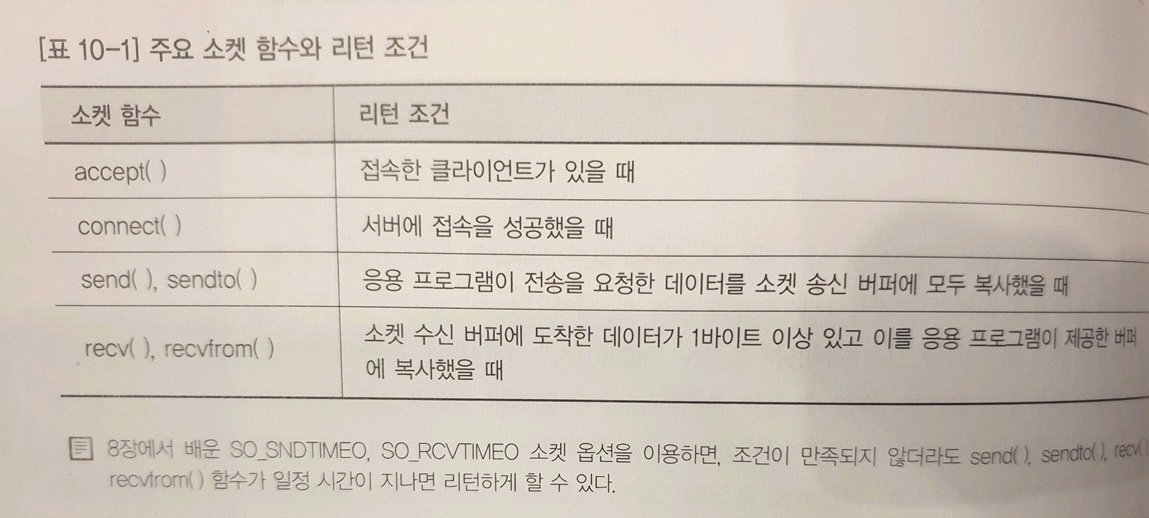
**소켓 입출력 모델(1)**

**소켓입출력 모델 개요**

(360P)



**UDP는 연결소켓이 아니기 때문에 connect를 쓰지않고 매번 send할때마다 목적지를 지정하여 sendto()와 recvfrom()를 사용한다.**

(송신 할때마다 IP와 포트를 실어서 보내는 것)

361P예제

**Ioctlsocket()**

**보통 ioctlsocket은 논블락 소켓을 만들 때 사용한다.(이밖에 쓸일이 거의없음)**

//소켓 생성

SOCKET sock = socket(AF\_INET, SOCK\_STREAM, 0);

if (sock == INVALID\_SOCKET) return -1;

//NON-blocked 소켓으로 전환

int err;

u\_long on = 1;

err = ioctlsocket(sock, FIONBIO, &on);

if (err == SOCKET\_ERROR) return -1;

**이때 non-blocked소켓은 항상 에러가 날수있는 상황이 되었기 때문에, WSAEWOULDBLOCK(조건이 만족되지않음)으로 진짜 에러와 구분한다.**

362P부터 들어가는 예제코드는 실무에서 전혀 사용되지 않는 모습이다.

**Non-blocked일 경우 send-recv가 반복될 때 위에서 accept가 되지않는다면 다른소켓의 recv로 들어가게된다. 이때 recv도 받을게 없다면 그냥 넘어갈 것.**

연결된 모든소켓을 훑어내야 하므로 list나 array로 관리하면서 recv를 시작하는것이다.

이 예제는 함수가 완료될때까지 함수단위로 무한루프를 돌고있다.

**모든 프로그램은 할일이있으면 깨어나서 해야하고 없다면 쉬어야한다. 의미없는 루프는 코어 하나를 모두 먹게될 것.**

(키보드 입력도 입력을 받으면 행위를하고, 우리가만든 WinAPI로 만든 2D게임도 프레임을 맞추기위해 sleep을 했었다.)

**Socket model**

**소켓모델은 recv send, accept를 할 수 있는 상황인지 판단해준다.**

우리는 결국 소켓모델을 채용하여 로직을 짜게된다.

**Socket model - Selet**

select모델은 버클리소켓 (리눅스)에도 똑같이 존재하며, 윈속에도 존재한다. 구시대적인 옛날 방식이라는 이유로 대부분 배우지 않기도 하지만 **IOCP를 잘못 만들경우 select보다 성능이 떨어지며, select를 이해하지못하고 IOCP를 통달하기란 불가능하다.**

언리얼의 데디케이트서버가 select를 채용하였으며, 우리도 select를 기본으로 작업한다.

**select모델은 구조가 간단하여 안전하다.(**또한 멀티스레드로 만들수는 있지만 멀티스레드를 전제로 만들어진 것은 아니다.)

따라서 제공될 더미들은 셀렉트 모델로 만들어질 것이며 더미는 서버를 테스트하는 도구이기 때문에 완벽해야한다. 나중에 문제가 발생할때도 가장 정확하고 확실한 검증도구인 셀렉트를 사용한다.

**select모델을 쓸경우 blocked소켓은 조건이 맞지않아 blocked상태가 되는 것을 막을 수 있고,Non-blocked 소켓에서는 호출이 되지않아 나중에 호출해야되는상황을 막을수있다.**

**((**

이때 예외가 하나 존재하는데, send는 블락걸릴 가능성이 존재한다.

**recv는 샐랙트를 거쳐서 남았다면 받을게 있다는 얘기이므로, 이것은 block상태가 될일이 없다.**

**send의 경우는 크기를 반환해주는 것이 아니기 때문에 크기가 모자라게되면 blocked상태가 될것이다.**

**따라서 blocked soket에서 select model을 썼다고해서 완벽하게 모든 blocked상황을 없앤다는 상황은 아니며, select model을 쓰더라도 non-blocked를 쓰는 것이 거의 정석적인 방법이다. (만에하나 blocked가 나오면 서버가 멈춰버리기 때문이다.)**

(여기서우리가 TCP스택의 버퍼에 카피하는 행위를 말하기 때문에 상대방의 윈도우사이즈는 상관없음)

**))**

**결국 recv, send, accept를 호출해야 하는 상황을 판단해준다는 얘기.**

**Select에 listenSocket을 넣으면 backlog큐를 확인해주며,**

**recv에서는 수신버퍼에 받은 데이터가 있는지 확인해주며,**

**send에서는 송신버퍼에 공간이 있는지 확인해준다.**

**따라서 소켓API함수를 호출하는 모든 행위들이 select라는 기반하에 호출되는 것이다.**

**FD\_SET구조체**

#define FD\_SETSIZE 64

typedef struct fd\_set

{

u\_int fd\_count; /\* how many are SET? \*/

SOCKET fd\_array[FD\_SETSIZE]; /\* an array of SOCKETs \*/

} fd\_set;

int WSAAPI select(

int nfds, //무시

fd\_set \*readfds,

fd\_set \*writefds,

fd\_set \*exceptfds, //OOB메시지. 제외.

const timeval \*timeout

);

반환값 : 조건을 만족하는 소캣의 개수 또는 0(타임아웃), 실패시 SOCKET\_ERROR.

**(recv와 send둘다 확인하고싶다면 read,write에 같이 넣어줘야할 것.)**

**Select를 호출하기전에 먼저 소켓 구조체에 확인하고자하는 소켓을 셋팅해야한다.**

FD\_SETSIZE가 64이므로 64개의 소켓을 한번에 검사를 해줄 것이며, fd\_count를 이용해 몇 개를 사용하고있는지 파악한다.(ex 개수가 4개라면 배열에서 4칸까지만 써먹을 것)

**Select – readset/writeset**

**우리가 해야할일은 FD\_SET를 하나 정의하고 readset(수신버퍼 체크용)과 writeset(송신버퍼 체크용)을 따로 준비하여 소켓을 셋팅한다. (셋팅 함수존재)**

**이후 함수를 호출하고 나면 모든 송수신버퍼를 확인 해줄 것. 조건에 합당한 소켓만 남겨두면 readset에는 받을게 있는(수신버퍼에 뭔가 들어온) 리시브를 할 소켓을, Writeset은 송신버퍼에 공간이 있는 소켓을 남길 것이다.**

때문에 우리는 select함수를 통해 확인이 필요한 소켓을 준비하고 검증하여 그에맞는 소켓API를 호출하게 될것이다.

때문에 blocked socket이라 하더라도 block이 걸리지않으며, Non-blocked socket이라면 절대 에러가 나지않을 것. (가능한 소켓만 남겨둘 것이기 때문)

**accept는 readset에 들어가게되며, 사용자가 들어와서 백로그 큐가 들어왔을떄 반응을 보이므로 이를 accept를 하면될 것.**

**이때 64개라는 개수를 늘리고 싶다면 msdn에서는 define된 FD\_SETSIZE라는 값을 winsock headr를 include하기전에 define을 새로걸고 하라고 한다.**

(리눅스 같은 경우도 커널을 재컴파일 해야하는상황)

일단은 64로 두고 진행한다.

추가로, 원칙대로라면 FD\_SET구조체를 직접적으로 건드리지 않아야 한다.

**조금더 성능을 높이고자한다면 결과를 확인할 때 FD\_SET구조체 멤버를 직접접근해서 확인해도 되지만 혹시 내부가 바뀔것을 대비해 함수를 랩핑해놨으므로 이것을 사용해야 한다.**

**Select – exceptfds**

인자 exceptfds 같은 경우, OOB를 뜻하므로 사용하지않으나, 딱한가지 예외가있다.

**Non-blocked소켓으로했을 때 connect도 블락걸리지않고 바로 WSAWOULDBLOCK를 리턴하므로, 완료 여부를 알려주지않는데,**

**writeset이 성공한다면 연결의 성공여부를 알수 있을 것.**

**또한 해당 exceptfds로 반응이 반응이오면 이는 connect에 실패한것이므로 connect를 시도할때는 예외셋을 셋팅할 것.**

(단 서버쪽이라면 connect를 하지않으므로 그냥 NULL값을 넣게된다.)

**select - timeval**

typedef struct timeval {

long tv\_sec; //초

long tv\_usec; //마이크로초

} TIMEVAL, \*PTIMEVAL, \*LPTIMEVAL;

**Select를 통해서 아무도 반응이 없을경우 기다리는 시간을 뜻한다.**

**하지만 어떤 반응이 있을경우 즉시 리턴하므로 해당시간을 기다리지않는다.**

**해당인자를 0을 넣을경우 무한정 대기하게 되는데, 이는 blocked상태로 CPU를 0%점유한다.**

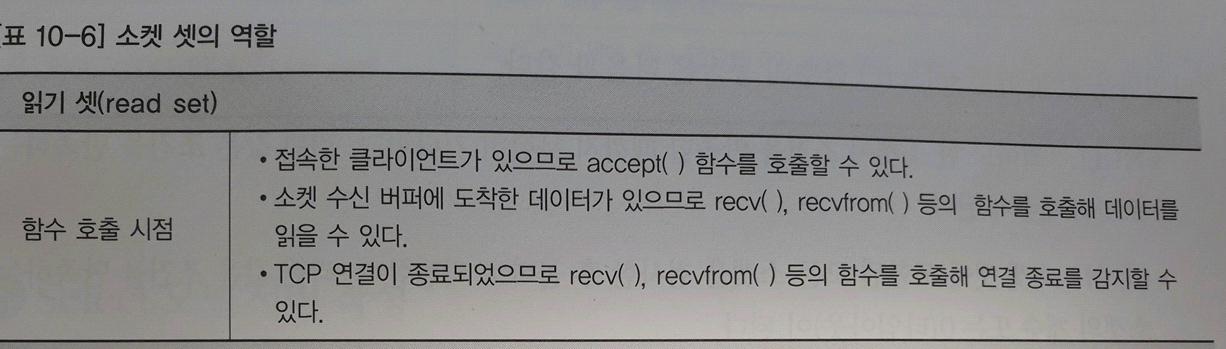
**Select의 리턴값**

Select는 반응을 보인 개수를 리턴해준다.

FD\_SET구조체는 64개가 최대치이므로 이를 한번에 검사하며, 여기서 하나만 반응을 보였다면 1을 리턴할것이고 얘를 찾아서 써먹어야한다.

Select의 리턴값이 timeout일 경우(timeval에 지정한 시간을 초과했을 경우) 0이나올 것.

**Readset**



Listen socket도 accept를 하고싶다고 막할 수는 없다.

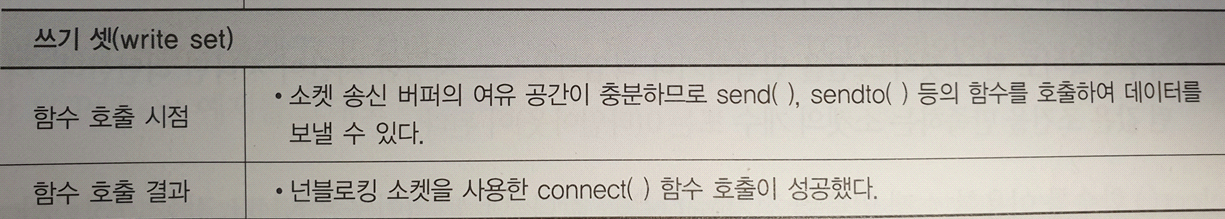
blocked소켓이면 blocked 걸릴것이고, Non-blocked라면 무한정 매 루프마다 돌려보는 것은 비효율 적이다.

**Listen socket을 readset에 넣고, 반응이 있을때(누군가로부터 연결이 성공이되서 backlog큐에 새로운 소켓이 들어온 경우이다.) 이때 accept를 한다.**

그럼 block걸리지않고, 에러도 나지않고 정확하게 새로운 소켓을 뽑아올 것.

정상적으로 연결이 종료(FIN전송)가 됐을때도 readset으로 반응이오고, 이때 recv를 할경우 리턴값이 0이 될 것이다.

**WriteSet**



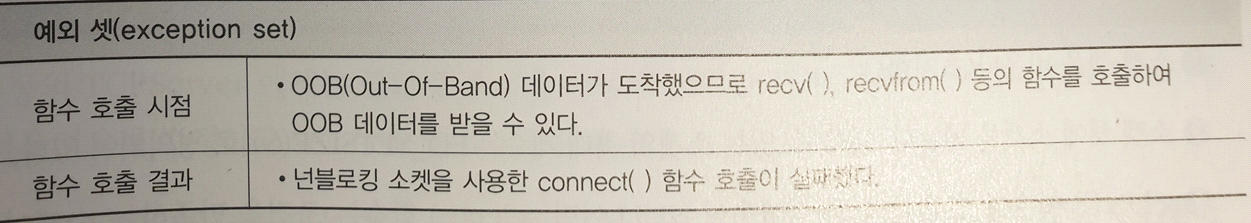
단, 여기서 충분이라는 기준이 없기 떄문에 그냥 공간만 있으면 바로 반응을보인다.

언급했듯이, write셋이 반응을 보여 send를 하는데 blocked소켓이라 block걸릴수도 있다.

그래서 select모델을 씀에도 불구하고 Non-blocked를 사용하라고 할 것.

일반적으로 select모델을 사용한다면 blocked소켓을 써도 괜찮다. blocked걸릴일이 거의 0에 가깝다. 딱한가지 우려하는 상황이 이것.

**Exception set**



우리가 클라를 개발할떄 Non-blocked소켓으로 전환해서 connect를 시도한다고 하면 이부분의 확인이 필요한 것.

(우리가만든 서버의 라이브러리의 연동될수있는 클라쪽 라이브러리는 우리가 만들어야하는 것.)

//qwer

**Select()함수를 이용한 소켓 입출력 절차.**

TimeOut은 0으로 가정한다.

**1.먼저 소켓셋을 비운다.(초기화)**

**2. 소켓셋에 소켓을 넣는다. 넣을 수 있는 소켓의 최대개수는 FD\_SETSIZE(64)로 정의됨.**

**3. select()함수를 호출한다. 타임아웃이 NULL이면 select()함수는 조건을 만족하는 소켓이 있을때까지 리턴하지않는다.**

**4.select()함수가 리턴하면 소켓셋에 남아있는 모든 소켓에 대해 적절한 소켓함수를 호출하여 처리한다.**

위사항을 반복한다.

이과정이 매우 비효율적으로 보일수도있으나 이를 반복하는 것이 정석이다.

(listen socket을 제외한다면 소켓셋에 넣을 수 있는 클라소켓은 63개.)

**Select모델을 사용하는 기본적인서버**

Select모델을 사용한다면 모든제어가 select를 통할 것이다.

게임서버와 일반서버의 차이는?

**일반적인서버(웹서버, 안드로이드 앱 등)는 클라이언트가 뭔가를 보내주면 받아서 대응하는 일을 처리하고 다시 blocked상태로 들어가지만, MMORPG라면 응답과 관계없이 월드를 돌려야한다.**

(timeout값이 0이면 blocked상태가 될건데, 할일이 없으면 서버는 쉬어야하므로 상관없음. )

**echo서버 만들기**

writeset까지 사용하려면 구조가 복잡해지기 때문에 일단은 readset만 사용한다.

**왜냐면 writeset은 send를 해도되는지 판단하기 위해서 호출하는데 send를 해야되는 상황마다 select를 다시할수는없다.**

그리고 내가 send를 해야하는데 select를 했더니 반응이없으면 그다음은 뭘할것인가?

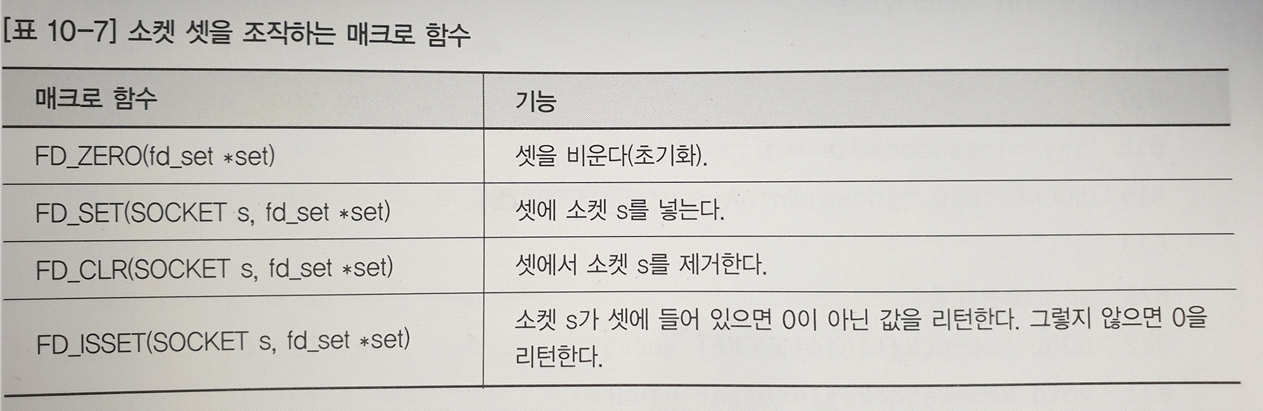
**지금상황에선 당장 할 수 있는게 없음. 그렇다고 멈출수도없다. 이 서버는 현재 63명의 클라이언트가 마구잡이로 요청을해도 아무도 block당하지않게 동시처리하는 것이 목적이며, 수시로 새로운사용자가 들어올수도 있어야한다.**

또한 송신버퍼가 꽉 찬다는 것은 상대방의 수신버퍼가 가득차야하고, 이것은 상대방이 받지않는것이므로 우리가 테스트하는 환경에서는 의도하지 않는이상 이런 상황은 나오지 않을 것이다.

Writeset은 굳이 체크하지않아도 정상적인 상황이라면 문제가 없을 것이므로 한동안은 체크하지않고 진행한다.

(원래대로라면 send가능 여부를 판단할 수 없으므로 L7계층의 버퍼를 만들어서, send를 하고싶을때마다 우리버퍼에 모아서 select를 거쳐서 보낼수있다라면 몰아서 send하게될 것.)

**FD\_SET을 제어하는 함수**



**FD\_ZERO – 초기화작업으로 Count만 0으로 만든다.**

**FD\_SET에 내가 확인하고자 하는 모든소켓을 셋팅한다.(**먼저 리슨소켓을 넣고 루프를 돌려 나머지 소켓을 넣는다.)

**echo서버**

**while(1)**

**{**

**FD\_SET ReadSet;**

**FD\_ZERO(&ReadSet);**

**FD\_SET(Listen Socket, &ReadSet);**

**for(클라이언트 소켓 ㅡ) //매번 새로 셋팅한다. 재활용은편법이므로 비권장**

**FD\_SET(Socket, &ReadSet);**

**ret = Select(0, &Readset, NULL, NULL, NULL);**

**//Select 호출후에는 반응을 보인 소켓만 남게되고 얘들을찾아 recv,send를 한다.**

**//우린 반응이없을 때 할게 없으므로 무한대기로 들어간다.**

**if(ret > 0) //이 구조라면 리턴값이 0이나올일은 없다. 타임아웃값이 0이기때문**

**{**

**if(FD\_ISSET(LitenSocket, &ReadSet)) //이값이 살아있다면 TRUE가 나오는 것**

**{**

**accept(ㅡ) //블락없이 새로운사용자.**

**신규 접속자 처리.**

**}**

**}**

**for(클라소켓 순회 ㅡ) <ㅡ여기가 맘에안들지만 방법없음**

**if(FD\_ISSET(ClientSocket, &ReadSet)) //클라이언트 소켓 전체확인**

**recv(ClientSocket.....);**

**ㅡ> send(ㅡ);**

**}**

**우리가 이전에 만들었던 에코서버는 세션이라는 개념이없었다.**

**(accept하여 socket하나 뽑아내고 소켓대상으로 send recv)**

지금 여기는이제 접속자들을 관리하는 것.

**이제 세션(연결의 단위)이라는 개념이 나오게된다. 지금 예제에서는 에코서버는 소켓만 관리해도 되므로 소켓하나를 하나의 세션으로 볼 것. (list로 했다면 그냥 push하면 될 것.)**

클라이언트를 순회하는 for문을 돌면서 어떤 소켓이 남았는지 확인하고, fd\_readset에 걸렸다면 recv를 하면될 것. (echo서버라면 대응되는걸로 바로 send.)

(만약 FD\_SET에서 직접 소켓을 뽑아서 진행하는 방식으로간다면 좀더 번거로움을 덜수있지만, 이전에 언급했듯 제공하는 함수를 통해 정석적으로 진행한다.)

372P분석

SOCKETINFO가 나온다. 연결의 단위. 들어올때마다 SOCKETINFO가 확보

밑에 버퍼가 나오는데 이 버퍼가 송신버퍼이자 수신버퍼를 겸하게된다.

지금우리도 대비가 안되어있다. 위코드로 보자면 recv를하고 send를 하는데 여기서 나온 리턴값이 100이라면 그리고나서 send를해서 100을 넣으면 여기리턴이 100이라는 보장이없다.

논블락은 (샌드는 리턴 값이카피에 성공한 값이다. 블락소켓에서는 요청한 길이가 그대로나온다. )할 수 있는 데까지하고 사이즈를 리턴한다.

그럼만약에 100을 받아서 100을 넣었는데 리턴값이 50이 나왔다. 50을 안보냈네? 그럼이거 어떻게되나? 다음에 보내야한다. 이를 보존시키고.

각 연결단위 세션마다 통신 버퍼를 만들어두고 보존시켜놓고 보낼수있는 상황이 되면 보내야된다.

이걸그냥 버리면 데이터가 날아가는 것.

그래서 sockeetinfo는 수신겸 송신버퍼가 있을 것.

원칙대로면 송/수신버퍼로 2개가 들어가야할 것.

이구조는 하프 디플렉스이다.

((

**전이중 통신**(Full Duplex): 두 대의 단말기가 데이터를 송수신하기 위해 동시에 각각 독립된 회선을 사용하는 통신 방식이다. 대표적으로 전화망, 고속 데이터 통신을 들 수 있다.

**반이중 통신**(Half Duplex): 한 쪽이 송신하는 동안 다른 쪽에서 수신하는 통신 방식으로, 전송 방향을 교체한다. 마스터 슬레이브 방식의 센서 네트워크가 대표적이다.

))

송수신이 동시에 되는 구조가 아니다. 핑퐁처럼. 리시브하고, 샌드하고.

그래서 버퍼하나를 겸용으로 쓴다. 잘못만들어진것.

소켓인포안에 recvbytes, sendbytes가 있을 것. 몇바이트를 받았고 몇바이트를 보냈는가 판단기준

(ㅒ103번째줄)Recv를 했을 때 error만 확인하고 추가적인 판단은 하지않는데, 원칙대로라면 우드블럭이냐 아아니냐 판단할 것. 어쩄든 지금은 싱글스레드고 셀렉트모델을 거쳐서왔기 때문에 그럴가능성이 없다라고 전제하에 간다.

셀렉트모델에선 리시브는 항시. 보낼게있으면 writeset에 보내는 것.

**과제. Star 서버-클라 만들기**

GetAsyncKeyState를 써서 키보드입력을 받기 때문에 프로세스 구분이 없어서 동시에 움직이게됨.(윈도우용으로 getch, scanf\_s는 활성화된 콘솔만 받았음.)

추가로, 서버에서 이런식으로 현재 접속자를 눈으로 보이게하지않고 렌더한다.

**TCP는 언제든지 뭉쳐지거나 조각날수있다는 전제로 설계를 해야하므로 항상 링버퍼가 나오게된다.**

**뭉쳐진다면 버퍼에 받아서 나눠서 처리를 해야하고, 조각이 났다고 한다면 몇바이트를 받긴받았는데 아직 완료가 안됐다면 보관해놓고 이어받아야한다. (링버퍼:바이너리환형 큐의 사용)**

링버퍼가 없는 상태에서 가장 간단하게 이 문제를 해결하는 방법은 모든 메시지를 규격화 것.

하지만 데이터를 받은다음 어떤 종류인지 확인, 길이확인, 잘라내는 작업 등이 매우 번거로운데,**기본적인 단계이므로 16바이트로 규격을 지키고 뭉쳐져오는걸 해결하기위해서 리시브를 16단위로 한다.** **(수신버퍼에 데이터가 있다고해도 나머지는 그냥 수신버퍼에 둘것)**

이는 마치 소켓에 있는 수신버퍼를 링버퍼처럼 쓰자는 것인데, 그렇다고해서 조각이난건 아직 해결 되지않는다.

**데이터가 조각 나는 경우는**

**1. 상대방의 윈도우사이즈가 없는 경우**

**2. Nagle알고리즘이 작동이되는 상태에서 send를 한다면 네이글이 작동되면 아크가 오거나 MSS에 도달이 되면 일단 MSS만큼 보내버릴 것.**

하지만 이런상황이 나올가능성이 정상적인 상황이라면 없다.

**1. -> 상대방이 바로바로받는다면 윈도우사이즈가 없는상황은 나오지않을 것이고**

**2. -> 어차피 MSS에 16바이트를 메시지로 만들어보내는데 송신버퍼에 16바이트까지 송신버퍼에 쌓일가능성이 거의없다. 그래서 LAN환경에서는 잘될거라고 생각하고 그냥보낸다.**

**메시지 프로토콜 - 모든 메시지를 16바이트 고정**

**ID할당(0) Type(4Byte) | ID(4Byte) | 안씀(4Byte)| 안씀(4Byte)**

**별생성(1) Type(4Byte) | ID(4Byte) | X(4Byte) | Y(4Byte)**

**별삭제(2) Type(4Byte) | ID(4Byt** IsMove**e) | 안씀(4Byte)| 안씀(4Byte)**

**이동(3) Type(4Byte) | ID(4Byte) | X(4Byte) | Y(4Byte)**

**(0~2는 서버에서 클라로 보내는 패킷 3번은 쌍방향으로주고받는 패킷)**

**처음에 어떤 클라이언트가 연결하면, 서버입장에서는** (실제 서버입장에서는 그 유저에 데이터에 따라서 생성을 시킬 것) **임의 좌표에 유저에 해당하는 별과 번호를 만들고, 어디 생성됐다고 클라이언트에 알려준다.** (이때 다른클라이언트(별) 도 알려줘야 할 것)

**Stateful에서는 모든 것을 서버기준으로 가기 때문에 실제 이동을하면 서버에서도 이동을 한다.접속을 끊으면 해당 유저를 지울것이며, 다른클라한테도 이를 전파한다.**

**TYPE**

이 프로젝트에서 헤더가 된다.

우리는 4byte단위로 가기 때문에 pragma pack을 하지않아도 상관없음.

(이 패킷의 경위 무조건 16비트로 맞추다보니 불필요한 공간이 존재)

**별생성(Type 1) : ID와 좌표를 받아서 별객체를 생성한다.**

**별삭제(Type 2) : 이프로젝트에서는 나간경우 밖에 없을 것.**

(필드가 커서 Sector처리를 할경우 클라는 해당Sector에 있는 클라만 인식할것이며, 클라가 종종료한 것과 sector에서 벗어난 것을 구분하지않는다.)

**별이동(Type 3) : 서버가 클라한테 주는메시지.**

**원칙적으로는 클라->서버, 서버->클라 메시지로 분리해야 한다. 설령 내용이 똑같더라도 반드시방향성을 부여할것.**

클라이언트가 서버에 최초 접속시

- ID할당(0) 패킷 수신

- 별생성(1) 패킷 수신 (자기 자신꺼)

- 별생성(1) 패킷 수신 (남들이 있을 경우에만)

대부분 클라 개발자들이 메시지와 메시지들을 묶어서 종속시킨다. (ID할당이 되면 생성하는식)**하지만 메시지는 반드시 독립적이어야하며, 설령 순서가 바뀌어도 문제가 없어야한다.**

((

**메시지가 독립적이어야 하는 이유(예시)**

유저가 마우스를 클릭하는건 서버한테 요청하는 것이며, 서버로부터 결과가 오면 클라에서 움질일 것이다.

이때, 이 서버 프로토콜을 설계를 보고 대부분의 클라이언트 개발자가 클릭하는것과 이동경로가 오는 로직을 종속시켜 버린다.

이 경우 이동경로가 오지않았는데 다른경로를 클릭할 경우 코드가 점점 복잡해진다.

**클릭하는건 서버한테 요청하면 끝이며, 그뒤는 생각하지않는다.**

**이동은 클릭을 했던안했던 메시지가 오면 가는 것.**

))

**MOVE**

이동할 때 클라에서 자기좌표를 먼저 바꾸고 서버한테 통보한다.

이후 서버가 그걸받고 다른 클라한테 뿌리는 것.

서버사용 포트 : 3000

이동범위 : 80 x 23

**# 참고 사항 : 일어날수 있는 현상들**

**1. 클라이언트를 껐는데 별이 서버에 남아있다.**

(이는 지금에는 잘 나타나지않지만 더미를 대량으로 할때는 나타남)

클라이언트 프로세스를 강제종료 하거나 디버깅 종료 과정중 가끔 나타나는데, 네트워크 연결이 실제로는 끊어졌는데 이에대한 끊기 과정을 수신하지 못하하여 서버쪽에선 여전히 접속이 된 것으로 오해를 하게된다.

**만약 ESTABLISH상태가 남아있다면 이는 TCP계층에서의 종료처리가 제대로 안된 경우이다.**

이는 하트비트나 Keepalive 로 해제를 시켜주어야 하므로 현재로서는 그냥 넘어가도 무방.

**(Select model)Client로직설계**

대신 클라이언트라는 객체는 여러 개 나와야할 것. (렌더해야 하므로)

**select코드는 루프에 단 한번만 들어가서 로직을 중심으로 돌아가야한다.**

**(정석아님. 우리코드한정)**

SOCKET g\_Socket;

main(ㅡ)

{

소켓생성

커넥트 (Non-blocked소켓으로 연결의 성공은 writeset을, 실패는 예외셋을 확인한다.)

while(1)

{

키보드

네트워크() ㅡ 밑에따로빠짐

나의 player검색. 찾았다면~

if(VK\_LEFT)

right, up, down 판단 //좌표가 바꿁경우에만 send!! 눌릴떄마다 send아님.

send예외처리.

(send예외처리중 16바이트 이하로 나오는 것과, WSAEWOULDBLOCK의 반환도 고려하지않는다.이는 송신버퍼가 다찬것으로, 가능성이 없음.)

**원래대로라면 이 부분은 컨텐츠 루프에서 L7송신버퍼에 디큐한 다음, (전체 컨텐츠 루프에서 보낼게 있을때마다 송신버퍼에 인큐) 메인 셀렉트에 들어와서 체크하는 것.**

이는 보낼것들을 모아놓고 한번에 보내는 것을 뜻한다.

실제로 리시브와 send는 한프레임당 한번씩 하는 것. 필요할때마다 하는게 아니다.

}

(내좌표가 이동된 경우)

msg

msg.type = 3

ID = myId...

}

**(Select model)Client로직설계 – Connect**

(Connect는 해당 프로젝트에서는 blocked소켓으로 가도 상관없으나, Nonblocked로 진행)

Connect함수는 block이 걸릴경우 게임의 main루프가 멈춰버리기 때문에 connect의 결과 판단을 main select까지 끌고가지 않게끔 설계를 잡아봤다.

**1. 이는 connect함수를 랩핑하는 함수를만들어, 이 함수의 return값으로 메인select까지 가져오지 말자는 뜻이다. 하지만**

이 경우 멈추는 문제(포트막힘, SYN응답없음)는 위해서 Non-blocked로 바꾼것이고, Select를 거쳐야하는데 **이 문제의 경우 Select역시 결과를 바로 주지못함.**

**2. 때문에 멈추는 현상을 없애기위해 connect후에 이 연결여부에 대한 판단이 main루프 안으로 들어가야 한단 얘기가 된다.**

우리가 원하는 것은 프로그램이 정상적으로 도는 것처럼 (연결중입니다…)와 같은 씬이 뜨게끔하는것이다. 이경우 루프를 돌면서 연결이라는 행위가 또다른 별도의 씬이 나와야 할것이므로 굉장히 복잡해진다.

**Non-Blocked Connect를 하는경우는 블락걸리는 시간을 제어하겠다는 뜻이므로, 결국 우리가 택하는 방법은 300m/s를 초과할경우 연결실패로 판단하고 바깥으로 빠지는 것이다.**

(Connect는 timeout제어가 되지않으므로.. recv소켓은 가능할것.)

**(Select model)Client로직설계 – NetWorkPart**

struct MSG

{

type

id

x

y

}

g\_MyID;

FD\_SET, ReadSet;

초기화, 소켓생성

timeval time;

time.tv\_sec = 0; //(프레임에 맞게 돌아가야한다.키보드에 입력에 돌아가야하는 상황.)

NULL값(=무한정대기)이 들어가야 하는 상황은 아니다. (0.0은 가능)

그럼 약간의 시간을 넣을 것. 몇m/s만.

ret = select(ㅡ)

if(ret>0)

msg;

r= recv(socket, &msg..)

switch(msgtype)

case 0: g\_myID=msg\_ID;

case 1: CreateStar(msg.ID,x,y);

case 2:삭제..이동.,..등..

case 3:

**이구조대로라면 select를 하고 recv를 16바이트씩 메시지 한 개씩만 받는다. 이렇게되면 한프레임에 메시지를 하나만 받게될것이고, 여러클라가 동시에 붙는경우 내 클라이언트에서 점점 느리게 반응하게 될것이다.**

**원칙대로 간다고하면 recv를할 때 받을수 있는만큼 최대한 받아서 일괄처리하는 것이 원칙이다.**

버퍼를 크게잡아 루프를 돌면서 처리하거나, select의 리턴이 0이 나올때까지 Select-recv를 반복적으로 돌리라는 얘기이다.

(이는 비효율적인 방법이며, 링버퍼가 없기 때문에 다음과같이 한다)

위 로직에서, Nagle알고리즘을 키게되면 버퍼에 데이터를 쌓아놨다가 전송하기 때문에 약간씩 버벅거리는 것을 확인할 수 있다.

**(Select model) Server 로직 설계**

**Sessionㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ**

**Socket**

**IP, Port (sockaddr). 유저에 따라 연결정보.**

**ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ**

**이제는 세션(연결의 단위)이라는 개념이 나오게된다.**

**세션마다 고유한 키가 나올것이고, 이를 활용한 플레이어가 나오게 될 것.**

**이때 모든 네트워크 어플리케이션에서는 관리차원에서 연결된 사람의 IP와 포트는 항상 알고있어야한다**.(ex 로그를 찍거나 차단을 하기위함) 따라서 Accept를 받을때 나오는 것들은 sockaddr구조체를 들고있거나, 처음부터 문자열로 변경시킨 형태로 보관하기도 한다.

(이후 원래대로라면 송수신 버퍼가 들어가지만 지금 우리한테는 필요 없는상황.)

**다음으로 컨텐츠 파트인데 이프로젝트에서는 ID,좌표 두개밖에 없으나 실제 게임이라면 이정보들은 분리가 될 것.(**ex 플레이어쪽 아이템정보/퀘스트정보/파티정보/…)

플레이어는 모든 컨텐츠를 담고있기 때문에 객체 덩어리가 커질수밖에 없다.

이때 플레이어 객체를 관리하기위해 자료구조를 통해 보관을 한다.

해당프로젝트에서는 이미 (send와 recv를 수행했다는 것은 알고있는 상태) 누군지 정확히 알고 접근을 하는 상황이다.

**만약 n번유저가 n번유저에게 특정 행동을 한다면 이 경우를 검색해야 할 것.**

Listen Socket()

MyAccept()

Disconnect(세션\*)

MoveStar(세션\*,X,Y)

ID, X, Y //서버가 ID보유

**실제 게임이라면 회원의 고유번호를 쓸 것.(회원이 가입할때마다 부여되는 주민번호)**

list<seesion\*>

g\_ID = 0; // Counting

sendPacket\_unicast( 세션, char\*,size(16고정))

//패킷을보내는 함수를 만든것(인자는 소켓이나 세션)

sendPacket\_Broadcast (제외시킬세션, char\*)

**//여기에서는 broadcast는 UDP가 아님. MOVE와 같이 모두에게 뿌리는 상황을 말하며, 자기자신은 제외하고 뿌려야한다.**

(이때 이 프로젝트 구조에서는, A클라이언트가 이동한 것을 다시 A클라이언트한테 전송한다면 왔던 자리를 되돌아가는 현상이 보일 것. 클라에서 이동을 하고 서버에서 뿌려주는 형태이므로 클라가 이동했으나 예전이동에 대한 결과를 또다시 받으면 되돌아가게됨.)

**이메시지를 모두한테 다 보내는 상황이라면 매개변수로 NULL포인터를 넣으면 될 것.**

(추가로, 지금 프로젝트에서는 Render를 하지만 테스트할때는 모니터링 정보만 찍을것이며, 실제 라이브서비스에는 무언가를 출력하는것자체가 컨텍스트 스위칭이 되므로 낭비이기 때문에 출력조차 하지않는다.)

학원에서 프로젝트를 진행할때는 몇몇가지 모니터링상태와 필요한 정보 레이아웃을 가지고 갈 것.

main()

리슨소켓 준비 Listen()

listen(ㅡ)

while(1)

{

FD\_SET REadSet;

초기화

FDSET(ListenSocket,&readSet);

for(세션 리스트 순회) //만약 64명단위로 끊어야할 것. 우리느 고려x

{

FD\_SET(\*iter->Socket. ,&ReadSet);

}

timeval -> 0

//서버라면 timeout값은 NULL. 들어와야 할일이 하기 때문에. 기본프레임자체가 의미가 없는상태.

r = Select(0, &ReadSet, NULL, NULL, NULL);

if ( r > 0 ) **//여기서 (리슨소켓이)실패하는경우는 아예 소켓이 아닌걸 잘못넣었을때이며, 소켓이 끊어지는 경우 (소켓은 핸들) 일반 숫자값으로 변하기 때문에 실패하게되는 것.**

if(FD\_ISSET(ListenSocket, &ReadSet))

MyAccept();

for(세션리스트 ㅡ)

if(FD\_ISSET(\*iter->Socket, ReadSet)) <이조건이 맞으면 리시브.

**//사실상 거의 모든 서버의 부담은 로직이다. 생각외로 네트워크 파트의 성능이 크게 좌우하지않는다. 따라서 서버엔진도 오버스팩이라고 많이들 말한다.**

recv(세선->socket, &MSG, 16, 0) //WOULD블락 뜰때까지 돌리기

if( == 0 에러)

Disconnect(세션)

Switch(MSG, Type)

{

case 3:

MoveStar(세션, Msg.X, Msg.Y)

}

}

(네트워크에 대한건 만들어져있고, 컨텐츠를 붙여야 하는 상황일 때를 코딩을 시작할때는 생각나는대로 적는게 아니라, 대강 설계하는 형태로 할 것.)

**Accept() 함수**

**FD\_ISSET에서 반응이 왔을 때 accept하고 새로운세션만들고 삽입하는 일들을 함수로뺸다.**

Socket = accept(ListenSocket)

if(예외처리 : 소켓이 나오지않았을 때)

//위 accept의 예외의 경우는 Listen소켓이 닫혀진 상황말고는 없다. 누군가가 연결을 하고 끊었다고 해도 backlog큐에 들어왔기 때문에 accept는 떨어진다.

**실제 서버에서는 accept스레드를 별도 쓰레드로 빼서 blocked소켓으로 관리하게 된다.**

세션생성

세션IP, Port 셋팅

세션 ID = ++g.ID

X= 랜덤

Y= 랜덤

세션

//기존의 다른 별들이 있다면 낱개의 생성패킷을 보내 모두에게 쏴줘야한다.

//그리고, 지금들어온 이 세션한테 기존별의 생성정보를 다른애들한테 쏴줘야한다.

**화면에 서버입장에서 기존별들이 있는데, 새로운별이 들어왔다면 이것에 대한 정보를 기존별들에게 전파를 해야할 것.**

이때, SessionList에 먼저 추가한다음에 broadCast를 해버리면 나한테도가고 남한테도 가게된다. **따라서 로직을 간단하게 하기위해 순서를 바꿔, 기존에 있던 별들의 정보를 지금막 생성한 별한테 보내고 이후 SessionList에 추가한뒤에 새로운 별을 브로드캐스팅 하여 코드를 좀더 간단하게 만든다.**

(프로토콜이 서로 독립적이기 때문에 나에대한 별생성이 뒤늦게와도 문제될건 없을것.)

**MSG. << 이 메시지 구조체가 여기저기서 사용된다고 전역에놓고 재활용하는 것은 오히려 성능이 떨어진다.**

Msg.Type = 0

ID = 세션 ID;

SendPacket\_Unicast(세션,메시지) // 이안에서는 해당세션에 접근을 해서 샌드를 할 것 이렇게 랩핑하는이유는 send에 대한 예외처리를 한곳에서 관리하기 위함이다.

(broadCast역시 내부에서 SendUnicast를 호출할 것.)

**관리용이를 위해 API Socket Send는 단 한곳으로 모을것**

**새로운 세션을 다른클라한테 뿌리는 로직**

for(세션리스트 순회) //한명씩 순회하면서 메시지를 다만든다.

{

Msg.Type = 1;

Msg.ID = iter->ID;

X = ㅡ

Y = ㅡ

SendPacket\_Unicaset(세션, &MSG);

}

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

**세션리스트에 방금들어온 세션삽입**

MSG.Type = 1

ID=세션.ID

X,Y셋팅

sendPacket\_BoradCast(NULL, ㅡ)

**게임서버에서 제일 큰 부담은 플레이어를 맵에 등록할 때, 주변의 오브젝트에게 이를 알리고 주변오브젝트의 정보도 플레이어게 알리는 것이다.** 주변에 있는 모든 NPC 캐릭터 몬스터 의 정보들을 다취합해서 보내야하고, 들어온애의 대한 정보도 주변에 다뿌려줘야한다.

더군다나 새로 접속하는 지점은 보통은 사람이 가장 많은 곳이다.

Sector처리가 되면 이 Sector를 옮겨다닐떄마다 이러한 로직이 계속 돌기 때문에 부담이 커질수밖에 없음.

**Disconnect(세션)함수.**

closeSocket(세션,Socket);

MSG.Type = 2;

ID = 세션ID;

세션리스트에서 삭제.

send Pacekt\_BroadCast(ㅡ); //재귀위험

누군가가 끊겼기 때문에 Disconnect가 들어온것인데, Disconnect의 결과로서 다른애들한테 보내야하는상황이다. 그럼 이안에서 샌드를 하다가 또 Disconnect가 호출이 되어 어느정도의 재귀현상이 발생할수있다.

**MoveStar(세션, X, Y)**

세션좌표 이동 (이동할수없는곳으로 이동하는것에대한 예외처리를 해줘야할 것)

실제 게임에서 컨텐츠적인 보안이 필요하다면 말도안되는 이동도 감지해야할 것.

두,세칸을 뛰어넘거나, 갈수없는곳을 가려고한다등의 메시지는 무시하는 식으로.

로그를 남겨서 핵유저로 처리해버릴수도있으나, 기본은 무시.

MSG.TYPE = 3;

ID = 세션.ID

X

Y

SendPacket\_BroadCast(세션, &MSG)

지금 보면 MSG구조체를 정의하는 경우가 계속 보이는데, 지금은 간략하기위해 그때그때 셋팅한다. 우리가나중에 원칙대로라면 메시지별로 메시지를 생성하는 함수를 하나씩 만들어서 일괄적으로 관리하게된다.

SendPacket은 똑같음

system(“cls”)의 경우 굉장히 느림.

**세션삭제 문제**

위 예제에서 세션자료구조를 list로 만들었다면 Disconnect에서 세션을 삭제하는 과정에서 로직이 꼬일 수 있다.(iterator)

정상적인 프로젝트라면 send도 select를 거칠것이기 때문에 해당구조에서는 볼 수 없는 문제. (브로드캐스팅 send를 한다고해서 끊어진다는 것을 감지할수 있는 상황이 아니다.)

굳이 이구조에서 해결을 보고싶다면, (현업에서도 종종사용하는 방법) **삭제할 데이터만 따로 관리해서 이후 일괄적으로 관리하여 삭제하는것이다.**

(멀티스레딩에서도 동기화문제가 터지기 때문에 동기화를 간단하게 하기위한 방법중하나.)

**추가적으로 알아둬야할 것**

**L2장비, L3장비도 포트단위로 고장이 나는 경우도있음.**

**우리는 수신버퍼에 들어온 데이터를 모두 처리하는 것을 원칙으로 한다.**

**1. 이때, 자료구조에 보관한 데이터(세션)를 for문을 돌면서 차례대로 메시지를 처리할것인데로직의 앞쪽에 있는 유저가 처리량이 더 빠른 것은 문제가 될것인가?**

**이는 for문을 돌고있는 시점에 따라 우선순위가 달라지는것이고, 이는 찰나의 순간이기 때문에 유저가 거의 느낄수도 없다.**

**2. 그리고 특정클라가 데이터를 다른클라보다 훨씬 많이 송신하는경우, 이 특정클라를 더 많이 처리를 할 수밖에 없는 문제는?**

**네트워크 라이브러리는 무조건 처리하는 것을 원칙하고 있으므로 이를 컨텐츠쪽에서 걸러내며, 이는 각각의 컨텐츠마다 메시지의 텀(규칙)과 메시지가 올수있는 횟수를 지정하는 방식으로 가게된다.**

**C#서버와 IOCP**

네트워크 라이브러리를 IOCP로 만들었을 경우 최소한 2천만건이상의 초당메시지를 처리할 수 있어야한다. (cgc의 경우 초당 4천만건 처리)

여기에 미치지 못할경우 C#으로 가는 것이 성능이 좋을것.

거의 대부분이 C#보다 성능이 느리지만 C++를 사용했다는것만으로 IOCP가 더 빠를것이라고 생각한다.

**UDP**

**1.연결설정을 하지않으므로 connect()함수를 사용하지않는다.**

**2.프로토콜 수준에서 신뢰성있는 데이터 전송을 보장하지 않으므로 필요하다면 응용프로그램수준에서 신뢰성있는 데이터전송 기능을 구현해야한다.**

**3.간단한 소켓함수 호출절차만 따르면 여러대상을 상대로 통신을 쉽게 구현할 수있다.**

**4.TCP와 달리 응용프로그램이 데이터경계구분을 위한 작업을 별도로 할 필요가 없다.**

Sendto, recvfrom

계속해서 주소를 넣어줘야한다.

**TCP의 connect는 3wayhandshaking의 진행완료가 아니며, 소켓 내부라이브러리에 기억시키는것.**

**UDP는 데이터가 왔을 때 포함된 주소로 우리가직접 세션을 찾아내야한다. 이는 매우 번거로운 작업.**

(recvfrom으로 온 주소를 그대로 넣고 sendto할경우 그대로 전송받은 곳으로 send하는 것)

**소켓이라는 구분자가 없기 때문에 구분자를 구현해야하는 번거로움이 있지만 데이터그램방식이기때문에 딱 보낸만큼만 송수신됨.**

**때문에 RUDP를 구현하고자한다면 시퀀스번호와 ACK와 같은 기본적인 개념부터 TCP의 거의 모든 개념을구현해야한다.**

**<4.17>**

**링버퍼**

TCP에서는 샌드/리시브를 하기위해 링버퍼를 사용한다.

데이터를 받고 붙이고를 반복해야 하기 때문에 배열을 사용한 환형 큐 방식과 일치한다.

**최소크기는 이 게임에서 받아내는 최소 메시지는 되어야 할 것이고, 최대한 큰게 좋을것. 대략 10000바이트로 잡고간다.**(recv를 최대 만바이트 하겠다.)

**Recv 할 때 사실 저게 다차는 것은 정상적이지 않은 상황이다.**

**(송신 링버퍼가 다 차는 경우는 1.상대의 수신버퍼, 2. 우리쪽의 송신버퍼, 3. 우리쪽의 l7링버퍼까지 다 찬것이므로 이는 비정상적인 상황)**

(때문에 resize를 하는 일은 없을 것)

링버퍼는 하나의 세션당 하나씩 주어지는 것이로, 전체를 대상으로 하나의 링버퍼가 있는 것이 아니다.

**하나의 연결당 개인개인마다 송신용 수신용 한 개씩 두는 것**

(UDP에서는 링버퍼가 없어도된다. 링버퍼는 STREAM방식이기에 사용하는것.)

헤더가 2byte일 경우를 가정한다

**select검사 이후,**

**if(FD\_ISSET(...))**

**{**

**char Buffer[1000];**

**size = recv(Socket, Buffer, 1000, 0);**

**session->RecvRingbuffer.Enqueue(Buffer,size);**

**if(RecvRingbuffer, GetUseSize()) < header.size)**

**중단(수신행위 중단)**

**// 최소한 헤더가 왔는지 체크. 이 헤더가 진짜 우리것이냐 맞느냐는 체크섬과 같은 안전장치로 파악을 해야하는 것.**

**ret = 세션->RecvRingbuffer.peek(&header,2);**

**//디큐를 했는데 메시지가 다 온상태가 아닐수있으므로 peek으로 헤더부터 살펴본다.**

**if(ret != 2)**

**중단**

**//(위에서 예외를 체크했는데 위 분기문으로 들어왔다는 것은 일어나서는 안되는 상황으로, 일부러 NULL포인터를 참조해(오류) 크래쉬를 유발하여 덤프를 찍을 것.)**

**RecvRingbuffer.MoveFront(2);**

**//헤더를 peek으로 확인했으므로 뒤의 데이터를 읽는다.**

ret = RecvRingbuffer.Deque(&packet, headr);

if(ret != header)

**강력한 예외처리(중단);**

Packet Proc(......)

**//모든 처리가 끝났다면 아직 처리해야할 메시지가 남아있는지 확인한다.**

}

**Send의 경우.**

**Select의 Writeset은 거의 모든 애들한테 반응이 오기때문에 보낼게있는 소켓을 추려서 넣는다. Send용 링버퍼에 뭔가 데이터가 1바이트라도 있는애들을 추려서 보내는 것.**

(이 구조라면 send는 한프레임당 1번.)

정석적인방법.

**수신메시지처리 – 송신 - 로직처리 - 추가:(송신)** (다시 센드하는 네트워크 코드 추가)

**싱글스레드 서버환경에서는 로직처리후 다시 수신메시지를 처리하고 송신한다면, 한박자늦게 반응한다고 생각할 수도있다.**

**(recv이후 바로send를 탈수도있겠지만 링버퍼에 저장되고 다음 프레임에 돌 확률이 올라간다.)**

**하지만 이것이 원칙이므로 그냥 이대로 간다.(ex 언리얼 데디케이트 서버)**

**이때 로직처리후 추가송신 을 붙이면 한프레임에 네트워크로직이 두개가되지만 이러면 반응속도가 좀더 좋아질 것.**

**이 경우는 메시지를 받고 이에대한 결과가 즉시 전달되는(ex이동) 경우 바로 결과값을 뽑아낼수있는 상황으로, 로직에서 처리를하고 결과가 나와야하는 메시지와는 다르다.**

**로직처리가 필요한 메시지들은 어쩔수없지만, 곧바로 결과를 전송할수있다면 send를 추가해 반응속도를 올릴 수도 있을 것이다.**

**추가로, send와 recv의 호출이 가장 느리기 때문에 수시로 링버퍼에서 메모리가 카피되는 것을 걱정하지 않아도 된다.** (send가 빈번한 서버에서는 성능이 향상될 것.)

select()..)

if(FD\_ISSET(writeSet))

char buffer[1000];

**//불필요한 copy를 줄이기위해 위 지역버퍼를 없앤다면 직접적으로 링버퍼를 건드려야하기 때문에 객체모듈화를 깨트리는 것이 된다.**

size = sendRingbuffer.**peek**(buffer,1000)

ret = send(socket, buffer,size,0);

if(ret>0)

sendRingbuffer.moveFront(ret);

**단 지역버퍼로 처음에 테스트를 한 다음, 그다음성공했다면 지역버퍼를 지울것**

size = recv(Socket, ringbuffer.GetRearPtr(), Ringbuffer, DirectEnquesize())

Ringbuffer.moveRear(size);

이때 우리의 버퍼가 끝지점에 도달했을 때(환형 큐지만 실제 끝은 존재하므로) 포인터를 다음으로 넘어가되, 이 사이즈를 구해주는함수를 DirectEnquesize로 한 것.

**이는 여유가있지만 링버퍼 포인터가 rear에 근접해있다면 rear까지밖에 받지 못할 것이므로, 이때는 다음번에 다시받는다.**

우리링버퍼는 10000바이트이므로, 메시지의 크기가 100바이트라고해도 1/100의 확률이기도하고,

**현재 고전recv가 아닌 winsock의 recv는 버퍼포인터를 두개이상 넣어서 분산처리 시켜주기때문에 때문에 지금은 이대로 진행한다.**

**링버퍼 검증**

단계단계 하나하나 라이브러리를 만들때마다 검증을 거쳐서 넘어가야한다.

**간단한 테스트 프로그램을 만들것인데, 인큐디큐를 계속반복하고 링버퍼는 작게(100~200)정도로잡으며 디큐만을 통해서 출력한다.**

인큐/디큐는 난수를 통해 반복하여 콘솔창에 맞게끔 출력하고 줄이 틀어지지않는지 검증한다.

링버퍼의 끝에 걸치는 지점에서 주로 문제가 발생하므로 이를 꼼꼼히 테스트할 것.

1,20분정도 테스트하다가 큰문제없으면 버퍼크기를 조절해서 다시 돌려본다.

오류가 발생했다면 해당값으로 돌리는건 기본. (모두 중단도 활용)

**Socket model - WSAAsyncSelet**.

**윈도우 핸들이 존재해야 하므로 클라이언트가 윈도우라는 전제하에 사용이 가능하다. 따라서 서버쪽에서는 쓸일이 없다. 멀티스레딩, 멀티코어에도 적합하지 않음.**

(실제로 빌드하면 경고가 뜨게됨) (#pragma warning(disable:4996)로 진행)

그래도 일단 AsyncSelect모델 기반으로 클라쪽 작업을 해볼것.

예전 서버에서는 활용이 됐던 모델이며, 아직까지 현역이다.(무시할 모델은 하나도없음)

**WSAAsyncSelect모델을 이용한 소켓 입출력 절차**

**1.WSAAsyncSelect 함수를 호출하여 소켓 이벤트를 알려줄 윈도우 메시지와 네트워크 이벤트를 등록한다. (ex #define WM\_SOCKET WM\_USER+1)**

**2.등록한 네트워크 이벤트가 발생하면 윈도우 메시지가 발생하여 윈도우 프로시저가 호출된다.**

**3. 윈도우 프로시저에서 받은 메시지의 종류에 따라 적절한 소켓함수를 호출하여 처리**

int WSAAsyncSelect

(

SOCKET s,

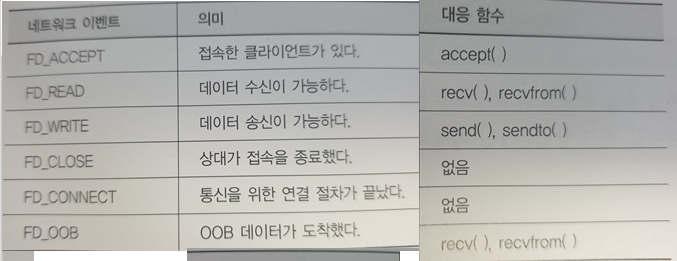
HWND hWnd,

u\_int wMsg,

//네트워크 이벤트가 발생하면 윈도우가 받을 메시지. 소켓을 위한 윈도우 메시지는 따로 정의되어있지 않으므로 WM\_USER+x 형태의 사용자 정의 메시지를 이용하면 된다.

long lEvent //관심있는 네트워크 이벤트를 비트마스크 조합으로 나타낸다.

);



연결된 클라이언트 대상으로는 다시 WSAAsyncSelect()함수에 등록한다.

클라대상이라면 FD\_CONNECT, FD\_WRITE, FD\_READ가 되어야 할것.

**WSAAsyncSelect모델사용시 주의할점**

**1.WSAAsyncSelect()함수를 호출하면 해당 소켓은 자동으로 Non-blocked모드로 전환된다.**(EventSelect도 동일하다)

**2.accept()함수가 리턴하는 소켓은 리슨 소켓과 동일한 속성을 갖게된다**. 연결 대기 소켓은 직접 데이터 송수신을 하지않으므로 FD\_READ, FD\_WRITE이벤트를 처리하지 않는다.

**반면 accept()함수가 리턴하는 소켓은 FD\_READ, FD\_WRITE이벤트를 처리해야 하므로 다시 WSAAsyncSelect()함수를 호출하여 관심있는 이벤트를 등록해야한다.**

**3.윈도우 메시지에 대응하여 소켓함수를 호출하면 대부분 성공하지만 WSAEWOULDBLOCK이 발생하는 경우가 드물게 있음. (이는 Select모델과 동일)**

**4. 윈도우 메시지를 받았을때 적절한 소켓 함수를 호출하지않으면, 다음번에는 같은 윈도우 메시지가 발생하지않는다.**

예를들어 FD\_READ이벤트가 발생했는데 recv함수를 호출하지않으면, 동일소켓에 대한 FD\_READ이벤트는 다시 발생하지않으므로 다시 발생시키고 싶다면 응용프로그램이 나중에 직접 메시지를 발생시켜야한다.

**(기존의 Select의 Polling방식과는 다른 Event방식. 변화를 알려주는것)**

**Polling방식은 재차 확인하기때문에 비효율적이고, Event방식은 변화만 알려주기 때문에 효율적이다. 단, 이 변화에 대해 적절한 대응이 없다면 영원히 반응이 없을 것.**

**FD\_WRITE는 송신버퍼에 공간이 있는 것을 확인하는 것이므로,(송신버퍼는 늘 공간이 존재)보낼수 없는상황에서 보낼수 있게 변했을때 한번 반응이 오게된다.**

FD\_WRITE가 한번이라도 반응이 온다음에는 늘 보낼수 있음.

**1. 연결에 처음 된 시점에 반응. (정상적인 상황)**

**2. 송신버퍼를 일부러 채운다음에 다시금 송신이 가능해지는 시점.**

(구체적으로는, send가 WSAEWOULDBLOCK상황에서 일정시간 후 송신버퍼가 지워지면 다시 FD\_WRITE가 반응할 것.)

따라서 AsyncSelect서는 send를 하고싶을때 하는구조.

만약 FD\_READ가 떴는데 우리가 recv를 호출하는데 우리쪽 버퍼의 크기가 작아서 프로토콜 스택의 수신버퍼를 다 빼내지 못했을 경우에는? 이때는 바로 FD\_READ가 뜰것.

그래서 이부분은 걱정할 필요없음!

LRESULT WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

**message : WSAAsyncSelect()함수 호출 시 등록했던 사용자 정의 메시지이다.**

**wParam : 네트워크 이벤트가 발생한 소켓이다. 이값을 SOCKET으로 형변환하여 소켓함수 호출에 그대로 사용하면 된다.**

**lParam : 하위 16비트는 발생한 네트워크 이벤트, 상위 16비트는 오류 코드를 담고있다**. (항상 오류코드를 먼저 확인해야 할 것)

**이때 WSAGetLastError()함수로 오류코드를 알아낼수 없으므로**

**WSAGETSELECTERROR(lParam) HIWORD(lParam)**

**WSAGETSELECTEVENT(lParam) LOWORD(lParam)**

위 매크로를 사용한다.

예제진행:386P

WinProc에서는 제일먼저 WSAGETSELECTERROR를 확인하여 에러가있다면 소켓(wParam)을 끊을것.

AsyncSelect를 쓴다고한다면 Event가 모두 소켓기준이다.

FD\_READ나 FD\_WRITE같은 이벤트는 동시에 발생되어도 낱개로 하나씩온다.

FD\_ACCEPT에서,그리고 방금접속한 클라를 대상으로 다시 AsyncSelect를 등록함.

**AsyncSelect구조 – Client Part**

기본적으로 SendQ와 RecvQ가 필요할것.

bool g\_Connect = false;

bool g\_Send = false;

main()

{

윈도우 생성

소켓생성

WSAAsyncSelect(Socket, hWnd, UM\_NETWORK, FD\_CONNECT | FD\_CLOSE | FD\_READ | FD\_WRITE);

ret = connect(...) //논블락이므로 거의 100%에러

if(ret == SOCKET\_ERROR)

if(WSAGetLastError() != WSAEWOULDBLOCK)

에러상황

while(PeekMessage(...))

{

윈도우 메시지 루프..

else

{

GameUpdate(); <ㅡ이게 호출됐어도 아직 연결여부는 모른다.

{

if(g\_Connect == false) **FD\_CONNECT가 뜨기전에는 게임로직이 돌아서는 안된다.**

연결 안된상태..

}

}

}

}

WndProc()

{

case UM\_NETWORK;

if(WSAGETSELECTERROR(lParam))

{

에러처리, 연결종료

}

switch(WSAGETEVENTSELECT(lParam))

{

case FD\_CONNECT:

g\_Connect = TRUE; (연결됐다는 알림을 추가해도 좋음.)

case FD\_READ

ProcRead(); break;

case FD\_WRITE;

g\_Send = true; //FD\_WRITE가 호출되었으므로 Send플래그를 바꿔준다.

ProcWrite()

}

//신호가왔을때 Send를 모두 보내야하므로, 다음과같은 반복문을 돌게된다.

while(1) //1234

{

if(SendQ.GetUseSize() <= 0) 또는 WOULDBLOCK이 떴다라면, 중지.

break;

}

case FD\_CLOSE:

}

void SendPacekt(buffer,size)

void ProcRead();

void ProcWrite(); //FD\_WRITE가 떴을때

void SendPacket(buffer, size)

**//클라이언트 대상이니 소켓도 필요없겠고.**

**//이 함수는 FD\_READ나 FD\_WRITE에서 호출하는 함수가 아니라, 우리가 send하고싶을때마다 호출하게되는 함수.**

{

ret = SendQ.Enque(buffer, size); **//성능을 조금 포기하고 절차대로, 맨처음부터 Enque한다.**

**//SendQ에 있던 것들을 send..**

**//이번에 보낼 buffer send..**

**//이경우 예외처리들이 너무 난잡해지기 때문에, 성능을 조금 포기하더라도 절차를 정하고간다.**

**//send의 에러상황은 WOULDBLOCK과 우리가 요청한 사이즈 이하가 나오는것.**

if(ret != size) **//send큐에 공간이없는것으로, 이경우 개발자는 방법이없음.**

{

**//공간이 다찼으면 연결이 끊는다. (로그 남기기)**

**//클라에선 가능성 거의 x**

}

**//뭔가 보낼게 있어서 SendPacket이 호출되었다. send를 안할거라면 Peek도 필요없음. 여기서 만약 send플래그가 false라면 나가버린다.(\*2)**

**if(g\_bSend == false)**

**return;**

char sendbuffer[5000];

sendsize = SendQ.Peek(SendBuffer, 5000);

ret = send(Socket, buffer, size, 0); //에러가 나지않았다면 뭔가 보내졌다는 얘기.

if(ret == SOCKET\_ERROR)

{

DWORD err = WSAGetLastError();

**//WSAGetLastError()는 소켓전용함수가 아니라 GetLastError()와 똑같다. 에러가 난이후 printf와같은 다른 함수를 사용한다면 이 결과는 덮어씌워져서 사라질것.**

**if** (err != WSAEWOULDBLOCK)

{

종료처리

//함수를 만들어서 로그를 남길만한 가치가있음. 마냥 버릴것이아님

**//이부분 그냥 넘어가도 상관없지만, WOULDBLOCK이 뜬다음에는 아무리 send를 시도해도 FD\_WRITE가 뜨기전까지 계속 WOULDBOLCK상태이다. 따라서 send플래그를 넣어서, 판단한다. (\*2)**

}

//만약 Enque는 했는데 못보내는 상황이 만들어졌다면? 이건 누가 보내냐?

// Polling방식인 Select에서는 재차 확인하여 보내므로 큐에만 쌓아놓고 빠지면됐었다.

**Event방식인 AsyncSelect에서는 지금 보내지 못한다면 FD\_WRITE를 기다려야만 한다.**

**따라서 FD\_WRITE가 떴을때 SendQ에 있는것을 모두 뽑아 보내야한다. 그렇지 않으면 아무도안보냄.**

//연결 종료는 아닌데 send는 못하는 상황. 보관해야한다. (SendQ로)

//SendQ.Enque(buffer, size);

**//SendQ에 있는 것을 먼저보내고 못보냈으면 SendQ에 넣는 절차가 되야한다.**

**else**

{

g\_Send = false;

return;

}

}

**SendQ.RemoveFront(ret);** **//Peek로 보냈으므로 Front를 이동시켜 지운다.**

SendPacket도 ProcWrite를 호출시키는 것으로 끝날수있다는 말이된다.

}

void ProcWrite() – FD\_WRITE가 떴을때 할 행동.

{

//이후 Send큐에 있는 것을 보내줘야 하는데 SenPacket과 같다. (복붙)

char sendbuffer[5000];

sendsize = SendQ.Peek(SendBuffer, 5000);

ret = send(Socket, buffer, size, 0);

if(ret == SOCKET\_ERROR)

{

DWORD err = WSAGetLastError(); //함수를 만들어서 로그를 남길만한 가치가있음

if(err != WSAEWOULDBLOCK)

{

종료처리

}

else

{

g\_Send = false;

return;

}

}

SendQ.RemoveFront(ret);

}

void ProcRead()

{

char RecvBuff[5000];

ret = recv(socket, recvbuff,5000,0);

if(예외처리)...

무조건 1번 리시브받고 완성된건 다뽑아내자는게 규칙이기떄문에 RecvQ에 남아있는 것은 완성되지않은 하나이하의 메시지 데이터일것.

만약 두개이상의 메시지가 들어있다면 이는 잘못만들어진 상황.

이런 상황이 아닌데도 RecvQ가 다찼다면 이는 누군가 의도적으로 이상한 메시지를 보낸것

while(1)

{

헤더 peek

헤더기준으로 완성된 패킷 있는지 사이즈 확인.

RecvQ.Deque(PacketBuffer, PacketSize);

메시지 프로시저 호출

}

}

**그림판 예제**

PORT : 25000

// 헤더

unsigned short Len;

// 패킷

struct st\_DRAW\_PACKET

{

int iStartX;

int iStartY;

int iEndX;

int iEndY;

}

2 byte – 패킷길이

4 byte - Start X

4 byte - Start Y

4 byte - End X

4 byte - End Y

패킷의 총 사이즈는 18바이트

# 자기 자신의 그리기도 서버로 패킷을 받아서 그림

WM\_MOUSEMOVE:

OldX OldY CurX CurY

stDraw\_PACKET packet;

packet.StartX = OldX

packet.StartY = OldY

packet.EndX = CurX

packet.EndY = CurY

SendPacket(&Packet, sizeof(Packet));

위구조에서는 구조체에 헤더를 넣지않았다. 구조체에 헤더까지 포함해야하지만, 헤더는 헤더고 Payload는 Payload여야한다. 여기 Len이라는 것은 원칙적으로 컨텐츠의 헤더가 아니라 네트워크의 헤더이다.

일단은 넣은다음 SendPacket안쪽에서 헤더를 만드는것. 컨텐츠에서는 헤더를 신경쓰지않으나 네트워크의 SendPacket에서 헤더를 만들어 주는것.

이경우 SendPacket의 모양이 바뀔것.,

바뀐 SendPacket(Packet, size);

{

unsigned short Header = size;

SendQ.Enque(&Header, 2);

SendQ.Enque(Packet, size);

//큐를 두번하지만 일단은 넘어가자.

//네트워크 라이브러리를 만들때는 Packet이라는 객체를 만들어 Packet안에 헤더의 공간이 미리 확보가 되있어야할것.

Send가 바뀌었으므로 Recv도 바뀌어야할것.

ProcWirte();

while(1)

{

if(RecvQ.GetUseSize() <= 2)

break;

//지금상황에서 헤더만 오는 패킷은 없다. 아직 완성되지않은것임. 나중에는 프로토콜이 헤더만 오는 경우도있을수있음. 근데 2byte len에 상황에서 헤더만 오는건 말이안되는상황 지금은 만약에 2보다 작으면 할게없음

RecvQ.Peek(&header,2); //예외는 생략

if(RecvQ.GetUseSize() < header + 2(??)) break;

헤더기준으로 완성된 패킷(메시지)있는지 사이즈확인

RecvQ.Movefront(2);

RecvQ.Deque(PacketBuffer, PacketSize);

메시지를 받은 다음에는 Type에따라 나눠 메시지를 처리해야 할 것이지만, 여기서는 Type이 없으므로 곧바로 그리기함수로 들어간다..

DrawLine(Packet.StartX, Packet.StartY,packetEndX,packetendY);

메시지 프로시저 호출

}

}

DRawLine(SX, SY, EX, EY)

{

GetDC..

}

예제가 성공했다면 지역버퍼를 지우고 링버퍼를 테스트 해본다.

**AsyncSelect구조 – Dummy Client Part**

#pragma comment(lib, "ws2\_32.lib")

#include <iostream>

#include <winsock2.h>

#include <ws2tcpip.h>

#include <tchar.h>

#include <time.h>

#define IP "1.236.10.181"

struct DRAW\_PACKET

{

int StartX;

int StartY;

int EndX;

int EndY;

};

void Error(DWORD err)

{

printf("error - %d\n", err);

exit(-1); //쓰면안되지만 테스트용으로 잠깐..

}

int main()

{

srand(time(NULL));

WSADATA wsa;

if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0)

return -1;

//이부분에서 소켓을 배열로만들어 계속 쏴준다.

SOCKET Socket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

if (Socket == INVALID\_SOCKET)

Error(WSAGetLastError());

SOCKADDR\_IN addr\_in;

memset(&addr\_in, 0, sizeof(addr\_in));

addr\_in.sin\_family = AF\_INET;

addr\_in.sin\_port = htons(25000);

InetPton(AF\_INET, \_T(IP), &addr\_in.sin\_addr);

if (SOCKET\_ERROR == connect(Socket, (SOCKADDR\*)&addr\_in, sizeof(addr\_in)))

Error(WSAGetLastError());

while (true)

{

DRAW\_PACKET Packet;

USHORT header = sizeof(Packet);

Packet.StartX = rand() % 500;

Packet.StartY = rand() % 500;

Packet.EndX = rand() % 500;

Packet.EndY = rand() % 500;

if (SOCKET\_ERROR == send(Socket, (char\*)&header, 2, 0))

Error(WSAGetLastError());

if (SOCKET\_ERROR == send(Socket, (char\*)&Packet, sizeof(Packet), 0))

Error(WSAGetLastError());

Sleep(5);

}

}

**AsyncSelect구조 – Server Part**

SOCKET g\_ListenSocket;

Session

{

SOCKET Socket;

Addrinfo;

RecvQ;

SendQ;

SendFlag; //클라에서 우드블럭이 뜨면 세션불가능이라고 나왔음. 그거랑같은얘기

}

SessionList로 관리

main()

{

윈도우생성

ListenSocket생성, bind, listen

WSAAsyncSelect(g\_listenSocket, hWnd, UM\_NETWORK, FD\_ACCEPT | FD\_CLOSE);

}

while (GetMessage)

{

case UM\_NETWORK:

{

if (WSAGETSELECTERROR(lParam))

**//(모든 클라와 Listen소켓의 에러가 모두 이쪽으로 들어올것.)**

{

**//3WH은 TCP가 할일로, accept와는 무관하다. 따라서 connect가 실패하는것과는 전혀관련없음**

// 리슨소켓은 에러가날 확률은 0에 가깝다. ( 의도적으로 close하지않는이상

**// 정상적인 종료보다는 비정상적 종료(에러)가 더많다. 따라서 연결을 끊을것**

**wParam으로 소켓이 들어오는데,이걸끊는다.세션리스트로 항상 검색해야함.**

Session\* p = FindSession((SOCKET)wParam);

p->socket;

}

switch (WSAGETSELECTEVENT(lPARAM))

case FD\_ACCEPT:

ProcAccept(); break;

case FD\_READ:

여기도 찾아서 가도됨.

ProcRead(wParam); break;

case FD\_WRITE:

pSession = FindSession(wParam);

ProcWrite(pSession); break;

//ProcWrite(wParam); break; 세션이 훨씬 편해지기 때문에 변경!

case FD\_CLOSE:

RemoveSession(wParam); break;

}

}

Session\* FindSession(SOCKET ClientSocket)

{

}

void RemoveSession(SOCKET ClientSocket)

{

**SessionList에서 ClientSocket검색 후 closesocket(pSession->Socket);**

**세션에서 list에서 삭제.**

//작업관리자에 소켓이 핸들로 포함되기 때문에 이를 확인할수 있을것.

검색실패했따면 return null < 말이안되는상황 리슨소켓이 들어온게 아니라면

접속자에 대한 로그를 출력

/\*

이때, 바깥에서 list를 돌면서 조건에 의해 삭제되는 구조이므로, 여기서 iterator를 제해버린다면 바깥에서 꼬일수있음. 삭제한 것을 또 broadcast해야 하는 구조라면 복잡하겠지만 여기서는 포인터를 받아놓고 iter를 ++해서 sendpacket(다음로직)으로 들어가도 상관없음.

이 구조가 마음에 들지 않는다면 여기선 직접적으로 삭제를 하지않고 별도의 플래그를 둬서 체크하고 빠진다음 별도의 루프에서 삭제하는것.

여기서 이벤트방식으로 간다면 별도의 네트워크 메시지를 만드는 것도 괜찮을것. (실제 세션을 삭제하는.) 이게 성능이 더 좋은 구조는 아님.

\*/

}

void ProcAccept()

{

Socket = accept(...); **//혹시 모를 WOULDBLOCK확인**

Session 동적할당

Session->Socket = Socket;

Session->AddrInfo = 셋팅

SessionList추가

WSAAsyncSelect(SOcket, g\_hWnd, UM\_NETWORK, FD\_READ | FD\_WRITE | FD\_CLOSE);

접속자에 대한 로그를 출력

}

ProcRead(ClientSocket)

{

Session\* p = FindSessioon(ClientSocket);

if (p == null)

{

서버를 죽이고 판단해야 하는 중대한 상황이다.

절대나오면 안되는상황

이같은 상황에서 대충수습하고 넘어갈경우 회사가 망할수있음

}

ret = recv(p->Socket, ....p->RecvQ로 수신)

if (에러 || ret == 0)

err

while (1)

{

해당세션의 RecvQ에 들어있는 완성된 메시지 확인

DRAW\_PACKET Packet;

RecvQ.Deque(&Packet, header); //헤더에는 16바이트

SendBroadCast(&Packet); 자기자신도 보냄

}

해당세션의 RecvQ에 들어있는 완성된 모든 메시지를 처리.

}

**/\***

**실제로 이 구조라면 완성된 메시지를 확인할 필요가 없다. 메시지 단위로 2byte, 16byte단위로 뽑아서 보내지않고 4byte만 온것을 broadcasting해도 딱히 상관없는 구조.**

**클라-서버가 똑같은 프로토콜과 메시지를 사용하고있고 서버는 하는일없이 뿌려주기만 하면 끝나기 때문이다.**

**따라서 Recv하면 곧바로 broadCast하는 방식으로 진행한다.**

**\*/**

SendBroadCast(\*Packet)

{

SessionList순회->\*p;

//별도의 RemoveSession을 등록.

//아니면 RemoveSession에서 플래그만체크

//WndProc에서 case UM\_REMOVESESSION: 세션삭제(Socket); 할수도있음.

SendPacket(p, Packet);

}

SendPacket(Session\*, Packet\*)

{

Session->SendQ.Enque(Packet, 16);

if (Session->SendFlag == false)

return;

ProcWrite(Session);

}

ProcWrite(Session)

{

while (1)

{

ret = Send(Session->Socket, Session->SendQ열심히 샌드)

//지금아니면 보낼기회가 없을지도 모르기 때문에 최대한 보내는것.

if (ret || WOULDBLOCK이 아니면)

RemoveSession(...);

Session->SenQ를 모두 보내면 반복문 중단

}

}

**Socket model - EventSelet**.

가볍게 살펴볼 모델

**Socket model – Overlapped I/O**

윈도우 비동기입출력을 사용한 모델이다.

**Socket model – Completion port**

OverLapped모델에서 발전된 모델로, IOCP를 뜻한다. 우리가 최종적으로 사용하는 모델이될것.

(overlap과 completion port는 몇 달뒤 진행한다.)