모델을 Overlapped 입출력 모델이라 한다. 비동기 입출력은 단순히 운영체제가 각 입출력을 독립된 스레드로 수행한다고 이해하면 쉽다. 사실 내부적으로 그렇게 동작한다.

다음은 비동기 입출력을 지원하는 함수다.

AcceptEx(), ConnectEx(), DisconnectEx(), TransmitFile(), TransmitPackets(), WSALocatl(), WSANSPloctl(), WSAProviderConfigChange(), WSARecv(), WSARecvFrom(), WSARecvMsg(), WSASelect(), WSASelectTo() 등

이 중 자주 사용되는 WSARecv()와 WSASelect() 를 공부해 보자.

WSASelect()는 비동기 전송함수로

```
int WSASelect(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

- s 는 비동기 입출력 소켓의 핸들이다.
- lpBuffers 는 WSABUF 구조체 배열의 주소다. 이 구조체의 여러 정보를 한 번에 전송할 수 있다. WSABUF 구조체는

```
typedef struct __WSABUF {
    u_long    len;
    char FAR *buf;
} WSABUF, *LPWSABUF;
```

len 은 보낼 데이터의 크기를 지정하며 buf 는 보낼 애플리케이션 버퍼의 시작 주소를 지정한다.

- dwBufferCount 는 WSABUF 구조체의 개수다.
- lpNumberOfBytesSent 는 보낸 데이터의 바이트 수를 반환하는 out parameter 다.
- dwFlags 는 보낼 데이터의 함수의 옵션으로 대부분 0 이다. send()의 마지막 인자와 같다.
- lpOverlapped 는 WSAOVERLAPPED 구조체 변수의 주소로 비동기 입출력 함수의 핵심 인수다. 이 변수가 NULL 이면 동기 입출력 함수로 동작한다. 이 구조체 변수의 역할은 비동기 입출력의 정보를 보관하는데 사용되며 또 입출력이 완료되면 운영체제가 애플리케이션에 완료보고를 수행하기 위한 용도로 사용된다.

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

Internal 과 InternalHigh 는 운영체제 내부에서 입출력 정보를 유지하는데 사용되며 offset 과 offsetHigh 는 입출력의 시작 위치를 설정할 수 있는 멤버로 offset 은 하위 4 바이트, offsetHigh 는 상위 4 바이트 시작 위치를 설정할 수 있다. 보통 모두 0 으로 설정한다. 마지막 멤버 hEvent 는 Overlapped 입출력 모델의 첫 번째 완료 보고를 받을 수 있는 방법에 사용되는 멤버로 완료 보고를 받기 위한 커널 이벤트의 핸들로 사용된다.

- lpCompletionRoutine 은 Overlapped 입출력 모델의 두 번째 완료 보고를 받을 수 있는 방법에 사용되며 완료 루틴 함수의 시작 주소다.

다음은 비동기 수신 함수 WSARecv() 함수로

```
int WSARecv(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
```

```

LPDWORD lpNumberOfBytesRecvd,
LPDWORD lpFlags,
LPWSAOVERLAPPED lpOverlapped,
LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);

```

WSASend()와 대부분의 인수가 비슷하며 수신하는데 사용하는 인수들을 갖는다.

데이터를 송신 수신하는 방법은 간단하게 아래와 같다.

```

WSABUF DataBuf;
char SendBuf[1024] = "Test data to send.";
int BufLen = 1024;
...
DataBuf.len = BufLen;
DataBuf.buf = SendBuf;
...
WSASendTo(SendSocket,
    &DataBuf,
    1,
    &BytesSent,
    Flags,
    (SOCKADDR*)&RecvAddr,
    RecvAddrSize,
    &Overlapped,
    NULL);

```

다음 함수는 입출력 소켓의 Overlapped 결과를 확인하는 함수로

```

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

```

- s 는 입출력 소켓의 핸들이다.
- lpOverlapped 는 입출력 함수에 사용된 Overlapped 구조체의 주소다.
- lpcbTransfer 는 입출력 바이트 수를 반환하는 out parameter 다.
- fWait 는 비동기 입출력이 완료하기를 대기하기 위한 TRUE, FALSE 설정이다.
- lpdwFlags 는 입출력과 관련한 부가정보 out parameter 로 대부분 사용되지 않는다.

<절제목>

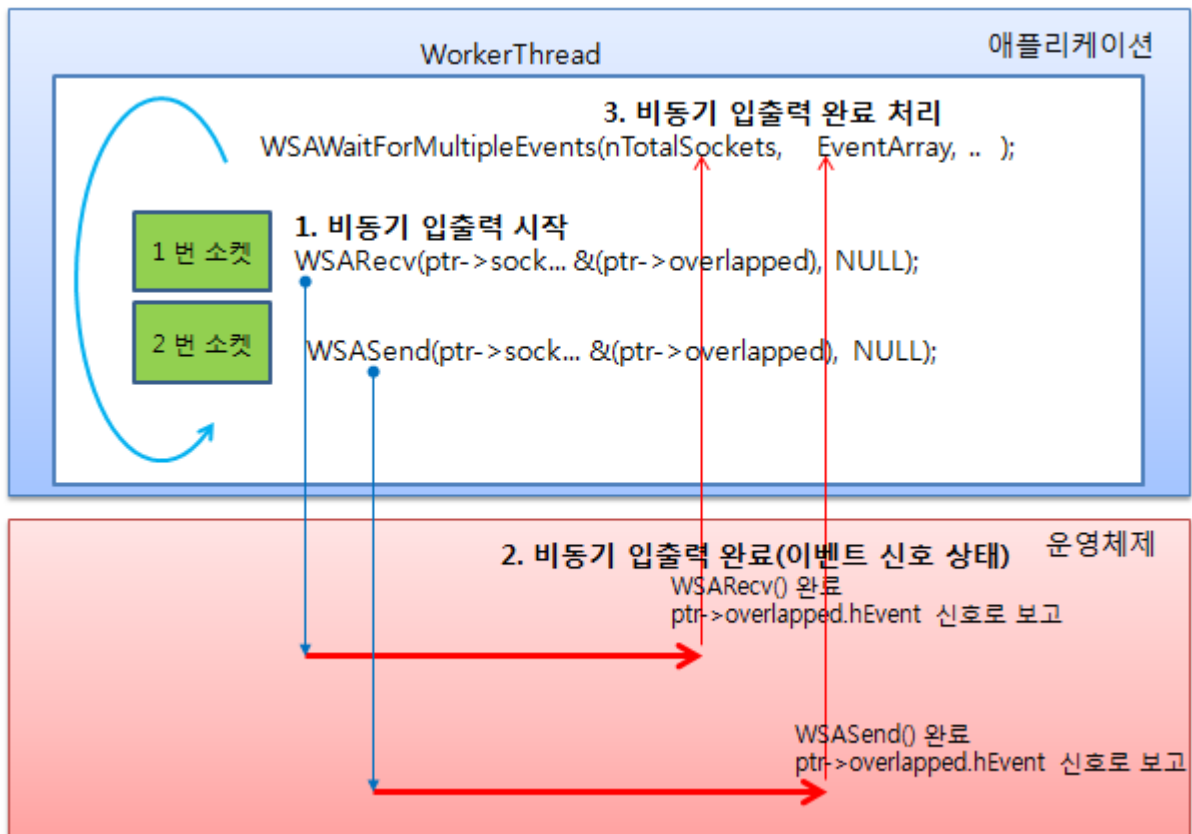
02. Overlapped 이벤트 입출력 모델 서버

</절제목>

Overlapped 입출력 모델 서버에서 클라이언트 접속을 처리하는 부분은 독립적인 대기 스레드를 생성하여 동기 입출력(`accept()`)을 사용하고 데이터 송수신(`WSASend()`, `WSARecv()`) 부분은 비동기 입출력을 사용한다. 지금까지는 클라이언트의 추가 제거 시 동기화 메커니즘을 사용하지 않았지만 Overlapped 입출력 모델에서는 클라이언트 접속 정보를 추가하는 스레드와 클라이언트 종료로 접속 정보를 제거하는 스레드에 동기화 메커니즘(크리티컬 섹션)을 사용하여 동기화한다. 하나의 스레드에서 클라이언트 정보 추가, 제거가 모두 수행된다면 동기화 메커니즘을 사용하지 않아도 되지만 이전 코드에서도 두 스레드 이상에서 클라이언트 정보 추가, 제거가 수행된다면 공유 자원(소켓 정보 배열, 커널 이벤트 배열)의 동기화를 수행하기 위한 메커니즘이 필요하다.

Overlapped 이벤트 입출력 모델은 애플리케이션에서 입출력 작업을 수행하고 운영체제에서 완료를 커널 이벤트를 통해 보고받는 형태로 동작한다.

다음 그림은 이벤트를 사용한 Overlapped 입출력 모델을 그림으로 표현한 것으로 `WSARecv()`나 `WSASend()`로 비동기 입출력을 시작하면 운영체제는 overlapped 구조체 변수에 입출력 정보를 보관했다가 비동기 입출력이 완료하면 overlapped 구조체에 지정된 커널 이벤트를 신호 상태로 변경하므로써 애플리케이션에 완료를 보고한다. 그러면 애플리케이션은 완료를 처리하기 위해 커널 이벤트 신호를 대기하며 이벤트가 신호 상태가 되면 각 입출력 완료 작업에 맞는 처리를 수행한다.



다음은 이벤트를 사용한 Overlapped 입출력 모델 에코 서버 예제다.

<코드>

[예제 07-01] OverlappedEventTCPServer

```
#include <winsock2.h>
#include <process.h>
#include <stdio.h>
```

```
#define BUFSIZE 1024
```

```
// 소켓 정보 저장을 위한 구조체
```

```
struct SOCKETINFO
```

```
{
    WSAOVERLAPPED overlapped;
    SOCKET sock;
    char buf[BUFSIZE];
    int recvbytes;
    int sendbytes;
    WSABUF wsabuf;
};
```

```
CRITICAL_SECTION cs;
SOCKET listenSock;
int nTotalSockets = 0;
SOCKETINFO *SocketInfoArray[WSA_MAXIMUM_WAIT_EVENTS];
WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
```

```
void DisplayMessage()
```

```
{
    LPVOID pMsg;
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
```



```

        FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        WSAGetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR)&pMsg,
        0, NULL);

    printf("%s\n", pMsg);
    LocalFree(pMsg);
}

// 소켓 정보를 추가한다.
BOOL AddSocketInfo(SOCKET sock)
{
    EnterCriticalSection(&cs);

    if(nTotalSockets >= WSA_MAXIMUM_WAIT_EVENTS){
        printf("[오류] 소켓 정보를 추가할 수 없습니다!\n");
        return FALSE;
    }

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

    WSAEVENT hEvent = WSACreateEvent();
    if(hEvent == WSA_INVALID_EVENT){
        DisplayMessage();
        return FALSE;
    }

    ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
    ptr->overlapped.hEvent = hEvent;
    ptr->sock = sock;
    ptr->recvbytes = 0;
    ptr->sendbytes = 0;
    ptr->wsabuf.buf = ptr->buf;
    ptr->wsabuf.len = BUFSIZE;
    SocketInfoArray[nTotalSockets] = ptr;
    EventArray[nTotalSockets] = hEvent;
    nTotalSockets++;

    LeaveCriticalSection(&cs);
    return TRUE;
}

// 소켓 정보 삭제
void RemoveSocketInfo(int nIndex)
{
    EnterCriticalSection(&cs);

    SOCKETINFO *ptr = SocketInfoArray[nIndex];
    closesocket(ptr->sock);
    delete ptr;
    WSACloseEvent(EventArray[nIndex]);

    for(int i=nIndex; i<nTotalSockets; i++){
        SocketInfoArray[i] = SocketInfoArray[i+1];
        EventArray[i] = EventArray[i+1];
    }
    nTotalSockets--;

    LeaveCriticalSection(&cs);
}

bool CreateListenSocket()
{
    int retval;

```

```

// 대기 소켓 생성
listenSock = socket(AF_INET, SOCK_STREAM, 0);
if(listenSock == INVALID_SOCKET)
{
    DisplayMessage();
    return false;
}

// 대기 소켓을 SocketInfoArray에 추가한다.
if(AddSocketInfo(listenSock) == FALSE)
    return false;

// 통신 소켓이 생성될 때까지 workerThread 임시 대기를 위한 임시 이벤트 객체 생성
WSAEVENT hEvent = WSACreateEvent();
if(hEvent == WSA_INVALID_EVENT)
{
    DisplayMessage();
    return false;
}
EventArray[nTotalSockets++] = hEvent;

// 대기 소켓의 로컬 주소, 포트 설정
SOCKADDR_IN serveraddr;
ZeroMemory(&serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(40100);
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
retval = bind(listenSock, (SOCKADDR *)&serveraddr, sizeof(serveraddr));
if(retval == SOCKET_ERROR)
{
    DisplayMessage();
    return false;
}

// 대기 소켓의 접속 대기 큐 생성 및 클라이언트 접속 대기
retval = listen(listenSock, SOMAXCONN);
if(retval == SOCKET_ERROR)
{
    DisplayMessage();
    return false;
}

return true;
}
unsigned int WINAPI WorkerThread(void* pParam)
{
    int retval;

    while(1){
        // 이벤트 객체 관찰
        DWORD index = WSAWaitForMultipleEvents(nTotalSockets,
            EventArray, FALSE, WSA_INFINITE, FALSE);
        if(index == WSA_WAIT_FAILED){
            DisplayMessage();
            continue;
        }
        index -= WSA_WAIT_EVENT_0;
        WSAResetEvent(EventArray[index]);
        if(index == 0) continue;

        // 클라이언트 정보 얻기
        SOCKETINFO *ptr = SocketInfoArray[index];
        SOCKADDR_IN clientaddr;
        int addrlen = sizeof(clientaddr);
        getpeername(ptr->sock, (SOCKADDR *)&clientaddr, &addrlen);
    }
}

```

```

// 비동기 입출력 결과 확인
DWORD cbTransferred, flags;
retval = WSAGetOverlappedResult(ptr->sock, &(ptr->overlapped),
    &cbTransferred, FALSE, &flags);
if(retval == FALSE || cbTransferred == 0){
    if(retval == FALSE)
        DisplayMessage();
    RemoveSocketInfo(index);
    printf("[TCP 서버] 클라이언트 종료: IP 주소=%s, 포트 번호=%d\n",
        inet_ntoa(clientaddr.sin_addr),
        ntohs(clientaddr.sin_port));
    continue;
}

// 데이터 전송량 갱신
if(ptr->recvbytes == 0){
    ptr->recvbytes = cbTransferred;
    ptr->sendbytes = 0;
    // 받은 데이터 출력
    ptr->buf[ptr->recvbytes] = '\0';
    printf("[TCP/%s:%d] %s\n", inet_ntoa(clientaddr.sin_addr),
        ntohs(clientaddr.sin_port), ptr->buf);
}
else{
    ptr->sendbytes += cbTransferred;
}

if(ptr->recvbytes > ptr->sendbytes){
    // 데이터 보내기
    ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
    ptr->overlapped.hEvent = EventArray[index];
    ptr->wsabuf.buf = ptr->buf + ptr->sendbytes;
    ptr->wsabuf.len = ptr->recvbytes - ptr->sendbytes;

    DWORD sendbytes;
    retval = WSASend(ptr->sock, &(ptr->wsabuf), 1, &sendbytes,
        0, &(ptr->overlapped), NULL);
    if(retval == SOCKET_ERROR){
        if(WSAGetLastError() != WSA_IO_PENDING){
            DisplayMessage();
        }
        continue;
    }
}
else{
    ptr->recvbytes = 0;

    // 데이터 받기
    ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
    ptr->overlapped.hEvent = EventArray[index];
    ptr->wsabuf.buf = ptr->buf;
    ptr->wsabuf.len = BUFSIZE;

    DWORD recvbytes;
    flags = 0;
    retval = WSAREcv(ptr->sock, &(ptr->wsabuf), 1, &recvbytes,
        &flags, &(ptr->overlapped), NULL);
    if(retval == SOCKET_ERROR){
        if(WSAGetLastError() != WSA_IO_PENDING){
            DisplayMessage();
        }
        continue;
    }
}
}

return 0;

```

```

}
unsigned int WINAPI ListenThread(void* pParam)
{
    while(1){
        // accept()
        SOCKADDR_IN clientaddr;
        int addrlen = sizeof(clientaddr);
        SOCKET client_sock = accept(listenSock, (SOCKADDR *)&clientaddr,
            &addrlen);
        if(client_sock == INVALID_SOCKET){
            DisplayMessage();
            continue;
        }
        printf("[TCP 서버] 클라이언트 접속: IP 주소=%s, 포트 번호=%d\n",
            inet_ntoa(clientaddr.sin_addr),
            ntohs(clientaddr.sin_port));

        // 소켓 정보 추가
        if(AddSocketInfo(client_sock) == FALSE){
            closesocket(client_sock);
            printf("[TCP 서버] 클라이언트 종료: IP 주소=%s, 포트 번호=%d\n",
                inet_ntoa(clientaddr.sin_addr),
                ntohs(clientaddr.sin_port));
            continue;
        }

        // 비동기 입출력 시작
        SOCKETINFO *ptr = SocketInfoArray[nTotalSockets-1];
        DWORD recvbytes;
        DWORD flags = 0;
        int retval = WSAREcv(ptr->sock, &(ptr->wsabuf), 1, &recvbytes,
            &flags, &(ptr->overlapped), NULL);
        if(retval == SOCKET_ERROR){
            if(WSAGetLastError() != WSA_IO_PENDING){
                DisplayMessage();
                RemoveSocketInfo(nTotalSockets-1);
                continue;
            }
        }

        // 소켓의 개수(nTotalSockets) 변화를 알림
        if(WSASetEvent(EventArray[0]) == FALSE){
            DisplayMessage();
            break;
        }
    }

    // 대기 소켓 닫기
    closesocket(listenSock);

    return 0;
}

int main(int argc, char* argv[])
{
    WSADATA wsa;

    InitializeCriticalSection(&cs);
    if(WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("윈도우 소켓 초기화 실패!\n");
        return -1;
    }

    // 대기 소켓 초기화(socket()+bind()+listen())
    if( !CreateListenSocket( ) )
    {

```

```

    printf("대기 소켓 생성 실패!\n");
    return -1;
}

// 대기 쓰레드 종료를 기다림.
unsigned int threadID;
HANDLE threadArray[2];
threadArray[0] = (HANDLE)_beginthreadex(0,0, ListenThread, 0, 0,
    &threadID);
threadArray[1] = (HANDLE)_beginthreadex(0,0, WorkerThread, 0, 0,
    &threadID);

WaitForMultipleObjects(2, threadArray, TRUE, INFINITE);

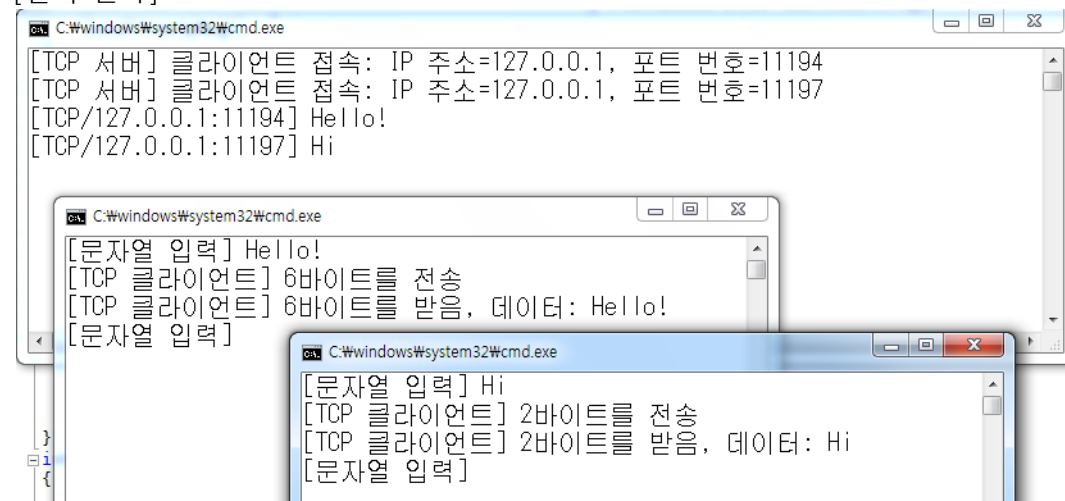
WSACleanup();
DeleteCriticalSection(&cs);
return 0;
}

```

</코드>

<결과>

[출력 결과]



</결과>

다음 코드는

```

struct SOCKETINFO
{
    WSAOVERLAPPED overlapped;
    SOCKET sock;
    char buf[BUFSIZE];
    int recvbytes;
    int sendbytes;
    WSABUF wsabuf;
};

```

소켓 정보 구조체로 하나의 소켓은 WSAOVERLAPPED 구조체 멤버를 갖는다. 에코 서버는 수신과 송신이 동시에 일어나지 않기 때문에 하나의 WSAOVERLAPPED 구조체를 사용할 수 있다. 만약 송신, 수신을 독립적으로 수행해야 한다면 독립적으로 동작하는 비동기 입출력이 두 개이므로 송, 수신 각각의 WSAOVERLAPPED 구조체를 사용해야 한다.

다음 코드는

```

CRITICAL_SECTION cs;

```

```
InitializeCriticalSection(&cs);
...
DeleteCriticalSection(&cs);
```

클라이언트 정보 추가 스레드와 클라이언트 정보 제거 스레드가 서로 다르므로 클라이언트 정보 추가, 제거를 동기화하기 위한 크리티컬 섹션을 초기화하고 마무리 하는 코드다.

다음 코드는

```
BOOL AddSocketInfo(SOCKET sock)
{
    EnterCriticalSection(&cs);

    ...

    SOCKETINFO *ptr = new SOCKETINFO;
    ...

    WSAEVENT hEvent = WSACreateEvent();
    if(hEvent == WSA_INVALID_EVENT){
        DisplayMessage();
        return FALSE;
    }

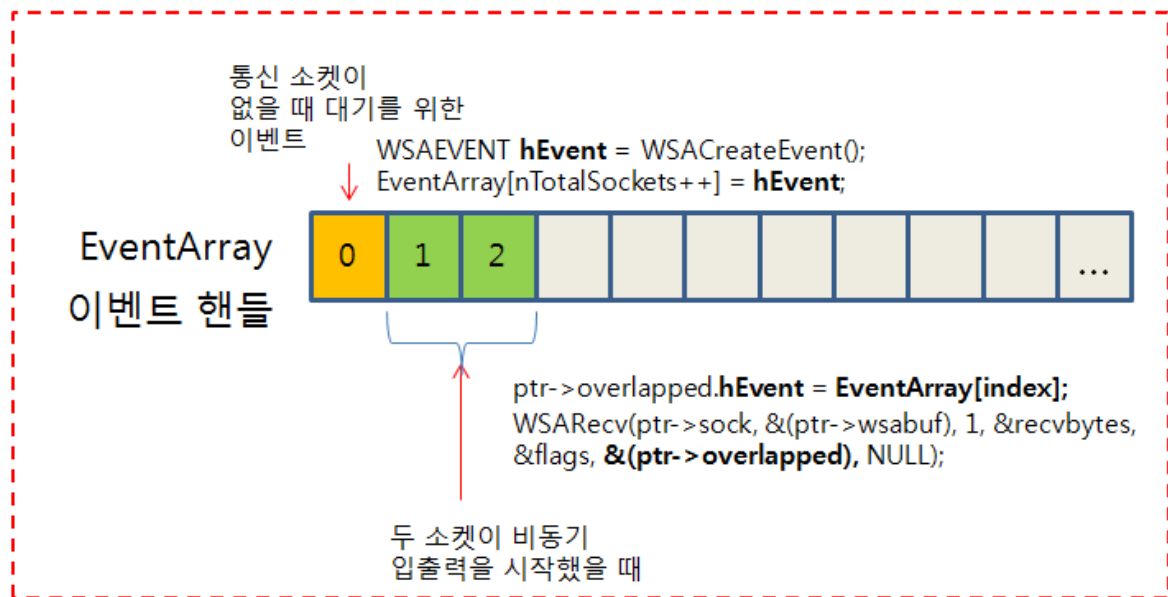
    ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
    ptr->overlapped.hEvent = hEvent;
    ptr->sock = sock;
    ptr->recvbytes = 0;
    ptr->sendbytes = 0;
    ptr->wsabuf.buf = ptr->buf;
    ptr->wsabuf.len = BUFSIZE;
    SocketInfoArray[nTotalSockets] = ptr;
    EventArray[nTotalSockets] = hEvent;
    nTotalSockets++;

    LeaveCriticalSection(&cs);
    return TRUE;
}
```

통신 소켓 정보를 생성하고 overlapped 를 사용하기 전 0으로 초기화하고 이 변수가 지정된 비동기 입출력의 완료 보고를 받기위한 커널 이벤트 hEvent 를 생성한 후 설정한다. 마지막으로 완료 신호 상태 대기를 위한 커널 이벤트 배열(EventArray)에 추가한다.

다음은 두 개의 통신 소켓이 비동기 입출력을 시작했을 때의 커널 이벤트 배열의 상태를 그림으로 표현한 것이다. 0번에는 임시 커널 이벤트가 추가되면 1, 2번은 통신 소켓의 입출력 완료를 위한 이벤트가 추가된다.

WorkerThread



다음 코드는

```
unsigned int WINAPI WorkerThread(void* pParam)
{
    int retval;
    while(1){
        // 이벤트 객체 관찰
        DWORD index = WSAWaitForMultipleEvents(nTotalSockets,
            EventArray, FALSE, WSA_INFINITE, FALSE);
        if(index == WSA_WAIT_FAILED){
            DisplayMessage();
            continue;
        }
        index -= WSA_WAIT_EVENT_0;
        WSAResetEvent(EventArray[index]);
        if(index == 0) continue;
        ...
    }
}
```

비동기 입출력의 이벤트를 대기하는 코드로 index 가 0인 경우는 0 번째 인덱스의 임시 커널 이벤트가 신호 상태로 소켓의 개수가 변경되어(클라이언트가 접속되었을 때) WSAWaitForMultipleEvents()를 다시 실행하여 개수 변경을 적용해야 함을 나타낸다. 또 모든 커널 이벤트는 WSAResetEvent()를 사용하여 다시 비신호 상태로 변경한다.

다음 코드는

```
DWORD cbTransferred, flags;
retval = WSAGetOverlappedResult(ptr->sock, &(ptr->overlapped),
    &cbTransferred, FALSE, &flags);
if(retval == FALSE || cbTransferred == 0){
    if(retval == FALSE)
        DisplayMessage();
    RemoveSocketInfo(index);
}
```

```

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

```

소켓의 입출력 정보를 확인하는 코드로 WSAGetOverlappedResult()에서 cbTransferred 에 입출력 바이트 수를 반환한다. 만약 retval 이 FALSE 이면 오류 처리를 하며 cbTransferred 가 0이면 정상 종료이므로 종료 처리한다.

다음 코드는

```

if(ptr->recvbytes > ptr->sendbytes){
    // 데이터 보내기
    ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
    ptr->overlapped.hEvent = EventArray[index];
    ptr->wsabuf.buf = ptr->buf + ptr->sendbytes;
    ptr->wsabuf.len = ptr->recvbytes - ptr->sendbytes;

    DWORD sendbytes;
    retval = WSASend(ptr->sock, &(ptr->wsabuf), 1, &sendbytes,
                     0, &(ptr->overlapped), NULL);
    if(retval == SOCKET_ERROR){
        if(WSAGetLastError() != WSA_IO_PENDING){
            DisplayMessage();
        }
        continue;
    }
}

```

보낼 바이트가 있다면 데이터를 보내기 위해 비동기 입출력(WSASend())을 시작한다. 입출력은 운영체제가 담당하며 그 정보는 &(ptr->overlapped)를 사용하여 입출력이 완료하면 ptr->overlapped.hEvent 에 설정한 커널 이벤트를 신호 상태로 변경하여 애플리케이션에 알린다.

WSASend() 함수를 실행하고 비동기 입출력이 바로 완료하지 않는다면 오류를 반환하고 오류값이 WSA_IO_PENDING 이다. 이때는 정상적인 동작이므로(운영체제가 입출력을 수행하고 있으므로) 다음 작업을 수행한다.

다음 코드는

```

else{

    ptr->recvbytes = 0;

    // 데이터 받기
    ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
    ptr->overlapped.hEvent = EventArray[index];
    ptr->wsabuf.buf = ptr->buf;
    ptr->wsabuf.len = BUFSIZE;

    DWORD recvbytes;
    flags = 0;
    retval = WSAREcv(ptr->sock, &(ptr->wsabuf), 1, &recvbytes,
                     &flags, &(ptr->overlapped), NULL);
    if(retval == SOCKET_ERROR){
        if(WSAGetLastError() != WSA_IO_PENDING){
            DisplayMessage();
        }
        continue;
    }
}

```


}
 데이터 수신을 위한 비동기 입출력을 시작하는 코드로 `&(ptr->overlapped)`을 사용하여 입출력
 완료 보고를 받는다. 데이터 수신을 시작하고 바로 완료하지 않는다면 오류값이
`WSA_IO_PENDING` 이며(운영체제가 데이터 수신 작업을 수행한다) 정상적인 입출력 작업을
 수행하고 있으므로 다음 작업을 수행한다.

다음 코드는

```

unsigned int WINAPI ListenThread(void* pParam)
{
    while(1){
        ...
        SOCKET client_sock = accept(listenSock, (SOCKADDR *)&clientaddr,
                                     &addrlen);
        if(AddSocketInfo(client_sock) == FALSE){
            ...
        }

        ...
        int retval = WSAREcv(ptr->sock, &(ptr->wsabuf), 1, &recvbytes,
                             &flags, &(ptr->overlapped), NULL);
        if(retval == SOCKET_ERROR){
            if(WSAGetLastError() != WSA_IO_PENDING){
                DisplayMessage();
                RemoveSocketInfo(nTotalSockets-1);
                continue;
            }
        }

        if(WSASetEvent(EventArray[0]) == FALSE){
            DisplayMessage();
            break;
        }
    }
    ...
    return 0;
}
  
```

클라이언트 접속을 처리하기 위한 대기 스레드로 클라이언트가 접속하면 소켓 정보와 커널
 이벤트를 배열에 추가하고 비동기 데이터 수신 작업을 시작한다.(`WSAREcv()` 함수) 이처럼
`Overlapped` 모델은 입출력을 비동기적으로(운영체제로) 먼저 수행하고 입출력 작업이 완료하면
 애플리케이션이 보고 받는 형태로 동작한다.

마지막으로 `WSASetEvent(EventArray[0])`를 사용하여 클라이언트가 접속하여 소켓의 변경을
`WorketThread`에 알리고 `WSAWaitForMultipleEvents()` 함수가 다시 실행되어 소켓의 개수를
 갱신하여 대기하도록 한다.

<절제목>

03. Overlapped 완료 루틴(Completion Routine) 입출력 모델 서버

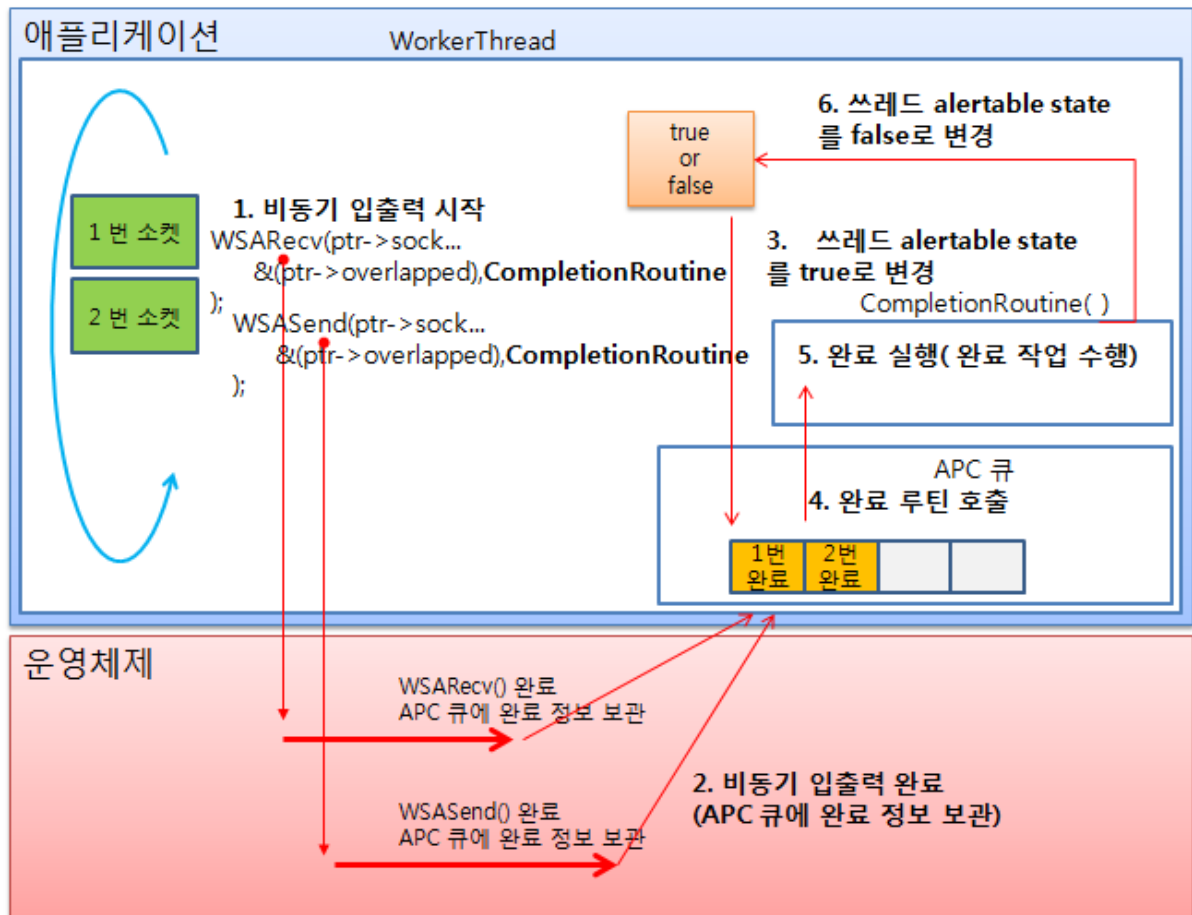
</절제목>

완료 루틴을 사용한 Overlapped 모델은 이벤트를 사용한 Overlapped 모델보다 조금더 효율적으로 동작한다. 이벤트 Overlapped 모델은 운영체제의 비동기 입출력 완료 보고를 확인하기 위해 애플리케이션이 직접 입출력 완료를 위한 이벤트를 관리해야 하기 때문이다. 완료 루틴을 사용한 Overlapped 모델은 운영체제가 입출력 작업을 완료하면 자동으로 애플리케이션의 콜백 함수(완료 루틴)를 호출하게 되므로 좀더 효율적으로 동작한다.

여기서 주의할 점은 운영체제가 자동으로 완료 루틴을 호출한다 하더라도 운영체제가 애플리케이션 모르게 아무때나 콜백 함수(완료 루틴)를 호출할 수는 없다. 당연한 이야기지만 애플리케이션도 어떤 작업을 수행하고 있을 것이기 때문에 애플리케이션이 완료 작업을 처리할 수 있는 준비가 되었을 때 운영체제가 콜백 함수(완료 루틴)를 호출해야 한다.

이때 애플리케이션은 완료 작업을 처리할 준비가 되었다는 것을 운영체제에 알리기위한 방법이 필요한데 스레드를 알림 가능 상태(alertable state)로 설정함으로서 가능하다. 여기서 애플리케이션이 자신의 스레드를 알림 가능 상태로 설정하면 완료된 입출력 작업이 존재할 때 운영체제는 애플리케이션의 완료 루틴 함수를 호출하게 되고 알림 가능 상태로 만든 스레드가 이 완료 루틴 함수를 실행한다. 운영체제는 입출력이 완료하더라도 애플리케이션이 보고받을 준비가 될 때까지(스레드가 알림 가능 상태가 될 때까지) 완료 정보를 보관할 장소가 필요하며 이 장소를 APC 큐(Asynchronous Procedure Call Queue)라 한다.

다음은 완료 루틴이 동작하는 절차를 그림으로 표현한 것이다.



1. 비동기 입출력을 시작한다. 이때 WSARecv()와 WSASend() 마지막 인수로 완료 루틴(CompletionRoutine)의 주소를 설정한다.
2. 비동기 입출력이 완료되면 운영체제가 APC 큐에 완료 정보를 보관한다.
3. 애플리케이션이 완료 보고를 받을 준비가 되었음을(alertable state를 true) 알린다.
4. 자동으로 완료 루틴을 호출한다.
5. 알림 가능 상태인 쓰레드가 APC 큐에 있는 모든 완료 정보를 처리한다.
6. 모든 완료 정보가 처리되면 애플리케이션의 쓰레드는 다시 alertable state를 false로 변경된다.

쓰레드를 알림 가능 상태(alertable state를 true)로 만들기 위해서는 몇 가지 다음과 같은 함수를 사용할 수 있다.

- WaitForSingleObjectEx()
- WaitForMultipleObjectsEx()
- SleepEx()
- WSAWaitForMultipleEvents() 등

다음 함수는

```
DWORD result = WaitForSingleObjectEx(..., TRUE);
```

WaitForSingleObject() 함수와 비슷하게 동작하는 함수로 마지막 인수만 TRUE 로 설정하여 이 함수를 호출하는 스레드를 알림 가능 상태로 만든다. 알림 가능 상태일 때 완료 루틴이 호출되어 완료 작업을 모두 처리하면 이 함수는 WAIT_IO_COMPLETION 값을 반환한다.

완료 루틴 함수는

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

- dwError 는 오류 값으로 0이면 오류가 아니다.
- cbTransferred 는 완료 바이트 수다.
- lpOverlapped 는 비동기 입출력 함수의 인수로 전달한 WSAOVERLAPPED 구조체의 주소로 입출력 정보에 대한 부가 정보를 덧붙여 사용되기도 한다.
- dwFlags 은 사용하지 않는다.

비동기 입출력 함수의 WSAREcv(), WSARecv() 등의 마지막 인수로 전달한다.

다음은 완료 루틴을 사용한 Overlapped 입출력 모델 에코 서버 예제다.

<코드>

[예제 07-02] OverlappedCompletionRoutineTCPServer

```
#include <winsock2.h>  
#include <process.h>  
#include <stdio.h>
```

```
#define BUFSIZE 1024
```

// 소켓 정보 저장을 위한 구조체

```
struct SOCKETINFO  
{  
    WSAOVERLAPPED overlapped;  
    SOCKET sock;  
    char buf[BUFSIZE];  
    int recvbytes;  
    int sendbytes;  
    WSABUF wsabuf;  
};
```

```
SOCKET listenSock;  
SOCKET clientSock;  
HANDLE hEvent;
```

```
void DisplayMessage()  
{  
    LPVOID pMsg;  
    FormatMessage(  
        FORMAT_MESSAGE_ALLOCATE_BUFFER |  
        FORMAT_MESSAGE_FROM_SYSTEM,  
        NULL,  
        WSAGetLastError(),  
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),  
        (LPTSTR)&pMsg,
```

```

        0, NULL);

    printf("%s\n", pMsg);
    LocalFree(pMsg);
}

// 완료 루틴
void CALLBACK CompletionRoutine(
    DWORD dwError, DWORD cbTransferred,
    LPWSAOVERLAPPED lpOverlapped, DWORD dwFlags)
{
    int retval;

    // 클라이언트 정보 얻기
    SOCKETINFO *ptr = (SOCKETINFO *)lpOverlapped;
    SOCKADDR_IN clientaddr;
    int addrLen = sizeof(clientaddr);
    getpeername(ptr->sock, (SOCKADDR *)&clientaddr, &addrLen);

    // 비동기 입출력 결과 확인
    if(dwError != 0 || cbTransferred == 0){
        if(dwError != 0) DisplayMessage();
        closesocket(ptr->sock);
        printf("[TCP 서버] 클라이언트 종료: IP 주소=%s, 포트 번호=%d\n",
            inet_ntoa(clientaddr.sin_addr),
            ntohs(clientaddr.sin_port));
        delete ptr;
        return;
    }

    // 데이터 전송량 갱신
    if(ptr->recvbytes == 0){
        ptr->recvbytes = cbTransferred;
        ptr->sendbytes = 0;
        // 받은 데이터 출력
        ptr->buf[ptr->recvbytes] = '\0';
        printf("[TCP/%s:%d] %s\n", inet_ntoa(clientaddr.sin_addr),
            ntohs(clientaddr.sin_port), ptr->buf);
    }
    else{
        ptr->sendbytes += cbTransferred;
    }

    if(ptr->recvbytes > ptr->sendbytes){
        // 데이터 보내기
        ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
        ptr->wsabuf.buf = ptr->buf + ptr->sendbytes;
        ptr->wsabuf.len = ptr->recvbytes - ptr->sendbytes;

        DWORD sendbytes;
        retval = WSASend(ptr->sock, &(ptr->wsabuf), 1, &sendbytes,
            0, &(ptr->overlapped), CompletionRoutine);
        if(retval == SOCKET_ERROR){
            if(WSAGetLastError() != WSA_IO_PENDING){
                DisplayMessage();
                return;
            }
        }
    }
    else{
        ptr->recvbytes = 0;

        // 데이터 받기
        ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
        ptr->wsabuf.buf = ptr->buf;
        ptr->wsabuf.len = BUFSIZE;
    }
}

```

```

        DWORD recvbytes;
        DWORD flags = 0;
        retval = WSARecv(ptr->sock, &(ptr->wsabuf), 1, &recvbytes,
            &flags, &(ptr->overlapped), CompletionRoutine);
        if(retval == SOCKET_ERROR){
            if(WSAGetLastError() != WSA_IO_PENDING){
                DisplayMessage();
                return;
            }
        }
    }
}

// 작업 쓰레드
unsigned int WINAPI WorkerThread(void* pParam)
{
    HANDLE hEvent = (HANDLE)pParam;
    int retval;

    while(1){
        while(1){
            // alertable wait
            DWORD result = WaitForSingleObjectEx(hEvent,
                INFINITE, TRUE);
            if(result == WAIT_OBJECT_0) break;
            if(result != WAIT_IO_COMPLETION) return -1;
        }

        // 접속한 클라이언트 정보 출력
        SOCKADDR_IN clientaddr;
        int addrlen = sizeof(clientaddr);
        getpeername(clientSock, (SOCKADDR *)&clientaddr, &addrlen);
        printf("[TCP 서버] 클라이언트 접속: IP 주소=%s, 포트 번호=%d\n",
            inet_ntoa(clientaddr.sin_addr),
            ntohs(clientaddr.sin_port));

        // 소켓 정보 구조체 할당과 초기화
        SOCKETINFO *ptr = new SOCKETINFO;
        if(ptr == NULL){
            printf("[오류] 메모리가 부족합니다!\n");
            return -1;
        }
        ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
        ptr->sock = clientSock;
        ptr->recvbytes = 0;
        ptr->sendbytes = 0;
        ptr->wsabuf.buf = ptr->buf;
        ptr->wsabuf.len = BUFSIZE;

        // 비동기 입출력 시작
        DWORD recvbytes;
        DWORD flags = 0;
        retval = WSARecv(ptr->sock, &(ptr->wsabuf), 1, &recvbytes,
            &flags, &(ptr->overlapped), CompletionRoutine);
        if(retval == SOCKET_ERROR){
            if(WSAGetLastError() != WSA_IO_PENDING){
                DisplayMessage();
                return -1;
            }
        }
    }

    return 0;
}

bool CreateListenSocket()
{

```

```

int retval;

// 대기 소켓 생성
listenSock = socket(AF_INET, SOCK_STREAM, 0);
if(listenSock == INVALID_SOCKET)
{
    DisplayMessage();
    return false;
}

// 임시 이벤트 객체 생성
hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
if(hEvent == NULL) false;

// 대기 소켓의 로컬 주소, 포트 설정
SOCKADDR_IN serveraddr;
ZeroMemory(&serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(40100);
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
retval = bind(listenSock, (SOCKADDR *)&serveraddr, sizeof(serveraddr));
if(retval == SOCKET_ERROR)
{
    DisplayMessage();
    return false;
}

// 대기 소켓의 접속 대기 큐 생성 및 클라이언트 접속 대기
retval = listen(listenSock, SOMAXCONN);
if(retval == SOCKET_ERROR)
{
    DisplayMessage();
    return false;
}

return true;
}
int main(int argc, char* argv[])
{
    WSADATA wsa;

    if(WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("윈도우 소켓 초기화 실패!\n");
        return -1;
    }

    // 대기 소켓 초기화(socket()+bind()+listen())
    if( !CreateListenSocket( ) )
    {
        printf("대기 소켓 생성 실패!\n");
        return -1;
    }

    // 대기 스레드 종료를 기다림.
    unsigned int threadID;
    CloseHandle((HANDLE)_beginthreadex(0,0, workerThread, (void*)hEvent, 0,
        &threadID));

    while(1){
        // accept()
        clientSock = accept(listenSock, NULL, NULL);
        if(clientSock == INVALID_SOCKET){
            DisplayMessage();
            continue;
        }
    }
}

```

```

    }
    if(!SetEvent(hEvent)) break;
}

```

```

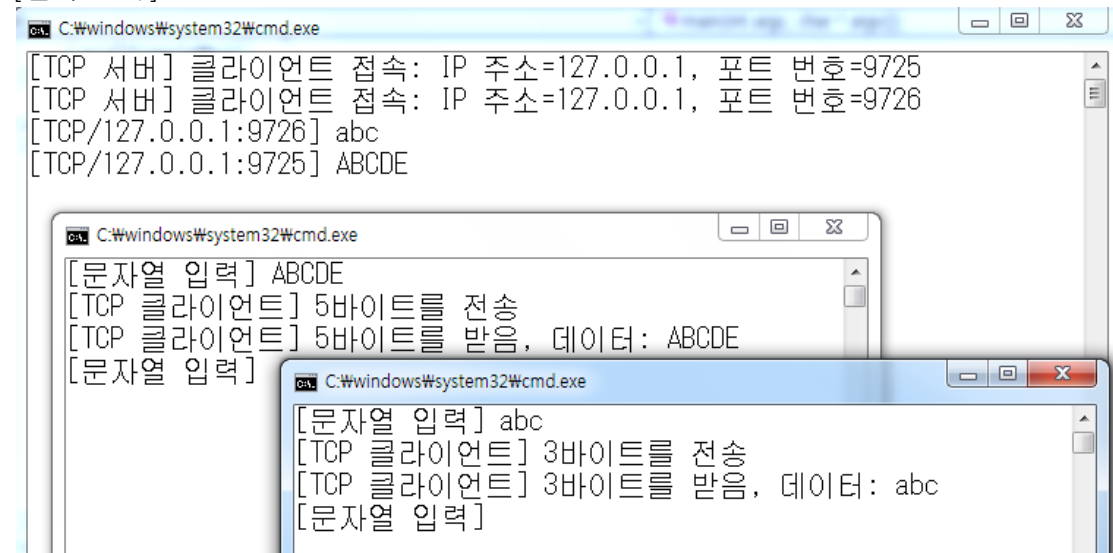
// 윈속 종료
WSACleanup();
return 0;
}

```

</코드>

<결과>

[출력 결과]



</결과>

다음 구조체는

```

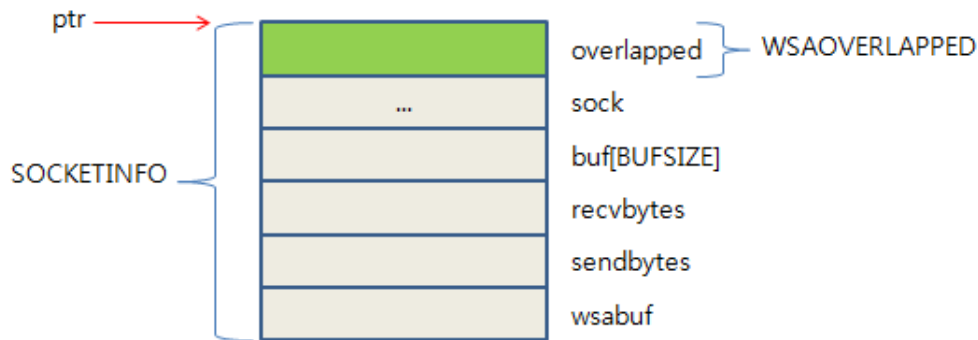
struct SOCKETINFO
{
    WSAOVERLAPPED overlapped;
    SOCKET sock;
    char buf[BUFSIZE];
    int recvbytes;
    int sendbytes;
    WSABUF wsabuf;
};

```

소켓 정보를 보관하는 구조체로 이전과 다른 점은 overlapped 구조체의 주소를 구조체와 연결하여 사용한다는 것이다. 비동기 입출력 함수의 마지막 인수로 overlapped 구조체의 주소가 지정되며 완료 루틴(Completion Routine)이 호출될 때 입출력 함수의 지정했던 overlapped 구조체의 주소가 완료 루틴 콜백 함수의 세 번째 인수로도 전달되므로 이 주소를 이용하여 구조체 정보를 완료 루틴에서 얻어 사용한다. 그래서 당연히 overlapped 멤버를 소켓 정보 구조체의 첫 번째 멤버로 사용한다.

다음 그림처럼 overlapped 구조체의 주소를 입출력 함수에 지정하면 overlapped 멤버의 주소는 곧 SOCKETINFO 구조체의 시작 주소이므로 완료 루틴에서 SOCKETINFO 구조체로 형식 변환하여 모든 멤버를 접근할 수 있다.


```
SOCKETINFO *ptr = (SOCKETINFO *)lpOverlapped;
```



다음 코드는

```
void CALLBACK CompletionRoutine(
    DWORD dwError, DWORD cbTransferred,
    LPWSAOVERLAPPED lpOverlapped, DWORD dwFlags)
{
    int retval;

    // 클라이언트 정보 얻기
    SOCKETINFO *ptr = (SOCKETINFO *)lpoverlapped;
    ...
    // 비동기 입출력 결과 확인
    if(dwError != 0 || cbTransferred == 0){
        if(dwError != 0) DisplayMessage();
        closesocket(ptr->sock);
        ...
    }
}
```

완료 루틴이 호출되면 입출력 소켓의 정보를 확인하기 위해 overlapped 멤버의 주소를 SOCKETINFO 구조체로 형식 변환하며 오류가 발생하거나 정상적인 종료(cbTransferred==0)이면 소켓을 마무리 작업한다.

다음 코드는

```
if(ptr->recvbytes == 0){
    ptr->recvbytes = cbTransferred;
    ptr->sendbytes = 0;
    // 받은 데이터 출력
    ptr->buf[ptr->recvbytes] = '\0';
    printf("[TCP/%s:%d] %s\n", inet_ntoa(clientaddr.sin_addr),
        ntohs(clientaddr.sin_port), ptr->buf);
}
else{
    ptr->sendbytes += cbTransferred;
}
```

recvbytes 가 0이면(데이터를 수신했다면) 데이터를 수신한 상태이므로 받은 데이터를 화면에 출력하고 아니면 보낸 데이터를 계산한다.

다음 코드는

```

if(ptr->recvbytes > ptr->sendbytes){
    // 데이터 보내기
    ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
    ptr->wsabuf.buf = ptr->buf + ptr->sendbytes;
    ptr->wsabuf.len = ptr->recvbytes - ptr->sendbytes;

    DWORD sendbytes;
    retval = WSASend(ptr->sock, &(ptr->wsabuf), 1, &sendbytes,
        0, &(ptr->overlapped), CompletionRoutine);
    if(retval == SOCKET_ERROR){
        if(WSAGetLastError() != WSA_IO_PENDING){
            DisplayMessage();
            return;
        }
    }
}

```

받은 데이터를 아직 덜 보냈다면 남은 데이터를 비동기 입출력을 사용하여 보내기를 시작하며

```

else{
    ptr->recvbytes = 0;

    // 데이터 받기
    ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
    ptr->wsabuf.buf = ptr->buf;
    ptr->wsabuf.len = BUFSIZE;

    DWORD recvbytes;
    DWORD flags = 0;
    retval = WSAREcv(ptr->sock, &(ptr->wsabuf), 1, &recvbytes,
        &flags, &(ptr->overlapped), CompletionRoutine);
    if(retval == SOCKET_ERROR){
        if(WSAGetLastError() != WSA_IO_PENDING){
            DisplayMessage();
            return;
        }
    }
}

```

받은 데이터를 모두 보냈다면 recvbytes 를 초기화하고 다시 데이터 수신을 위한 비동기 입출력 함수를 시작한다.

다음 코드는

```

hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
if(hEvent == NULL) false;

```

임시 커널 이벤트 객체를 생성하여 이 이벤트는 클라이언트 접속이 발생함을 알리는데 사용된다.

다음 코드는

```

while(1){
    DWORD result = WaitForSingleObjectEx(hEvent,
        INFINITE, TRUE);
    if(result == WAIT_OBJECT_0) break;
    if(result != WAIT_IO_COMPLETION) return -1;
}

```

임시 커널 이벤트가 신호 상태가 되는지를(클라이언트 접속) 판단하고 스레드를 알림 가능 상태로 만드는 코드로 완료 루틴이 실행되면 반환되고 WAIT_IO_COMPLETION 값이면 다시 스레드를 알림 가능 상태로 만든다. 만약 WAIT_OBJECT_0이면 클라이언트가 접속했으므로 접속을 위한 아래 코드를 수행한다.

다음 코드는

```
SOCKETINFO *ptr = new SOCKETINFO;
if(ptr == NULL){
    printf("[오류] 메모리가 부족합니다!\n");
    return -1;
}
ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
ptr->sock = clientSock;
ptr->recvbytes = 0;
ptr->sendbytes = 0;
ptr->wsabuf.buf = ptr->buf;
ptr->wsabuf.len = BUFSIZE;

// 비동기 입출력 시작
DWORD recvbytes;
DWORD flags = 0;
retval = WSAREcv(ptr->sock, &(ptr->wsabuf), 1, &recvbytes,
    &flags, &(ptr->overlapped), CompletionRoutine);
if(retval == SOCKET_ERROR){
    if(WSAGetLastError() != WSA_IO_PENDING){
        DisplayMessage();
        return -1;
    }
}
```

클라이언트가 접속되었으므로 소켓의 정보 메모리를 생성하고 데이터 수신을 위한 비동기 입출력을 시작한다. WSAREcv() 함수의 마지막 인수가 입출력 작업이 완료하면 실행할 완료 루틴의 콜백 함수 주소다.

다음 코드는

```
while(1){
    // accept()
    clientSock = accept(listenSock, NULL, NULL);
    if(clientSock == INVALID_SOCKET){
        DisplayMessage();
        continue;
    }
    if(!SetEvent(hEvent)) break;
}
```

클라이언트 접속을 처리하기 위한 코드로 Main 스레드에서 루프를 이용하여 accept()를 처리하고 SetEvent(hEvent)로 클라이언트 접속을 WorkerThread 에 알려 클라이언트 접속 처리를 수행하도록 한다.

<장제목>

8. IOCP(I/O Completion Port) 입출력 모델

</장제목>

IOCP(I/O Completion Port) 입출력 모델은 대용량의 비동기 입출력 위한 모델로 가장 성능이 뛰어나다고 알려져 있다.

IOCP 가 Overlapped 모델과 다른 점은 Overlapped 모델에서 APC 큐에 보관된 완료 정보는 APC 큐를 소유한 스레드만 확인하고 처리할 수 있지만 IOCP 는 어떤 스레드든지 IOCP 핸들을 이용하여 완료 처리를 수행할 수 있다. 그러므로 많은 입출력 완료처리를 여러 스레드로 나누어 처리할 수 있다. IOCP 는 완료 작업을 처리하기 위해 스레드를 알림 가능 상태로 만들지 않는다. 단지 IOCP 핸들을 이용하여 IOCP 에 보관된 완료 정보를 GetQueuedCompletionStatus() 함수를 이용하여 읽어가고 처리한다.

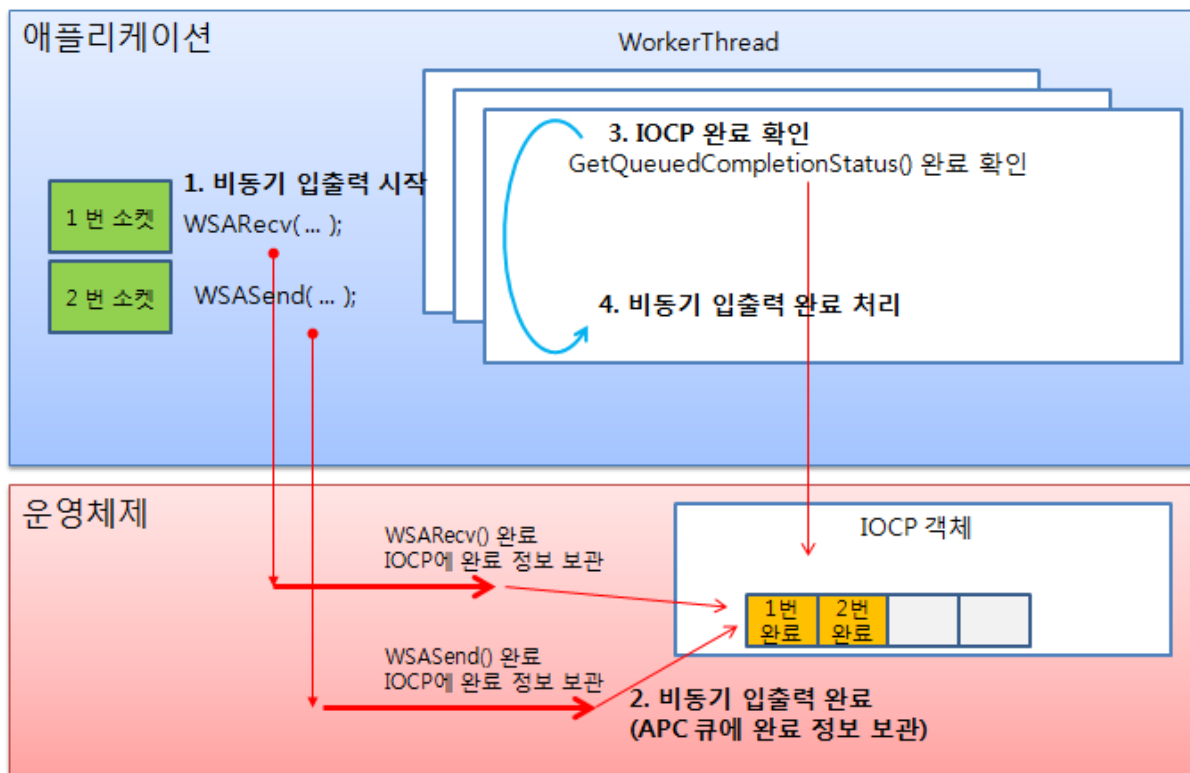
<절제목>

01. IOCP 모델의 이해

</절제목>

IOCP 모델은 윈도우 운영체제가 비동기 입출력에 대한 완료 처리를 수행하는 객체(IOCP)를 상용하여 동작한다. 입출력에 참여하는 모든 소켓은 IOCP 와 연결되며 입출력이 완료되면 IOCP 에 완료가 정보가 보관되며 애플리케이션에서는 언제든지 이 IOCP 를 확인하고 완료 정보를 얻어 처리할 수 있는 구조를 갖는다.

다음 그림은 IOCP 모델이 동작하는 절차를 표현한 그림이다.



1. IOCP 후 입출력에 참여하는 모든 소켓을 IOCP 와 연결하고 비동기 입출력을 시작한다.
2. 비동기 입출력이 완료되면 완료 정보가 IOCP 에 보관된다.
3. 애플리케이션은 적절한 스레드를 생성하여 IOCP 에 비동기 입출력이 완료하는지 확인한다.
4. 비동기 입출력이 완료되는 것을 확인하면 완료 처리 작업을 진행한다.

IOCP 를 사용하기 위해서는 가장 먼저 IOCP 객체를 생성해야 한다.

다음은 IOCP 객체를 생성하고 IOCP 와 소켓을 연결하는데 사용하는 함수다.

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

- FileHandle 은 입출력 완료 포트와 연결할 파일이나 소켓 등의 핸들이며 IOCP 를 생성할 때는 INVALID_HANDLE_VALUE 값을 지정한다.
- ExistingCompletionPort 은 파일, 소켓 등을 연결할 IOCP 핸들이다.
- CompletionKey 는 입출력 작업이 완료할 때 전달되는 4바이트 key 값으로 소켓과 IOCP 객체를 연결할 때 지정한 값을 GetQueuedCompletionStatus() 함수에서 확인 할 수 있다. 일반적으로 완료 작업 등을 구분하는 key 값으로 사용된다.
- NumberOfConcurrentThreads 는 동시에 실행할 수 있는 작업자 스레드의 개수로 GetQueuedCompletionStatus() 함수로 블로킹 상태에 빠진 스레드가 IOCP 에 입출력 완료가 발생하면 이 인수만큼의 스레드가 블로킹 상태에서 깨어 활동한다. 0으로 설정하면 모든 스레드가 깨어 활동한다.

CreateIoCompletionPort()는 IOCP 객체를 생성할 때나 소켓과 연결할 때 사용되며 IOCP 객체를 생성할 때는 다음과 같다.

```
HANDLE hcp = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
```

첫 인수를 INVALID_HANDLE_VALUE 로 설정하고 NULL, 0으로 초기화한다.

소켓과 연결할 때는 다음과 같다.

```
HANDLE hResult = CreateIoCompletionPort((HANDLE)client_sock, hcp,
    (DWORD)client_sock, 0);
```

첫 인수는 client_sock 은 IOCP 와 연결할 소켓의 핸들이고 두 번째 인수 hcp 는 생성한 IOCP 객체의 핸들이며 세 번째 인수는 입출력 Key 값을 소켓 핸들 번호로 설정한다. 마지막 인수가 0이므로 모든 워커 스레드(GetQueuedCompletionPort()를 호출한)를 깨운다.

다음 함수는 IOCP 에 완료 정보를 확인하는 함수로 IOCP 에 완료 정보가 없으면 스레드는 블로킹 상태가 된다.

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

- CompletionPort 는 IOCP 핸들이다.

- lpNumberOfBytes 는 입출력 완료 바이트 수를 반환하는 out parameter 다.
- lpCompletionKey 는 입출력을 구분하는데 사용하는 Key 값이다.
- lpOverlapped 는 비동기 입출력 함수에 지정된 WSAOVERLAPPED 구조체 주소다.
- dwMilliseconds 는 타임 아웃을 설정한다. INFINITE 는 완료될 때까지 무한정 대기한다.

다음 함수는 IOCP 에 입출력 정보를 수동으로 보관할 수 있는 함수로

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

인수는 GetQueuedCompletionStatus() 함수와 같다.

<절제목>

02. IOCP 입출력 모델 에코 서버

</절제목>

IOCP 모델은 윈도우 운영체제가 비동기 입출력에 대한 완료 처리를 수행하는 객체(IOCP)를 상용하여 동작한다. 입출력에 참여하는 모든 소켓은 IOCP 와 연결되며 입출력이 완료되면 IOCP 에 완료가 정보가 보관되며 애플리케이션에서는 언제든지 이 IOCP 를 확인하고 완료 정보를 얻어 처리할 수 있는 구조를 갖는다.

다음은 IOCP 입출력 모델을 사용한 에코 서버 예제다.

<코드>

```
[예제 08-01] IOCP TCPServer
#include <winsock2.h>
#include <process.h>
#include <stdio.h>

#define BUFSIZE 1024

// 소켓 정보 저장을 위한 구조체
struct SOCKETINFO
{
    WSAOVERLAPPED overlapped;
    SOCKET sock;
    char buf[BUFSIZE];
    int recvbytes;
    int sendbytes;
    WSABUF wsabuf;
};

SOCKET listenSock;

void DisplayMessage()
{
    LPVOID pMsg;
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        WSAGetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR)&pMsg,
        0, NULL);

    printf("%s\n", pMsg);
    LocalFree(pMsg);
}

// 작업 스레드
unsigned int WINAPI workerThread(void* pParam)
{
    HANDLE hcp = (HANDLE)pParam;
    int retval;

    while(1){
```



```

// 비동기 입출력 완료 기다리기
DWORD cbTransferred;
SOCKET client_sock;
SOCKETINFO *ptr;
retval = GetQueuedCompletionStatus(hcp, &cbTransferred,
    (LPDWORD)&client_sock, (LPOVERLAPPED *)&ptr, INFINITE);

// 클라이언트 정보 얻기
SOCKADDR_IN clientaddr;
int addrlen = sizeof(clientaddr);
getpeername(ptr->sock, (SOCKADDR *)&clientaddr, &addrlen);

// 비동기 입출력 결과 확인
if(retval == 0 || cbTransferred == 0)
{
    if(retval == 0){
        DWORD temp1, temp2;
        WSAGetOverlappedResult(ptr->sock,
            &(ptr->overlapped), &temp1, FALSE, &temp2);
        DisplayMessage();
    }
    closesocket(ptr->sock);
    printf("[TCP 서버] 클라이언트 종료: IP 주소=%s, 포트 번호=%d\n",
        inet_ntoa(clientaddr.sin_addr),
        ntohs(clientaddr.sin_port));
    delete ptr;
    continue;
}

// 데이터 전송량 갱신
if(ptr->recvbytes == 0)
{
    ptr->recvbytes = cbTransferred;
    ptr->sendbytes = 0;
    // 받은 데이터 출력
    ptr->buf[ptr->recvbytes] = '\0';
    printf("[TCP/%s:%d] %s\n", inet_ntoa(clientaddr.sin_addr),
        ntohs(clientaddr.sin_port), ptr->buf);
}
else
{
    ptr->sendbytes += cbTransferred;
}

if(ptr->recvbytes > ptr->sendbytes)
{
    // 데이터 보내기
    ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
    ptr->wsabuf.buf = ptr->buf + ptr->sendbytes;
    ptr->wsabuf.len = ptr->recvbytes - ptr->sendbytes;

    DWORD sendbytes;
    retval = WSASend(ptr->sock, &(ptr->wsabuf), 1,
        &sendbytes, 0, &(ptr->overlapped), NULL);
    if(retval == SOCKET_ERROR){
        if(WSAGetLastError() != WSA_IO_PENDING){
            DisplayMessage();
        }
        continue;
    }
}
else
{
    ptr->recvbytes = 0;

    // 데이터 받기

```

```

        ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
        ptr->wsabuf.buf = ptr->buf;
        ptr->wsabuf.len = BUFSIZE;

        DWORD recvbytes;
        DWORD flags = 0;
        retval = WSAREcv(ptr->sock, &(ptr->wsabuf), 1,
            &recvbytes, &flags, &(ptr->overlapped), NULL);
        if(retval == SOCKET_ERROR)
        {
            if(WSAGetLastError() != WSA_IO_PENDING){
                DisplayMessage();
            }
            continue;
        }
    }
}

return 0;
}

bool CreateListenSocket()
{
    int retval;

    // 대기 소켓 생성
    listenSock = socket(AF_INET, SOCK_STREAM, 0);
    if(listenSock == INVALID_SOCKET)
    {
        DisplayMessage();
        return false;
    }

    // 대기 소켓의 로컬 주소, 포트 설정
    SOCKADDR_IN serveraddr;
    ZeroMemory(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(40100);
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    retval = bind(listenSock, (SOCKADDR *)&serveraddr, sizeof(serveraddr));
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }

    // 대기 소켓의 접속 대기 큐 생성 및 클라이언트 접속 대기
    retval = listen(listenSock, SOMAXCONN);
    if(retval == SOCKET_ERROR)
    {
        DisplayMessage();
        return false;
    }

    return true;
}

int main(int argc, char* argv[])
{
    WSADATA wsa;

    if(WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("윈도우 소켓 초기화 실패!\n");
        return -1;
    }

    // 입출력 완료 포트 생성
    HANDLE hcp = CreateIoCompletionPort(

```

```

        INVALID_HANDLE_VALUE, NULL, 0, 0);
if(hcp == NULL) return -1;

// CPU 개수 확인
SYSTEM_INFO si;
GetSystemInfo(&si);

// 작업자 스레드 생성
HANDLE hThread;
unsigned int ThreadId;
for(int i=0; i<(int)si.dwNumberOfProcessors*2; i++)
{
    hThread = (HANDLE) _beginthreadex(NULL, 0, workerThread, (void*) hcp, 0,
                                     &ThreadId);
    if(hThread == NULL) return -1;
    CloseHandle(hThread);
}

// 대기 소켓 초기화(socket()+bind()+listen())
if( !CreateListenSocket( ) )
{
    printf("대기 소켓 생성 실패!\n");
    return -1;
}

while(1){
    int retval;
    SOCKADDR_IN clientaddr;
    int addrlen = sizeof(clientaddr);
    SOCKET client_sock = accept(listenSock, (SOCKADDR *)&clientaddr,
                                &addrlen);
    if(client_sock == INVALID_SOCKET){
        DisplayMessage();
        continue;
    }
    printf("[TCP 서버] 클라이언트 접속: IP 주소=%s, 포트 번호=%d\n",
           inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port));

    // 소켓과 입출력 완료 포트 연결
    HANDLE hResult = CreateIoCompletionPort((HANDLE)client_sock, hcp,
                                             (DWORD)client_sock, 0);
    if(hResult == NULL) return -1;

    // 소켓 정보 구조체 할당
    SOCKETINFO *ptr = new SOCKETINFO;
    if(ptr == NULL)
    {
        printf("[오류] 메모리가 부족합니다!\n");
        break;
    }
    ZeroMemory(&(ptr->overlapped), sizeof(ptr->overlapped));
    ptr->sock = client_sock;
    ptr->recvbytes = 0;
    ptr->sendbytes = 0;
    ptr->wsabuf.buf = ptr->buf;
    ptr->wsabuf.len = BUFSIZE;

    // 비동기 입출력 시작
    DWORD recvbytes;
    DWORD flags = 0;
    retval = WSAREcv(client_sock, &(ptr->wsabuf), 1, &recvbytes,
                     &flags, &(ptr->overlapped), NULL);
    if(retval == SOCKET_ERROR)
    {
        if(WSAGetLastError() != ERROR_IO_PENDING){

```

```

        DisplayMessage();
    }
    continue;
}

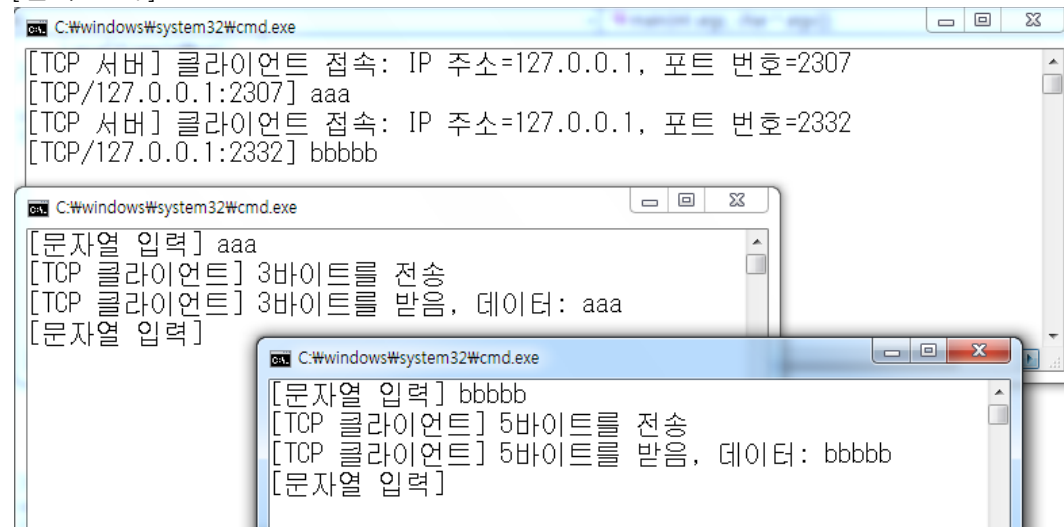
// 원속 종료
WSACleanup();
return 0;
}

```

</코드>

<결과>

[출력 결과]



</결과>

다음 코드는

```

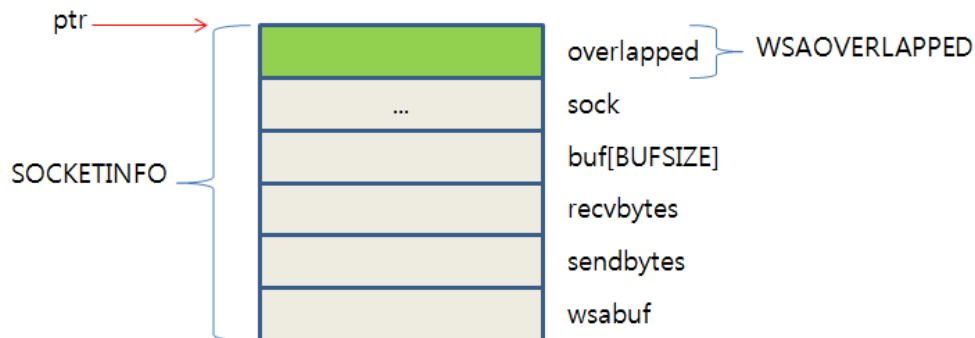
struct SOCKETINFO
{
    WSAOVERLAPPED overlapped;
    SOCKET sock;
    char buf[BUFSIZE];
    int recvbytes;
    int sendbytes;
    WSABUF wsabuf;
};

```

는 소켓 정보를 저장하기 위한 구조체로 첫 번째 멤버가 WSAOVERLAPPED 구조체로 완료 루틴에서 사용한 방법처럼 비동기 입출력 함수에 사용되는 인수의 WSAOVERLAPPED 주소를 형식변환하여 구조체 정보를 얻는데 사용된다.

다음은 비동기 입출력 함수의 인수로 사용되는 WSAOVERLAPPED 구조체 주소를 WorkerThread 의 GetQueuedCompletionStatus() 함수의 네 번째 인수로 받아 소켓 정보 구조체로 형식 변환하여 사용하는 그림을 표현한 것이다.

```
SOCKETINFO *ptr;
retval = GetQueuedCompletionStatus(hcp, &cbTransferred,
                                   (LPDWORD)&client_sock, (LPOVERLAPPED *)&ptr, INFINITE);
```



다음 코드는

```
unsigned int WINAPI workerThread(void* pParam)
{
    while(1){
        // 비동기 입출력 완료 기다리기
        DWORD cbTransferred;
        SOCKET client_sock;
        SOCKETINFO *ptr;
        retval = GetQueuedCompletionStatus(hcp, &cbTransferred,
                                           (LPDWORD)&client_sock, (LPOVERLAPPED *)&ptr, INFINITE);

        // 클라이언트 정보 얻기
        SOCKADDR_IN clientaddr;
        int addrlen = sizeof(clientaddr);
        getpeername(ptr->sock, (SOCKADDR *)&clientaddr, &addrlen);

        // 비동기 입출력 결과 확인
        if(retval == 0 || cbTransferred == 0){
            ...
        }
    }
}
```

IOCP 의 완료 정보를 확인하는 WorkerThread 로 이 스레드의 핵심 기능은 GetQueuedCompletionStatus() 함수를 사용하여 IOCP 의 완료 정보를 확인하여 완료 정보가 없으면 스레드가 대기(블로킹) 상태에 놓이게 된다.

```
retval = GetQueuedCompletionStatus(hcp, &cbTransferred,
                                   (LPDWORD)&client_sock, (LPOVERLAPPED *)&ptr, INFINITE);
```

의 hcp 는 IOCP 의 핸들이며 cbTransferred 는 입출력 완료한 바이트 수이고 client_sock 은 IOCP 입출력 작업을 구분하는 key 값으로 소켓과 IOCP 를 연결할 때 소켓 핸들을 지정했으므로 입출력을 완료한 소켓 핸들이 out parameter 로 반환된다. ptr 은 WSAOVERLAPPED 의 주소로 소켓 정보 구조체로 형식 변환되어 사용된다. INFINITE 는 입출력 작업이 완료할 때까지 스레드가 무한정 대기한다.

retval == 0 || cbTransferred == 0 부분은 오류나 소켓의 정상 종료이므로 종료 처리를 하는 코드다.

다음 코드는

```
if(ptr->recvbytes == 0){
    ...
}
else{
    ptr->sendbytes += cbTransferred;
}

if(ptr->recvbytes > ptr->sendbytes){
    // 데이터 보내기
    retval = WSASend(ptr->sock, &(ptr->wsabuf), 1,
        &sendbytes, 0, &(ptr->overlapped), NULL);
    ...
}
else{
    ptr->recvbytes = 0;

    // 데이터 받기
    retval = WSAREcv(ptr->sock, &(ptr->wsabuf), 1,
        &recvbytes, &flags, &(ptr->overlapped), NULL);
    ...
}
}
```

데이터의 송, 수신이 완료되었을 때 에코 작성을 처리하는 부분으로 완료 루틴과 크게 다르지 않다. 비동기 입출력 함수 마지막 인수로 완료 루틴 콜백 함수를 지정하지 않는다.

다음 코드는

```
HANDLE hcp = CreateIoCompletionPort(
    INVALID_HANDLE_VALUE, NULL, 0, 0);
if(hcp == NULL) return -1;
```

IOCP 객체를 생성하는 부분으로 성공하면 IOCP 핸들(hcp)을 반환한다. IOCP 객체를 생성하기 위해서는 첫 인수로 INVALID_HANDLE_VALUE 를 지정한다.

다음 코드는

```
SYSTEM_INFO si;
GetSystemInfo(&si);

// 작업자 스레드 생성
HANDLE hThread;
unsigned int ThreadId;
for(int i=0; i<(int)si.dwNumberOfProcessors*2; i++){
    hThread = (HANDLE) _beginthreadex(NULL, 0, workerThread, (void*) hcp, 0,
        &ThreadId);
    if(hThread == NULL) return -1;
    CloseHandle(hThread);
}
```

시스템 정보를 읽어와 CPU(코어) 개수의 두 배만큼 완료 작업을 처리하는 WorkerThread 를 생성하는 부분으로 완료 작업을 처리하는 스레드가 CPU 의 두 배일때 IOCP 의 성능이 뛰어나다고 알려져 있다. IOCP 는 완료 루틴과 다르게 IOCP 의 핸들만 알고 있으면 어떤 스레드든지 IOCP 의 입출력 완료 정보를 처리할 수 있다.

다음 코드는

```

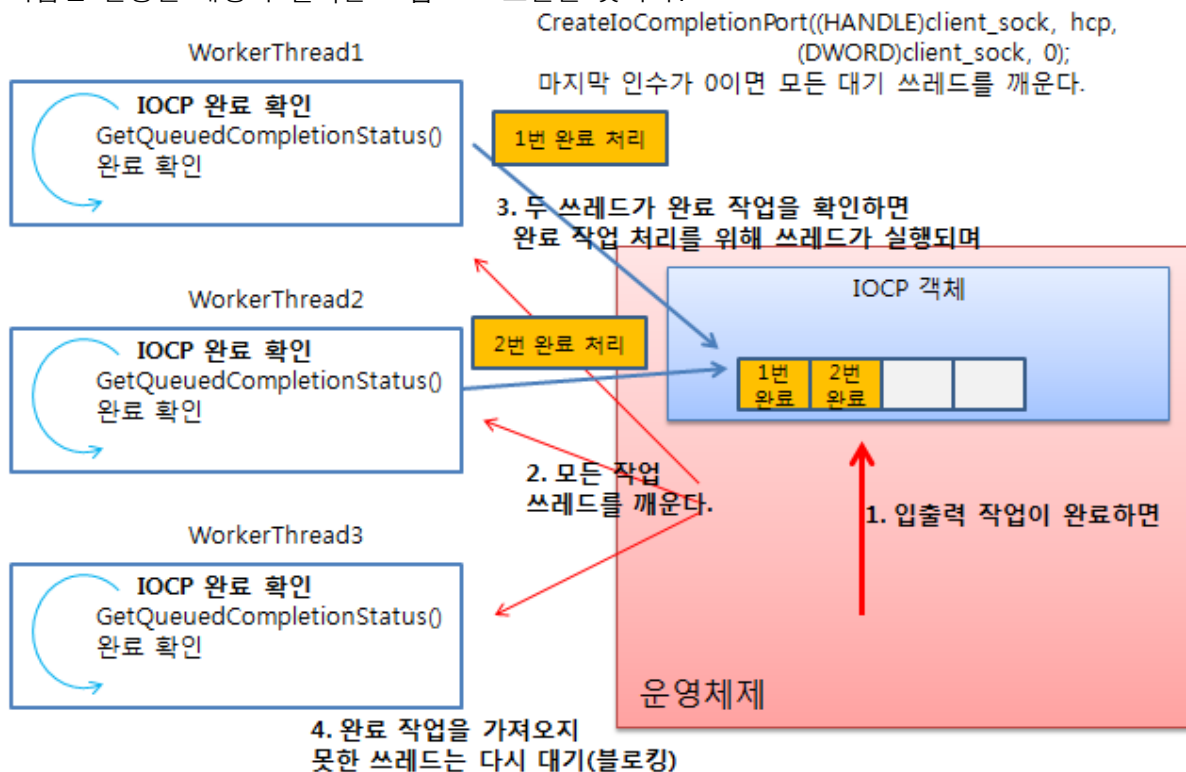
SOCKET client_sock = accept(listenSock, (SOCKADDR *)&clientaddr,
    &addrLen);
if(client_sock == INVALID_SOCKET){
    DisplayMessage();
    continue;
}
printf("[TCP 서버] 클라이언트 접속: IP 주소=%s, 포트 번호=%d\n",
    inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port));

// 소켓과 입출력 완료 포트 연결
HANDLE hResult = CreateIoCompletionPort((HANDLE)client_sock, hcp,
    (DWORD)client_sock, 0);
if(hResult == NULL) return -1;

```

통신 소켓이 생성되면 생성된 통신 소켓과 IOCP 객체를 연결하는 코드로 client_sock 은 hcp 와 연결되고 client_sock 이 입출력 정보를 구분하는 Key 로 설정되며 CreateIoCompletionPort()의 마지막 인수가 0 이므로 IOCP 의 완료 정보를 대기하는 모든 스레드를 대기(블로킹) 상태에서 깨워 실행 시킨다. 깨어난 스레드는 GetQueuedCompletionStatus() 함수로 완료 작업을 확인하고 함수가 반환된 스레드는 완료 작업 수행하기 위해 실행되지만 반환되지 않은 스레드는 다시 대기 상태에 놓인다.

다음은 설명한 내용의 절차를 그림으로 표현한 것이다.



다음 코드는

```

SOCKETINFO *ptr = new SOCKETINFO;
if(ptr == NULL){
    printf("[오류] 메모리가 부족합니다!\n");
    break;
}
...

```

```

// 비동기 입출력 시작
DWORD recvbytes;
DWORD flags = 0;
retval = WSAREcv(client_sock, &(ptr->wsabuf), 1, &recvbytes,
    &flags, &(ptr->overlapped), NULL);
if(retval == SOCKET_ERROR)
{
    if(WSAGetLastError() != ERROR_IO_PENDING)
    {
        DisplayMessage();
    }
    continue;
}

```

통신 소켓과 IOCP 를 연결한 후 소켓 정보를 메모리에 생성하고 데이터 수신을 위한 비동기 입출력을 시작한다. (비동기 입출력 이 시작되면 입출력이 완료됐을 때 IOCP 에 완료 정보가 저장된다.)

클라이언트는 지금까지 실행한 스레드 사용 클라이언트 애플리케이션과 동일하다.