다중 접속 IO 모델

MM4220 게임서버 프로그래밍

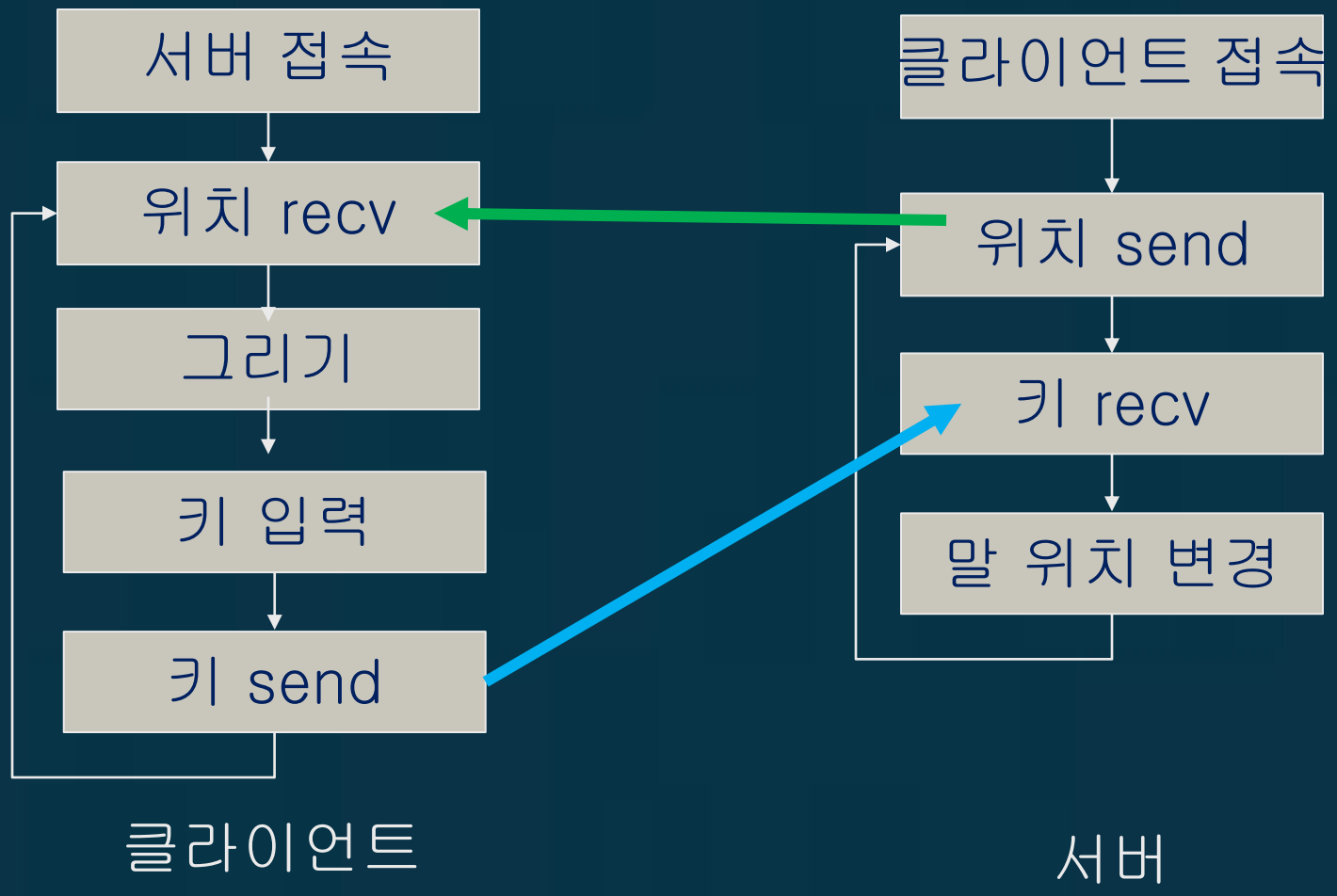
정내훈

1 : 1 접속

- 과제 #2

1 : 1 접속

- 지난 주 과제

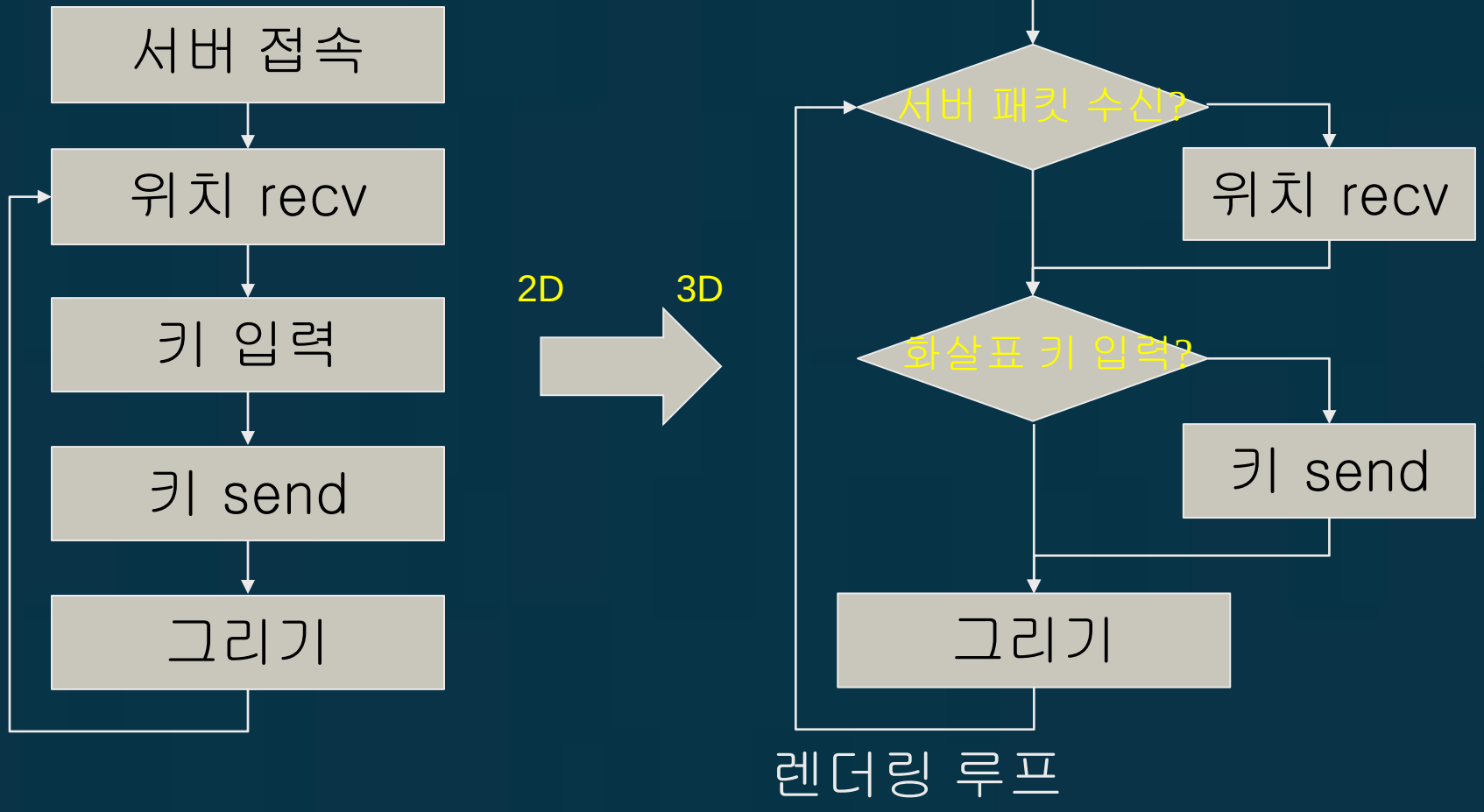

서버 접속
위치 recv
그리기
키 입력
키 send
클라이언트
클라이언트 접속
위치 send
키 recv
말 위치 변경
서버

1 : 1 접속

- 게임프로그램의 특징
  - 게임의 진행 속도는 CPU의 속도가 아니라 실제 시간이다.
  - 내가 명령을 입력할 때까지 기다리지 않는다.
    - 적어도 렌더링은 계속 된다.
  - 모든 객체가 독립적으로 행동한다.
    - 게임의 진행이 미리 정해져 있지 않다.

1 : 1 접속

- 클라이언트는 3D



1 : 1 접속

- 데이터 수신 확인 방법
  - 소켓의 모드를 Non-Blocking모드로 바꾸고 WSARecv를 한다.
    - 데이터가 도착하지 않았을 경우에도 그대로 리턴한다.
- Non-blocking I/O

```
unsigned long noblock = 1;
int nRet = ioctlsocket(sock, FIONBIO, &noblock);
```

  - Socket의 모드를 blocking에서 non-blocking으로 변환
  - WSARecv() 호출을 즉시 완료할 수 없을 때.
    - 즉 도착한 데이터가 없을 때
    - WSAEWOULDBLOCK 에러를 내고 끝난다.
    - 기다리지 않는다

다중 접속 서버

- 그러면 n : 1 온라인 게임의 구현은?
  - 클라이언트는 달라지는 것이 거의 없다.
    - 서버가 지시하는 여러 개의 객체를 동시에 그려야 한다.
  - 서버가 문제이다.
    - 성능 문제
    - 많은 클라이언트로 인한 잦은 WSARecv 실패
  - 효율적인 네트워크 API 호출을 위한 서버용 네트워크 I/O 모델 필요.
    - recv 뿐만 아니라 accept도 고려해야 한다.
    - 멀티쓰레드 고려 필요

다중 접속 서버

- 서버 메인 루프 (Non-Blocking I/O)

```
while (true) {
    // Process Client Packet
    ret = recv(client_A, buf);
    if (SUCCESS == ret) process_packet(client_A, buf);
    ret = recv(client_B, buf);
    if (SUCCESS == ret) process_packet(client_B, buf);

    // Update World
    // do NPC AI, do Heal, …
    …
}
```

  - 무엇이 문제인가?
    - 실패한 Recv == CPU 낭비

네트워크 I/O 모델

- 게임 서버에서의 다중 접속
  - 정해지지 않은 동작 순서
    - 그래도 멈추지 않아야 하는 게임
  - 수 천 개의 접속
  - 상대적으로 낮은 접속 당 bandwidth
    - 효율적인 API 호출 필요
- 네트워크 I/O 모델이 필요한 이유
  - 블럭킹 방지
  - 비 규칙적인 입출력의 효율적 관리
  - 대규모 다중 접속 관리

Windows I/O 모델(2023)

- Non-blocking I/O
- Socket Thread
- Select
- WSAAsyncSelect
- WSAEventSelect
- Overlapped I/O (Event)
- Overlapped I/O (Callback)
- I/O Completion Port

Windows I/O 모델

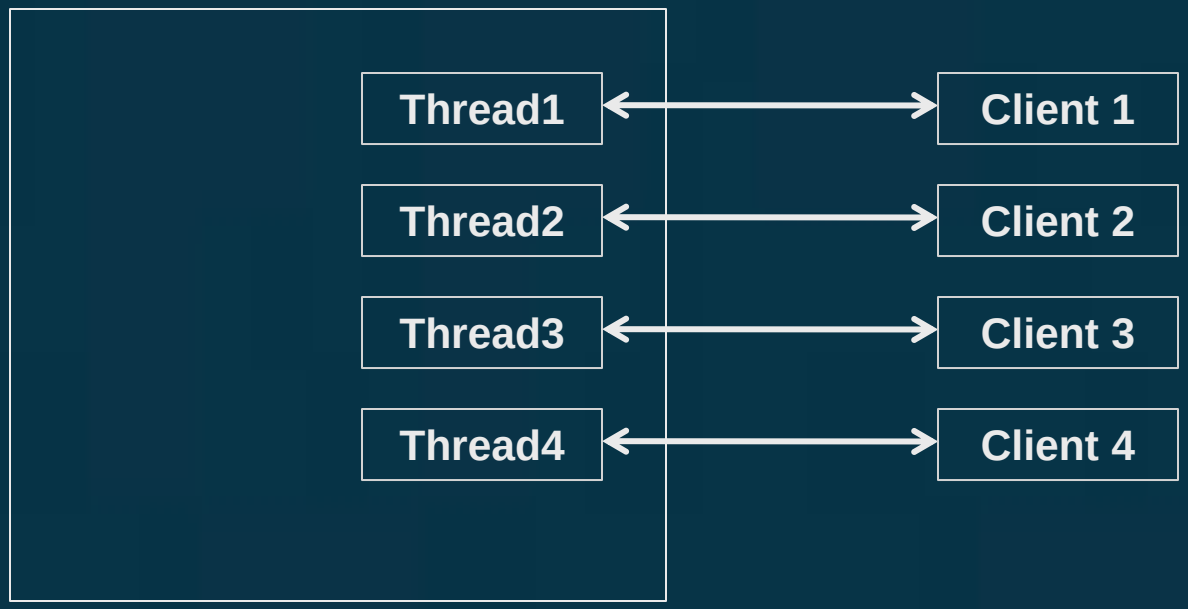
- Non-blocking I/O
  - 단점 : Busy Waiting
    - 모든 소켓의 recv를 돌아가면서 반복 호출해야 함.
    - 많은 실패한 recv
      - 잦은 시스템 Call -> CPU낭비 -> 성능 저하

서버 메인 루프

```
while (true) {
    for (SOCKET s : sockets) {
        recv(s,…);
        if (success) 패킷 처리;
    }
    다른 작업;
}
```

Windows I/O 모델

- Socket Thread
  - Thread를 통한 처리
    - 1:1 게임서버가 여러 개 있는 것과 비슷하다.



Windows I/O 모델

- Thread를 통한 처리

```
while (!shutdown) {
    new_sock = accept(sock, &addr, &len);
    thread t = thread { do_io, new_sock };
}
```

```
void do_io(mysock) {
    while (true) {
      recv(mysock);
      process_packet(); }}
```

  - 다중 소켓 처리 가능
  - non-blocking 불필요 => 효율적인 API 호출
  - 과도한 thread 개수로 인한 운영체제 Overhead
    - thread당 오버헤드 : 컨텍스트 스위치, Thread Control Block, Stack 메모리

Windows I/O 모델

- Coroutine을 사용한 처리
  - 프로그래밍 방법은 thread-IO와 동일
  - Coroutine API에서 오버헤드 전부 해결
    - Thread Pool을 사용하여 적은 개수의 Thread로 많은 개수의 coroutine 실행
- 문제
  - C++20에 coroutine이 들어갔는데, 아직 초기단계
  - C++23에 제대로 구현예정
- 지금 사용하고 싶다면
  - GO 언어나 Erlang계열의 언어를 사용

Windows I/O 모델

- Select
  - Unix시절부터 내려온 고전적인 I/O 모델
  - 여러 개의 소켓의 데이터 도착 여부를 하나의 API로 검사
    - 효율적인 API 호출
  - 단점
    - socket 개수의 한계 존재
      - unix: 64, linux: 1024
    - socket의 개수가 많아질수록 성능 저하 증가
      - linear search

Windows I/O 모델

- Select

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

  - nfds : 검사하는 소켓의 개수, Windows에서는 무시
  - readfds : 읽기 가능 검사용 소켓 집합 포인터
  - writefds : 쓰기 가능 검사용 소켓 집합 포인터
  - exceptfds : 에러 검사용 소켓 집합 포인터
  - timeout : select가 기다리는 최대 시간
  - return value : 사용 가능한 소켓의 개수

Windows I/O 모델

- Select
  - 실제 코딩

```
FD_SET(sock1, &rfds);
FD_SET(sock2, &rfds);

select(0, &rfds, NULL, NULL, &time);

if (FD_ISSET(sock1, &rfds)) recv(sock1, buf, len, 0)
if (FD_ISSET(sock2, &rfds)) recv(sock2, buf, len, 0)
```

  - 서버 구조

```
while (true) {
    select(&all_sockets);
    for (i : all_sockets)
        if (IS_READY(i)) {
            recv(socket[i], buf, …);
            process_packet(i, buf);
        }
    다른 작업;
}
```

Windows I/O 모델

- WSAAsyncSelect
  - 소켓 이벤트를 특정 윈도우의 메시지로 받는다

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

  - s : 소켓
  - hWnd : 메시지를 받을 윈도우
  - wMsg : 메시지 번호
  - lEvent : 반응 event 선택

Windows I/O 모델

- WSAAsyncSelect
  - 클라이언트에 많이 쓰임
    - 윈도우 필요
    - 윈도우 메시지 큐를 사용 -> 성능 느림

lEvent

비트 값	의미
FD_READ	Recv 할 데이터가 있음
FD_WRITE	Send할 수 있는 버퍼 공간 있음
FD_OOB	Out-of-band 데이터 있음
FD_ACCEPT	Accept 준비가 됨
FD_CONNECT	접속이 완료됨
FD_CLOSE	소켓연결이 종료됨

Windows I/O 모델

- WSAAsyncSelect
  - 자동으로 non-blocking mode로 소켓 전환
  - 아래와 같은 함수로 이벤트 처리

```
LRESULT CAsyncSelectDlg::OnSocketMsg(WPARAM wParam, LPARAM lParam)
{
        SOCKET sock=(SOCKET)wParam;
        int nEvent = WSAGETSELECTEVENT(lParam);
        switch(nEvent){
                case FD_READ :
                case FD_ACCEPT :
                case FD_CLOSE :
```

Windows I/O 모델

- WSAEventSelect

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

  - s : 소켓
  - hEventObject : 데이터 처리 필요 알리미.
  - lNetworkEvents : WSAAsyncSelect와 같음
  - 메시지를 처리할 윈도우가 필요 없음.

Windows I/O 모델

- WSAEVENT란?
  - 운영체제가 관리하는 플랙.
  - 세마포어나 컨디션변수와 비슷
  - 운영체제가 응용프로그램에게 어떠한 EVENT가 발생했음을 알리는 용도로 사용,
  - 응용프로그램은 EVENT가 true가 될 때까지 실행을 멈출 수 있음 (CPU낭비 없이)
    - 이때 여러 개의 이벤트를 하나의 명령으로 대기할 수 있음.

Windows I/O 모델

- WSAEventSelect
  - socket과 event의 array를 만들어서 `WSAWaitForMultipleEvents()` 의 리턴값으로 부터 socket 추출
  - 소켓의 개수 64개 제한!
    - 멀티 쓰레드를 사용해서 제한 극복가능

Windows I/O 모델

- WSAEventSelect
  - 다음의 API로 socket대기 상태 검출

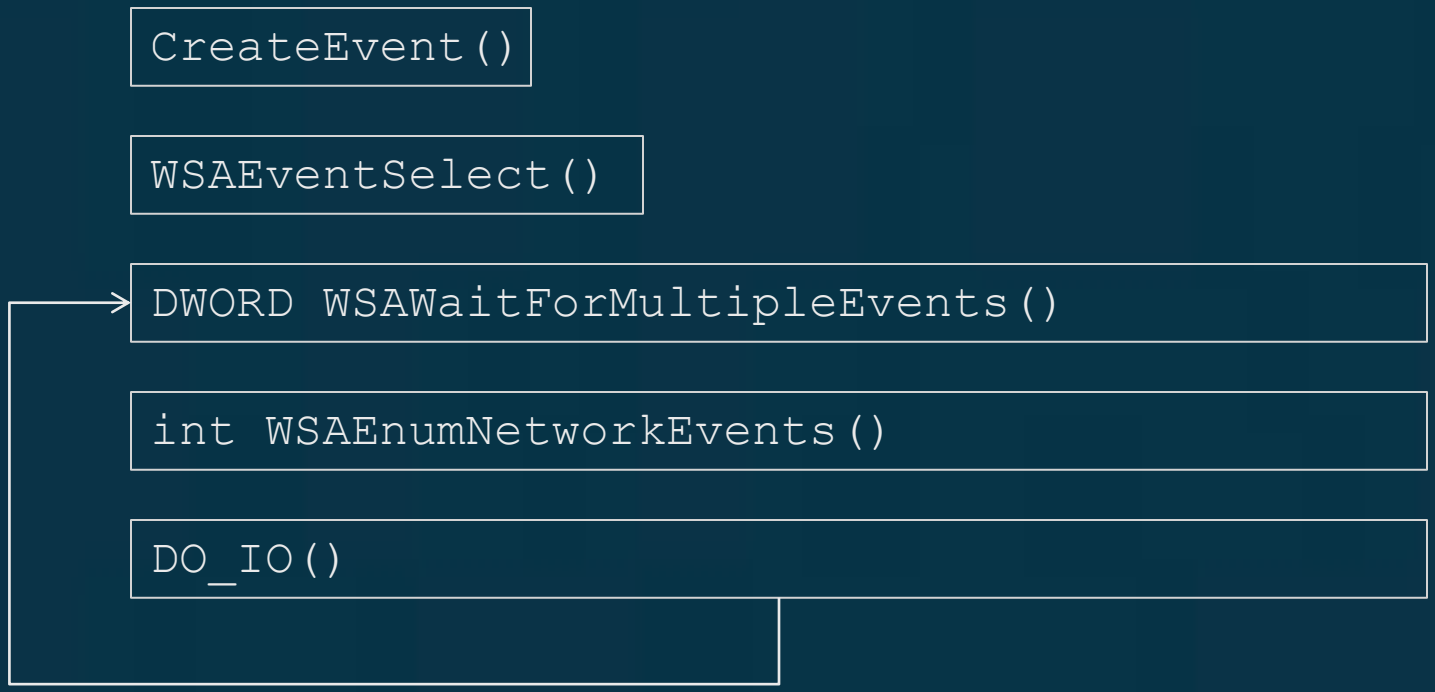
```
DWORD WSAWaitForMultipleEvents(
        DWORD nCount,
        const WSAEVENT *lphEvents,
        BOOL bWaitAll,
        DWORD dwMiliseconds);
```

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

```
lpNetworkEvents->lNetworkEvents;     // FD_READ, FD_WRITE,
                                     // FD_ACCEPT, ….
```

Windows I/O 모델

- WSAEventSelect : 동작



Windows I/O 모델

- WSAEventSelect : 예제
  - http://perfectchoi.blogspot.com/2009/09/wsaeventselect.html

Windows I/O 모델

- Overlapped I/O 모델
  - Windows에서 추가된 고성능 I/O 모델
  - 다른 이름으로는 Asynchronous I/O 또는 비동기 I/O
    - 리눅스의 경우 boost/asio 라이브러리로 사용 가능
  - 대용량 고성능 네트워크 서버를 위해서는 필수 기능
  - IOCP도 Overlapped I/O를 사용
  - 사용 방법이 select style의 I/O 모델과 다르다.
    - I/O요청을 먼저하고 I/O의 종료를 나중에 확인한다.
      - 요청 후 즉시 리턴
      - Non-Blocking과는 다르게 거의 실패하지 않는다.
    - I/O요청 후 기다리지 않고 다른 일을 할 수 있다.
    - 여러 개의 I/O요청을 동시에 할 수 있다.

Windows I/O 모델

- Overlapped I/O 모델

```
while (true) {
    select(&all_sockets);
    for (i : all_sockets)
        if (IS_READY(i)) {
            recv(socket[i], buf, …);
            process_packet(i, buf);
        }
}
```

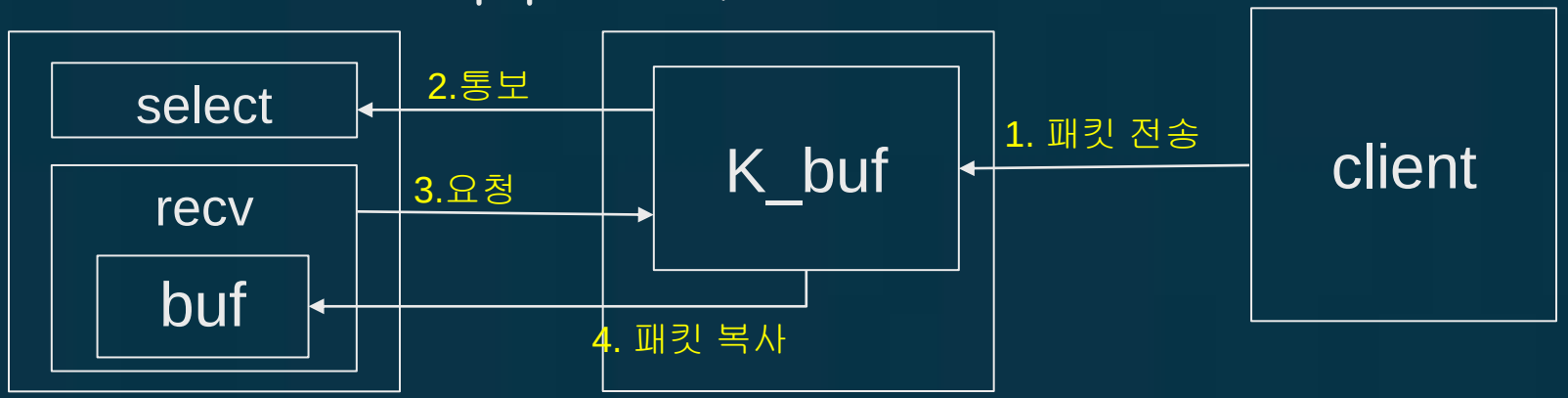
non-overlapped I/O
aka. select 모델

```
for (sock : all_sockets)
    recv(sock, buf[sock]);
while (true) {
    sock = wait_for_completed_recv();
    process_packet(sock, buf[sock]);
    recv(sock, buf[sock], …);
}
```

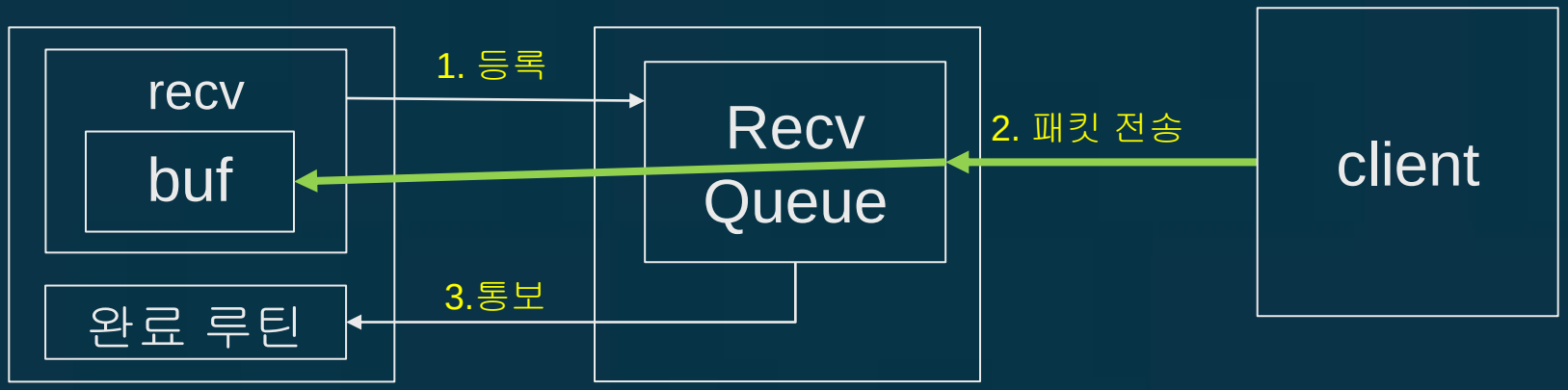
overlapped I/O

Windows I/O 모델

- Non-Overlapped I/O 모델



- Overlapped I/o



Windows I/O 모델

- Overapped I/O 모델
  - 소켓 내부 버퍼를 사용하지 않고 직접 사용자 버퍼에서 데이터를 보내고 받을 수 있다. (옵션)

```
int result;
int buffsize = 0;
result = setsockopt(s, SOL_SOCKET, SO_SNDBUF, &buffsize, sizeof(buffsize));
```

  - Select I/O 모델들은 recv와 send의 가능 여부만 검사 후 I/O수행, Overlapped는 I/O 요청 후 실행 완료 통보
- Non-Blocking IO와의 차이
  - I/O 호출과 완료의 분리
    - Non-Blocking IO는 실패와 성공이 있고 거기서 끝, 뒤끝이 없음
    - Overlapped IO는 요청이 커널에 남아서 수행됨. 버퍼가 커널에 등록됨, 실패 없음
  - I/O 다중 수행, Network Data Copy 감소

Windows I/O 모델

- Overapped I/O 모델
  - Send와 Recv를 호출했을 때 패킷 송수신의 완료를 기다리지 않고 Send, Recv함수 종료
  - 이때 Send와 Recv는 송수신의 시작을 지시 만하는 함수
    - 이미 도착한 데이터가 있으면 받을 수도 있지만, 그렇게 하지 않을 것임.
  - 여러 개의 Recv, Send를 겹쳐서 실행함으로써 여러 소켓에 대한 동시 다발적 Recv, Send도 가능
    - 하나의 socket은 하나의 recv만 가능!!!

Windows I/O 모델

- 운영체제의 입장
  - 동기식 : 네트워크에서 데이터가 왔을 때 따로 버퍼에 저장해 두고 Recv요청이 올 때 까지 기다린다. 요청이 오면 복사해서 보낸다.
  - 비동기식 : 네트워크에서 데이터가 왔을 때 저장된 Recv요청이 있다면 요청된 버퍼에 데이터를 저장하고 즉시 완료 시킨다.

Windows I/O 모델

- Overlapped I/O
  - SOCKET WSASocket(int af, int type, int protocol, LPWSAPROTOCOL_INFO lpProtocolInfo, GROUP g, DWORD dwFlags)
    - af : address family
      - AF_INET만 사용 (AF_NETBIOS, AF_IRDA, AF_INET6)
    - type : 소켓의 타입
      - tcp를 위해 SOCK_STREAM사용 (SOCK_DGRAM)
    - protocol : 사용할 프로토콜 종류
      - IPPROTO_TCP (IPPROTO_UDP)
    - lpProtocolInfo : 프로토콜 정보
      - 보통 NULL
    - g : 예약
    - dwFlags : WSA_FLAG_OVERLAPPED

Windows I/O 모델

- Overlapped I/O
  - int WSARecv(SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount, LPDWORD lpNumberofBytesRecvd, LPDWORD lpFlags, LPWSAOVERLAPPED lpOverlapped, LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine)
    - s : 소켓
    - lpBuffers : 받은 데이터를 저장할 버퍼
    - dwBufferCount : 버퍼의 개수
    - lpFlags : 동작 옵션(MSG_PEEK, MSG_OOB)
    - lpNumberofBytesRecvd : 받은 데이터의 크기 => NULL
    - lpOverlapped : NULL이 아닌 경우 overlapped I/O로 동작
    - lpCompletionRoutine : NULL이 아닌 경우 overlapped I/O가 callback 모드로 동작

Windows I/O 모델

- Overlapped I/O
  - LPWSAOVERLAPPED lpOverlapped

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

    - Internal, InternalHigh, Offset, OffsetHigh : 0으로 초기화 후 사용
    - hEvent I/O가 완료 되었음을 알려주는 event 핸들
  - LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
    - callback 함수, 뒤에 설명

Windows I/O 모델

- Overlapped I/O 모델
  - Overlapped I/O가 언제 종료되었는지를 프로그램이 알아야 함
  - 두 가지 방법이 존재
    - Overlapped I/O Event모델
    - Overlapped I/O Callback모델

Windows I/O 모델

- Overlapped I/O Event 모델
  - WSARecv의 LPWSAOVERLAPPED lpOverlapped 구조체의 WSAEVENT hEvent 사용
  - 작업 결과 확인
    - `WSAWaitForMultipleEvents()` : 완료 확인
    - `WSAGetOverlappedResult()` : 정보 얻기

Windows I/O 모델

- Overlapped I/O Event 모델
  - WSAGetOverlappedResult()

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

  - s :socket
  - lpOverlapped : WSARecv에 넣었던 구조체
  - lpcbTransfer : 전송된 데이터 크기
  - fWait : 대기 여부
  - lpdwFlags : Recv의 lpFlag의 결과

Windows I/O 모델

- Overlapped I/O Event 모델
  1. WSACreateEvent()를 사용해서 이벤트 생성
  2. WSAOVERLAPPED구조체 변수선언, 0으로 초기화 hEvent에 1의 이벤트
  3. WSASend(), WSARecv()
     - 2의 구조체를 WSAOVERLAPPED에
       - 중복 사용 불가능!! 호출 완료 후 재사용
     - lpCompletionROUTINE에 NULL
  4. WSAWaitForMultipleEvents()함수로 완료 이벤트 감지
  5. WSAGetOverlappedResult()함수로 정보 획득

Windows I/O 모델

- Overlapped I/O Event 모델
  - 단점
    - 시스템 호출이 빈번하다.
    - 최대 동접 64이다.
  - 대안
    - Overlapped I/O Callback 사용

Windows I/O 모델

- Overlapped I/O Callback 모델
  - 이벤트 제한 개수 없음
  - 사용하기 편리
  - WSARecv와 WSASend의 LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine 함수 사용
  - Overlapped I/O가 끝난후 lpCompletionRoutine이 호출됨

Windows I/O 모델

- Overlapped I/O Callback 모델
  - Callback함수

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

  - dwError : 작업의 성공 유무
  - cbTransferred : 전송된 바이트 수
  - lpOverlapped : WSASend/WSARecv에서 사용한 구조체
  - dwflags : WSASend/WSARecv에서 사용한 flag

Windows I/O 모델

- Overlapped I/O Callback 모델
  - 누가/언제 Callback을 호출하는가?
  - 멀티쓰레드로 동시에 Callback이 실행되는가? NO
  - 운영체제가 실행중인 프로그램을 강제로 멈추고 Callback을 실행하는가? NO
    - Linux의 signal은 이렇게 동작.
      - 문제가 많다. signal handler의 작성이 까다롭다.
  - 프로그램이 운영체제를 호출하고 대기중일 때, 운영체제에서 실행을 요청한다.
    - 운영체제를 호출하는 프로그램에서 호출에 대한 응답인지 Callback실행 요청인지를 판단해야 한다.
    - 모든 운영체제 호출이 Callback을 검사하지 않는다.
      - 해당 운영체제 호출 : **SleepEx**, **SignalObjectAndWait**, **MsgWaitForMultipleObjectsEx**, **WaitForMultipleObjectsEx**,
  - Callback을 사용한 프로그램의 구조
    - 할 일이 있으면 하고, 없으면 SleepEx를 호출한다.

Windows I/O 모델

- Overlapped I/O
  - Overlapped I/O 구분

lpOverlapped	hEvent	lpCompletionRoutine	Completion 여부 식별
NULL	세팅 불가	무시됨	동기적 실행
Not-NULL	NULL	NULL	Overlapped 동작, 완료 검사 불가능
Not-NULL	Not-NULL	NULL	Overlapped 동작, Event객체로 완료 검사
Not-NULL	무시됨	Not-NULL	Overlapped 동작, completion routine을 통해서 완료 관리

실습 순서

- 1 : 1 Overlapped Echo Client
- 1 : 1 Overlapped Echo Server
- n : 1 Overlapped Echo Server
- n : 1 Overlapped Chatting Server

실습 : Overlapped_Client (1/2)

```
#include <iostream>
#include <WS2tcpip.h>
using namespace std;
#pragma comment(lib, "WS2_32.LIB")

constexpr short SERVER_PORT = 3500;
constexpr int BUF_SIZE = 200;
WSAOVERLAPPED s_over;
SOCKET s_socket;
WSABUF s_wsabuf[1];
char   s_buf[BUF_SIZE];

int main()
{
   WSADATA WSAData;
   WSAStartup(MAKEWORD(2, 2), &WSAData);
   s_socket = WSASocket(AF_INET, SOCK_STREAM, 0, 0, 0, WSA_FLAG_OVERLAPPED);
   SOCKADDR_IN svr_addr;
   memset(&svr_addr, 0, sizeof(svr_addr));
   svr_addr.sin_family = AF_INET;
   svr_addr.sin_port = htons(SERVER_PORT);
   inet_pton(AF_INET, "127.0.0.1", &svr_addr.sin_addr);
   WSAConnect(s_socket, reinterpret_cast<sockaddr*>(&svr_addr), sizeof(svr_addr), 0, 0, 0, 0);
   do_send_message();
   while (true) SleepEx(100, true);
   closesocket(s_socket);
   WSACleanup();
}
```

실습 : Overlapped_Client (2/2)

```
void CALLBACK send_callback(DWORD err, DWORD num_bytes, LPWSAOVERLAPPED over, DWORD flags);
void do_send_message()
{
   cout << "Enter Messsage: ";
   cin.getline(s_buf, BUF_SIZE - 1);
   s_wsabuf[0].buf = s_buf;
   s_wsabuf[0].len = static_cast<int>(strlen(s_buf)) + 1;
   memset(&s_over, 0, sizeof(s_over));
   WSASend(s_socket, s_wsabuf, 1, 0, 0, &s_over, send_callback);
}

void CALLBACK recv_callback(DWORD err, DWORD num_bytes, LPWSAOVERLAPPED over, DWORD flags)
{
   cout << "Server Sent: " << s_buf << endl;
   do_send_message();
}

void CALLBACK send_callback(DWORD err, DWORD num_bytes, LPWSAOVERLAPPED over, DWORD flags)
{
   s_wsabuf[0].len = BUF_SIZE;
   DWORD r_flag = 0;
   memset(over, 0, sizeof(*over));
   WSARecv(s_socket, s_wsabuf, 1, 0, &r_flag, over, recv_callback);
}
```

실습 : Overlapped_Server (1/2)

```
#include <iostream>
#include <WS2tcpip.h>
#pragma comment(lib, "WS2_32.lib")
using namespace std;
constexpr int PORT_NUM = 3500;
constexpr int BUF_SIZE = 200;
SOCKET client;
WSAOVERLAPPED c_over;
WSABUF c_wsabuf[1];
CHAR c_mess[BUF_SIZE];

int main()
{
   WSADATA WSAData;
   WSAStartup(MAKEWORD(2, 2), &WSAData);
   SOCKET server = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
   SOCKADDR_IN server_addr;
   memset(&server_addr, 0, sizeof(server_addr));
   server_addr.sin_family = AF_INET;
   server_addr.sin_port = htons(PORT_NUM);
   server_addr.sin_addr.S_un.S_addr = INADDR_ANY;
   bind(server, reinterpret_cast<sockaddr*>(&server_addr), sizeof(server_addr));
   listen(server, SOMAXCONN);
   SOCKADDR_IN cl_addr;
   int addr_size = sizeof(cl_addr);
   client = WSAAccept(server, reinterpret_cast<sockaddr *>( & cl_addr), &addr_size, NULL, NULL);

   do_recv();
   while (true) SleepEx(100, true);
   closesocket(server);
   WSACleanup();
}
```

실습 : Overlapped_Server (2/2)

```
void CALLBACK recv_callback(DWORD err, DWORD num_bytes, LPWSAOVERLAPPED over, DWORD flags);
void do_recv()
{
   c_wsabuf[0].buf = c_mess;
   c_wsabuf[0].len = BUF_SIZE;
   DWORD recv_flag = 0;
   memset(&c_over, 0, sizeof(c_over));
   WSARecv(client, c_wsabuf, 1, 0, &recv_flag, &c_over, recv_callback);
}

void CALLBACK send_callback(DWORD err, DWORD num_bytes, LPWSAOVERLAPPED over, DWORD flags)
{
   do_recv();
}

void CALLBACK recv_callback(DWORD err, DWORD num_bytes, LPWSAOVERLAPPED over, DWORD flags)
{
   if (0 == num_bytes) return;
   cout << "Client sent: " << c_mess << endl;
   c_wsabuf[0].len = num_bytes;
   memset(&c_over, 0, sizeof(c_over));
   WSASend(client, c_wsabuf, 1, 0, 0, &c_over, send_callback);
}
```

1:1 echo 실습

- 클라이언트 코드 중

```
while (true) SleepEx(100, true);
```

  - 클라이언트에서 Rendering Main Loop를 돌아야 하는데... 없어서 대신 넣음
  - SleepEx(100, true)
    - Busy Waiting으로 인한 CPU낭비 방지
    - True를 넣으면 Callback 함수를 처리해줌.

1:1 echo 실습

- 에러 처리 시 주의
  - WSASend/WSARecv는 항상 오류를 리턴한다.
  - 이때, Error 값이 WSA_IO_PENDING이면 정상적으로 IO가 등록이 된 것이다.

실습 순서

- 1 : 1 Overlapped Echo Client
- 1 : 1 Overlapped Echo Server
- n : 1 Overlapped Echo Server
- n : 1 Overlapped Chatting Server

n : 1 echo server

- 클라이언트는 변경할 내용이 없음
- 서버에서 여러 개의 접속을 유지해야 함
  - 여러 개의 소켓과 그 소켓에 대한 정보(추가 데이터)가 필요
  - 소켓에 대한 정보를 세션(SESSION)이라고 많이 부름
  - 따라서 unordered_map<int, SESSON>이 필요.
  - SESSION 내용 : recv를 위한 overlapped구조체, WSABUF, network_buffer가 필요함
    - Recv할 때마다 new/delete 하는 것 보다 효율적

n : 1 echo server

- 다중 접속 Accept

```
while (true) {
     SOCKET s_client = WSAAccept(server,
              reinterpret_cast<sockaddr*>(&cl_addr),
              &addr_size, NULL, NULL);
     add_session(s_client);
     do_recv(s_client);
   }
```

  - 모든 컨텐츠는 callback에서 실행되며 메인 루틴은 accept를 계속 받음
    - blocking socket API는 callback함수를 실행

n : 1 echo server

- Overlapped I/O사용시 주의
  - CallBack함수에서 어떤 소켓의 Callback인지 판단이 필요.
    - 해결 방법
      - 예) Overlapped 구조체의 hEvent 항목에 소켓 값 넣기.
      - 예) Overlapped 구조체의 주소를 socket값으로 매핑하는 자료구조 관리
  - Send/Recv에서 사용하는 Overlapped구조체는 독립적으로 사용해야 한다.
    - 하나의 구조체를 여러 곳의 Send/Recv에서 동시에 사용하면 안된다.
    - 재사용은 가능하나 Send/Recv가 완료된 후 Clear하고 재사용해야 한다.
    - echo_server의 경우는 하나의 overlapped 구조체의 재사용으로 충분 (게임서버는 아니다.)

n : 1 echo server

```
class SESSION {
private:
   int   _id;
   WSABUF _recv_wsabuf;
   WSABUF _send_wsabuf;
   WSAOVERLAPPED _recv_over;
   SOCKET _socket;
public:
   char   _recv_buf[BUFSIZE];
SESSION() {
   cout << "Unexpected Constructor Call Error!\n";
   exit(-1);
}
SESSION(int id, SOCKET s) : _id(id), _socket(s) {
   _recv_wsabuf.buf = _recv_buf;    _recv_wsabuf.len = BUFSIZE;
   _send_wsabuf.buf = _recv_buf;    _send_wsabuf.len = 0;
}
~SESSION() { closesocket(_socket); }
void do_recv() {
   DWORD recv_flag = 0;
   ZeroMemory(&_recv_over, sizeof(_recv_over));
   _recv_over.hEvent = reinterpret_cast<HANDLE>(_id);
   WSARecv(_socket, &_recv_wsabuf, 1, 0, &recv_flag, &_recv_over, recv_callback);
}
void do_send(int num_bytes) {
   ZeroMemory(&_recv_over, sizeof(_recv_over));
   _recv_over.hEvent = reinterpret_cast<HANDLE>(_id);
   _send_wsabuf.len = num_bytes;
   WSASend(_socket, &_send_wsabuf, 1, 0, 0, &_recv_over, send_callback);
} };
```

n : 1 echo server

```
#include <iostream>
#include <WS2tcpip.h>
#include <unordered_map>
using namespace std;
#pragma comment (lib, "WS2_32.LIB")
const short SERVER_PORT = 4000;
const int BUFSIZE = 256;
unordered_map <int, SESSION> clients;
void CALLBACK send_callback(DWORD err, DWORD num_bytes, LPWSAOVERLAPPED send_over, DWORD recv_flag)
{
   int s_id = reinterpret_cast<int>(send_over->hEvent);
   clients[s_id].do_recv();
}
void CALLBACK recv_callback(DWORD err, DWORD num_bytes, LPWSAOVERLAPPED recv_over, DWORD recv_flag)
{
   int s_id = reinterpret_cast<int>(recv_over->hEvent);
   cout << "Client Sent [" << num_bytes << "bytes] : " << clients[s_id]._recv_buf << endl;
   clients[s_id].do_send(num_bytes);
}
int main()
{
   WSADATA WSAData;
   WSAStartup(MAKEWORD(2, 2), &WSAData);
   SOCKET s_socket = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, 0, 0, WSA_FLAG_OVERLAPPED);
   SOCKADDR_IN server_addr;
   ZeroMemory(&server_addr, sizeof(server_addr));
   server_addr.sin_family = AF_INET;
   server_addr.sin_port = htons(SERVER_PORT);
   server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
   bind(s_socket, reinterpret_cast<sockaddr*>(&server_addr), sizeof(server_addr));
   listen(s_socket, SOMAXCONN);
   INT addr_size = sizeof(server_addr);
   for (int i = 1; ; ++i) {
      SOCKET c_socket = WSAAccept(s_socket, reinterpret_cast<sockaddr*>(&server_addr), &addr_size, 0, 0);
      clients.try_emplace(i, i, c_socket);
      clients[i].do_recv();
   }
   clients.clear();
   closesocket(s_socket);
   WSACleanup();
}
```

실습 순서

- 1 : 1 Overlapped Echo Client
- 1 : 1 Overlapped Echo Server
- n : 1 Overlapped Echo Server
- n : 1 Overlapped Chatting Server

채팅서버

- Echo Server 에서 채팅서버로 진화
  - 진정한 멀티플레이의 시작
  - 클라이언트에 다른 클라이언트의 메시지도 전달해 주어야 한다.
  - 메시지에 클라이언트 ID도 넣어 주어야 한다.
    - 전송 데이터의 Packet화
    - Packet 단위로 Send/Recv가 되어야 한다.
      - size정보 필요
      - 패킷의 구조 (size + id + message)
  - 더 이상 send와 recv가 차례로 호출 되지 않는다.
    - 정해진 순서가 없고, 동시 다발적으로 발생한다.

채팅서버

- Echo Server 에서 채팅서버로 진화
  - Send/Recv의 실행 순서가 존재하지 않는다.
    - Send_callback에서 더 이상 recv를 호출하지 않는다.
    - recv호출은 recv_callback에서 해야 한다.
  - Send/Recv가 더 이상 OVERLAPPED, WSABUF, BUF를 공유할 수 없다.
    - 완료를 기다리지 않고 send가 독립적으로 (비동기적으로) 실행되기 때문
    - 따라서 모든 send는 overlapped/wsabuf/buf를 별도의 객체를 사용해서 수행해야 한다.
      - 3개의 객체를 따로 관리하는 것이 비효율적이므로 하나의 클래스에 묶어서 관리한다. => class EXP_OVER
    - Overlapped 포인터로 WSABUF와 BUF의 주소를 찾아야 한다.

채팅 서버

```
class EXP_OVER {
public:
   WSAOVERLAPPED _wsa_over;
   size_t        _s_id;
   WSABUF        _wsa_buf;
   char          _send_msg[BUFSIZE];
public:
EXP_OVER(size_t s_id, char num_bytes, char *mess) : _s_id(s_id)
{
   ZeroMemory(&_wsa_over, sizeof(_wsa_over));
   _wsa_buf.buf = _send_msg;
   _wsa_buf.len = num_bytes + 2;

   memcpy(_send_msg + 2, mess, num_bytes);
   _send_msg[0] = num_bytes + 2;
   _send_msg[1] = s_id;
}

~EXP_OVER(){}
};
```

채팅서버

- WSABUF와 BUF의 관리
  - EXP_OVER를 통해 통합 관리

```
void do_send(int sender_id, int num_bytes, char *mess)
{
   EXP_OVER *ex_over = new EXP_OVER(sender_id, num_bytes, mess);
   WSASend(_socket, &ex_over->_wsa_buf, 1, 0, 0, &ex_over->_wsa_over, send_callback);
}
```

```
void CALLBACK send_callback(DWORD err, DWORD num_bytes, LPWSAOVERLAPPED send_over,
DWORD f)
{
   EXP_OVER* ex_over = reinterpret_cast<EXP_OVER*>(send_over);
   delete ex_over;
}
```

채팅 서버

- Recv Callback 수정

```
void CALLBACK recv_callback(DWORD err, DWORD num_bytes,
                    LPWSAOVERLAPPED recv_over, DWORD recv_flag)
{
   unsigned long long s_id =
      reinterpret_cast<unsigned long long>(recv_over->hEvent);
   cout << "Client [" << s_id << "] Sent[" << num_bytes;
   cout << "bytes] : " << clients[s_id]._recv_buf << endl;
   for (auto &cl : clients)
      cl.second.do_send(s_id, num_bytes, clients[s_id]._recv_buf);
   clients[s_id].do_recv();
}
```

채팅서버

- 클라이언트 수정 필요
  - send/recv가 정해진 순서 없이 실행
    - Scanf와 recv를 어떻게 동시에?? => Windows 프로그래밍 필요 => 아몰라!!!
  - recv시 자주 여러 개의 패킷이 뭉쳐서 도착
    - 분리 필요

```
void CALLBACK recv_callback(DWORD err, DWORD num_bytes, LPWSAOVERLAPPED recv_over, DWORD f)
{
   char* p = s_buf;
   while (p < s_buf + num_bytes) {
      char packet_size = *p;
      int c_id = *(p + 1);
      cout << "Client[" << c_id << "] Sent[" << packet_size - 2 << "bytes] : " << p + 2 << endl;
      p = p + packet_size;
   }
   do_send_message(); }
```

채팅서버

- 실습에서 누락된 사항
  - 서버에서도 클라이언트에 보낸 패킷이 뭉쳐왔을 때 처리가 있어야 한다.
  - 패킷이 뭉쳐 오기도 하지만 임의의 위치에서 잘려 오기도 한다. 이를 이어주는 코드도 필요

게임으로의 확장

- 클라이언트 변경
  - Avatar와 PC가 존재
    - Avatar : 나
    - PC : Playing Character, 다른 사람
    - 그래픽 상 구분 필요 (카메라는 아바타 연동)
- 서버 변경
  - 객체 상태 변화 => BroadCast (ID 필요: 클라이언트와 ID정보 공유)
  - Session확장 : X, Y, ID
- 패킷 종류의 다양화 => 프로토콜 정의 필요
  - 종류 : Object 추가/이동/삭제
  - 구성 : Size, Type, DATA(id, 좌표, Avatar여부)

숙제 (#3)

- 멀티플레이어 온라인 게임 서버/클라이언트 프로그램 작성
  - 내용
    - 숙제 (#2)의 프로그램의 다중 사용자 버전
    - Client/Server 모델, 서버는 반드시 Overlapped I/O callback 을 사용할 것
    - 클라이언트 10개 까지 접속 가능 하게 수정
      - 모든 클라이언트 에서 다른 모든 클라이언트의 말의 움직임이 보임
      - 클라이언트 접속 종료 시 해당 말이 사라지도록 구현
  - 목적
    - Windows 다중 접속 Network I/O 습득
    - Overlapped I/O 습득
  - 제약
    - Windows에서 Visual Studio로 작성 할 것
    - 그래픽의 우수성을 보는 것이 아님
  - 제출
    - Zip으로 소스를 묶어서 eclass로 제출
      - 컴파일 및 실행이 가능해야 함, 필요 없는 중간 파일 및 폴더들 포함 시키지 말 것

Reference

- https://www.microsoftpressstore.com/articles/article.aspx?p=2224047&seqNum=5
  - Overlapped I/O API 상세 설명

- 네트워크 연결상태 측정
  - https://github.com/microsoft/ctsTraffic