**<7.10> - 2**

**Overlapped Model**

**Overlapped 이라는 표현은 MS가 비동기 입출력의 이름으로 정한 이름이다.**

**Overlapped은 Select, AsycSelect, EventSelect과 근본적으로 다르다.**

((

**EventSelect는 소켓하나당 Event를 등록시켜 소켓으로 signal이 오면, Waitfor..함수를 이용하여 signal을 확인하여 작업한다.**

EventSelect는 여타 다른 Select모델과 마찬가지로 소켓이 64개로 제한되어있으므로 서버쪽에서 쓸일이 없으며, 클라쪽에서는 EventSelect를 권장한다.

(AsycSelect의 경우 Window Message으로, main thread기반(싱글 스레드)으로 돌수밖에 없으므로 MS에서도 권장하지않음.

))

**원래 Overlapped 입출력 방식은 윈도우 운영체제에서 고성능 파일 입출력을 위해 제공하는데, 이를 소켓 입출력에도 사용할수 있게 만든 것이 Overlapped소켓 입출력모델이다.**

파일을 읽고 쓰기할때도 Readfile(), Writefile() 을 통해 비동기가 가능하다.

지금껏 우리는 파일입출력을 CRL함수로 했는데, Overlapped I/O는 표준런타임 라이브러리나 C++에 존재하지않으므로 WINAPI를 직접 사용해야 한다.

**동기의 경우 입출력 요청시 커널쪽에서 작업하고 리턴되면 해당 작업의 결과로 사이즈를 알려줬으나, 비동기의 경우 요청만 하고 빠져나온뒤 커널쪽 백그라운드에서 별도의 작업이 들어간다.**

이후 완료통지는 어떤 방법을 통해서든 우리에게 알려줄 것.

이때 Overlapped모델을 책에서는 **IOCP와 Completion Routine**으로 나눈다. 할일이 없어지는 것이 아니기 때문에 CPU사용률은 큰 차이가 없을 것이다.

**단 동기화의 경우 기다려야하지만 비동기의 경우는 작업하는동안 다른일을 하다가 일이 완료되면신호를 받아서 마무리 작업을 하는 점이 다르다.**

(결과만 알려주기 때문에 ProActor패턴)

근데 동기같은경우는 기다려야하지만 비동기의 경우는 다른일을 할수있음. 요청했던게 완료되면 신호를 받아서 마무리작업을 하는 방식.

**Select, WSAAsyncSelect, WSAEventSelect는 동기 입출력이며,**

**(Signal을 주든 Queue에 넣든 수시로 falling해야 하는 것이 치명적인 단점.)**

**Overlapped, Comletion port은 모두 비동기 입출력이다.**

**Overlapped모델에서의 공통절차**

**1. 비동기 입출력을 지원해주는 socket()함수로 소켓생성.**

기본으로 제공되는 버클리소켓 함수를 사용한다.

Winsock에서 제공하는 WSAESocket() 함수의 경우는 옵션을 따로 지정해야만 OverlappedIO를 지원하기때문에 굳이 사용할 필요가 없음.

**2. 비동기 입출력을 지원하는 소켓함수를 지원한다.**

AcceptEx(), ConnectEx(), DisconnectEx(), TransmitFile(),

TransmitPackets(), WSAIoctl(), WSANSPIoctl(), WSAProviderConfigChange(),

**WSARecv(), WSARecvFrom(),**

WSARecvMsg(), **WSASend(), WSASendTo()**

**여기서 AcceptEx()는 사용하지않는다.**

accept()의 경우 backlog Queue에서 소켓을 만들어 연결시킨다음 리턴되는 구조이다.

AcceptEx()는 미리 소켓을 준비하여 등록시켜놓는다. accept절차 자체도 비동기로 가겠다는 뜻으로, backlogQ에 들어오는 즉시 뽑아서 바로 완료통지하는 형태이다.

Acceptex를 쓸때, 한단계 더들어가면 TreansmitFile()이란 함수가 나오는데, 이는 소켓조차 재활용하는 방식을 통해 closesocket을 하지않겠다는 의미이다.

**AcceptEx()를 사용한다면 accept보다 1.5배정도의 성능향상을 볼것이지만, 구조가 복잡해지고 accept를 고성능으로 낼 가치가 없기 때문에 사용하지않는다.**

**3. 운영제체가 IO작업완료를 알려주면(=비동기 통지), 응용프로그램은 입출력 결과를 처리한다.**

**Synchronous <-> asynchronous / Block <-> Non Block**

런타임 라이브러리의 block함수로는 키보드입력처리가 있고, (scanf, gets….) Win32프로젝트에서의GetMessage(block), PeekMassage(Non-block)는 모두 동기함수이다.

**비동기는 즉시 리턴 후 백그라운드에서 작업하고,**

**동기는 함수가 리턴될 때 작업이 끝나고 리턴된다.**

**이와 별개로 block은 해당일을 할 수 없을 때 블락여부를 말한다.**

**블락과 동기는 전혀 다른 개념이다.**

(당연히 비동기가되려면 논블락이다.)

**Overlapped모델의 종류 두가지**

**1. 소켓입출력 작업이 완료되면 운영체제는 응용프로그램이 등록해둔 이벤트객체를 신호 상태를 바꾸고, 응용프로그램은 이벤트 객체를 관찰함으로서 입출력 작업완료를 감지할 수 있다.**

**2. 소켓입출력 작업이 완료되면 운영체제는 응용프로그램이 등록해둔 함수를 자동으로 호출한다. 일반적으로 운영체제가 호출하는 응용프로그램함수를 콜백함수라고 하는데, 특별히 Overlapped모델에서는 완료 루틴(completion routine)이라 부른다.**

**(함수를 포인터형태 인자로 넣으면 내부 커널에서 자동으로 호출이 될 것)**

WinProc같은 경우는 WinMain의 DispatcherMessage가 호출한다. 이는 결국 우리스레드에서 호출호출하는 것, 기회를 줘야 호출한다.

맘대로 커널이 하는 것은 존재할수 없음.

**때문에 Completion routine의 Callback방법도 제한이 있다는 단점이 있다.**

**WSASend**

int WSAAPI WSASend(

SOCKET s,

LPWSABUF lpBuffers,

DWORD dwBufferCount,

LPDWORD lpNumberOfBytesSent,

DWORD dwFlags,

LPWSAOVERLAPPED lpOverlapped,

LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionRoutine

);

**LPWSABUF lpBuffers,**

버퍼 포인터와 길이를 요소로 가진 구조체를 포인터형태로 받고있다. **이는 여기저기 흩어진 메모리 버퍼들을 하나로 등록할 수 있다는 뜻이다.**

우리가 링버퍼를 사용해서 보낼 때 버퍼의끝에 걸리는경우가 존재했는데, 이제는 끝에 걸리는 경우에도 원하는만큼 데이터를 보낼수 있게 되었다.

n개의 버퍼를 보내기위해 send를 n번하는것보다 한번에 모아서 send하는 것이 훨씬 빠를 것.

**이는 하나의 버퍼만 사용할때랑 비교한다면 당연히 성능이 떨어지겠으나, 버퍼 개수를 여러 개 넣는다고 하여 버퍼개수에 비례하게 성능이 떨어지지 않는다.**

Scatter(흩어지다)/Gather(모으다)기능이라고 말하기도한다.

**LPDWORD lpNumberOfBytesSent,**

실제 Send행위가 성공한 크기(얼마만큼을 카피했는지)를 포인터값으로 뱉어준다.

(리턴값으로 뭔가를 알려준다는 것은 옛날 방식)

**단, 이는 동기입출력일 경우이며, 비동기입출력일 경우 아무값도 뱉어주지않는다.**

**DWORD dwFlags,**

OOB(Out-Of-Band: 긴급데이터 통신)를 뜻하며, 의미없으므로 항상 NULL이 들어간다.

**LPWSAOVERLAPPED lpOverlapped,**

OverlappedIO의 가장 필수적인 요소이다.

**LPWSAOVERLAPPED와 LPWSAOVERLAPPED\_COMPLETION\_ROUTINE인자에 둘다 NULL이 들어갈 경우 OverlappedI/O가 아닌 일반적인 Send/Recv처럼 사용이 가능하다.**

**해당 인자는 OverlappedIO의 flag역할을 한다.**

**LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionRoutine**

Callback방식을 사용할때만 셋팅한다. 사용하지않을 경우 **NULL**

**COMPLETION\_ROUTINE이 OVERLAPPED보다 우선순위가 높기 때문에 여기에NULL을 전달하지 않는다면 WSAOVERLAPPED구조체의 hEvent변수는 사용되지않는다.**

**WSABUF**

typedef struct \_WSABUF {

ULONG len;

CHAR \*buf;

} WSABUF, \*LPWSABUF;

길이와 버퍼 포인터가 들어간다. Send나 Recv에서 여러 개의 버퍼를 copy하고 싶다면 배열로 만들어 셋팅한다.

WSABUF는 인자전달의 버퍼 포인터일 뿐이므로, 보관시키거나 공유할 대상이 아니므로 일회성으로 사용하고 버린다.

(책에서는 WSABuf를 세션 멤버에 보관을 시키는데 보관시킬 대상이 아니다.)

리턴되는순간 제 역할을 다한 것.

**WSAOVERLAPPED**

typedef struct \_WSAOVERLAPPED {

DWORD Internal;

DWORD InternalHigh;

DWORD Offset;

DWORD OffsetHigh;

WSAEVENT hEvent;

} WSAOVERLAPPED, \*LPWSAOVERLAPPED;

WSAOVERLAPPED구조체는 비동기 입출력을 위한 정보를 운영체제에 전달하거나, 운영체제 비동기 입출력 결과를 응용프로그램에 알려줄 때 사용하는 **작업의 단위**이다.

소켓단위가 아닌 작업의 단위이므로 하나의 소켓으로 Send/Recv를 같이 한다면 WSAOVERLAPPED는 두개가 존재해야 한다.

**완료 통지가 오기전까지 WSAOVERLAPPED구조체는 변경을 가하거나 삭제해서는 안된다.**

**(많이 실수하는 부분이므로 반드시 잘 알아둘 것)**

**1~4번째 인자까지는 모두 운영체제가 내부적으로 사용하는것이고, 입출력이 완료되었을 때 hEvnet가 가리키는 이벤트 객체는 signal상태가 될것이다.**

((

윈도우 시스템프로그래밍 책에 파일IO부분에서도 OVERLAPPED구조체가 등장한다.

(WSAOVERLAPPED와 OVERLAPPED는 같은구조로, 모양새가 다른것처럼 보이지만 완전히 일치하는 모양새이다. )

DWORD Offset;

DWORD OffsetHigh;

**만약에 OVERLAPPEDIO로 파일쓰기를 한다면 직접 offset에 파일포지션을 전달해야 한다.**

우리가 지금까지 한 파일입출력은 자동으로 넘어가는 형태였으나, OVERLAPPEDIO를 사용하고자한다면 직접 Offset을 넣어야 한다.

**Offset을 0으로 지정하여 CreateFile로 연다음 WriteFile을 OVERLAPPEDIO로 연속해서 두번쓰기를 한다면 같은위치에 두개의 작업이 동시에 들어가게된다.**

**때문에 파일에 대한 쓰기읽기 할때는 반드시 Offset을 직접계산해서 overlapIO를 요청해야한다.**

**하지만 WSAOVERLAPPED는(소켓쪽에서는) offset을 같은 값으로 하더라도 덮어쓰지 않으므로 오프셋을 사용하지않는다.**

인터넷 자료중에 DWORD Internal 과 DWORD InternalHigh 을 사용하여 에러코드를 찾을수있다는 설명이 나오기도 하는데, 다른방법이 많이 존재하므로 사용하지않는다.

**결론적으로 1~4번째 인자는 단순히 초기화만 한다.**

))

**Overlapped Model - Evenet**

event방식은 쓰기가 번잡스럽기 때문에 잘 사용하지않는 방법이다.

**1. 먼저 비동기 입출력을 생성하고, WSACreateEvent()를 통해 대응하는 이벤트 객체를 생성.**

**(소켓당 이벤트 하나가 되어야 한다.)**

**2.비동기 입출력을 지원하는 소켓함수를 호출한다.(WSASend/ WSARecv)**

이때 인자로 들어갈 WSAOVERLAPPED구조체의 hEvent에 Event핸들값을 전달한다.

비동기 입출력 작업이 곧바로 수행완료 되지않는 경우 소켓함수는 오류를 리턴한다.

**WSAGetLastErorr()의 오류코드가 WSA\_IO\_PENDING인경우 OverlappedIO가 성공적으로 수행됐음을 의미한다.**

작업이 완료되면 전달한 event를 통해 signal을 줄것이다.

**3. WSAWaitForMultipleEvents()로 기다린다.**

4. 작업이 완료되어 signal이 온경우 WSAGetOverlappedResult()함수를 호출해 비동기 입출력 결과를 확인하고 데이터를 처리한다.

OVERLAPPEDIO로 작동이되서 일을 끝마쳤으나 우리는 Signal만이 전달되었기 때문에 얼마나 보냈는지 알아야 한다. 따라서 위 함수를 호출시켜 결과를 확인한다.

(Send인지 Recv인지는 우리가 이벤트객체로 구분한다.)

**WSAGetOverlappedResult**

BOOL WSAAPI WSAGetOverlappedResult(

SOCKET s,

LPWSAOVERLAPPED lpOverlapped,

LPDWORD lpcbTransfer,

BOOL fWait,

LPDWORD lpdwFlags

);

LPDWORD lpcbTransfer,

얼마만큼의 작업이됐는지 알려준다.

BOOL fWait,

작업이 완료되지않았을 때 블락되서 기다릴 것인지에 대한 여부.

우리가 WSAGetOverlapresult()함수를 호출했을 때 내부에서 작업이 완료되지 않았을 수 있으므로, 이에대한 상황을 파악하기 위해서 존재한다.

LPDWORD lpdwFlags

사용하지않는다.

**433P샘플예제**

책에 있는 예제는 Full Duplex가 아닌 Half Duplex구조를 취하고있다.

**따라서 send할때는 send만을하고, recv할때는 recv만을 하기 때문에 WSAOVERLAPPED구조체를 하나만 사용하여 singal을 받고있다.**

**일반적으로 OverlappedIO를 가장먼저 학습할때는 Event객체로 결과를 뽑아야 하므로 직접 관리보관해야한다.**

**IOCP같은 경우 어떤 Overlapped구조체 포인터가 끝났는지 알려주는 구조이기 때문에, 동적할당하여 포인터를 넣고 완료되면 해제하는 방식으로 사용하는 방식으로 써먹어도 상관없다.**

**언급했듯 WSABUF는 전역으로 보관할 이유가 없으므로 지역으로 두는 것을 권장한다.**

Event방식에서는 어떤소켓에 반응이 왔는지를 알아야 하기 때문에, 해당예제에서는 소켓배열과 이베트 배열을 만들어놓고 Index를 일치시켜 구분한다.

**일반적으로 accept는 별도의 스레드에서 처리한다.**

**listenSocket은 블락소켓으로 둔다.**

**모든스레드는 할일이 있으면 깨어나서 일을하고 없으면 쉬어야 하기 떄문에, 백로그큐로부터 소켓이 오지않는다면 일이 없는것이므로 그냥 블락당해서 쉰다.**

**OverlappedIO를 사용하는 소켓모델일 경우 기본적으로 accept하자마자 WSARecv를 걸고간다.(항시 걸려있어야 하기 때문)**

(우리는 지금까지 사용한 소켓모델들은 소켓함수가 존재했고 해당 함수를 통해 소켓에 대한 상황을 파악했었다. 하지만 OverlappedIO는 단지 비동기 입출력일뿐이다.)

Select모델인 경우 Recv가 왔는지 파악하기위해 Select함수를 호출했으나, OverlappedIO는 Recv를 걸고 반응을 확인하는 방식.

**물론 Recv에 반응이와서 데이터를 받은다음에 다시 Recv를 걸어놔야한다.**

**이제 WSAEWOULDBLOCK은 사용하지않는다.**

AsyncSelct 이후의(포함) 모델들은 모두 Non-block이기 때문에 이를 신경쓰지않는다.

Accept에 반응이왔다면 WSASetEvent로 Signal을 보내고, 다시 accept로 올라가서 block걸린다.

**WorkerThread는 루프를 돌면서 WSAWaitForMultipleEvent로 반응을 기다린다.**

(맨마지막 인자인 flag는 이후에 언급된다.)

멈춰있는 상황에서 accept스레드에서 새로운 사용자를 받아들이고, EventArr 뒤에 추가한다. 블락들어간 상태이므로, 방금들어온 사용자를 어떻게 할 방법이 없음.

따라서 이를 반영하기위해 TotalSocket의 개수가 변경되게 반영되어야하고, EventArr뒤로 받아줘야한다. 그래서 0번째 더미이벤트를 사용여 accept쪽에 새로운것이 들어왔다는 것을 알리는 목적으로 사용한다.

0번째 이벤트에 Signal 오면 일단 waitforMultipleEvent는 깨어나서 다시 셋팅해서 들어간다. 오직 그목적으로 사용된다.

(0번이 아닌 다른애들은 실제 소켓용도로 사용하는것이다.)

WaitForMultpleEvent로 확인하여, WAIT\_FAILED는 문제있는상황으로 처리.

WSA\_WAIT\_EVENT\_0은 해당 배열의 signal을 다시 Non-signal로 바꾼다.

만약에 Index가 0일경우 continue를 통해 다시 루프위로 올라간다. (0번째는 메인스레드에서 signal을 주는것이기 때문에 새로운애가 들어왔다는 뜻)

다시 올라가서 토탈소켓 개수를 반영, 이벤트 반영.

WSACreateEvent의 경우 ManualReset이 고정이다.

WSAGetOVerLappedResult 지금예제에서는 오버랩이 하나밖에 존재하지않으므로 바로 때려본다. 만약에 OVERLAPPED구조체가 두개인경우 Send/Recv 구분해야한다.

자그래서 일단 Transferred를 확인하여 0일 경우 Recv 0,에러인경우 종료 처리.

**Send 0은 상관없으나 Recv가 0인경우 종료되므로 주의해야 하는부분.**

EventSelect는 사용하지않으므로, 느낌만 기억하고 넘어간다.

위와같은 Event형태는 사용하지않으므로 느낌만 기억해둘것 .

**<7.13>**

Event방식의 OverlappedIO는 서버에 적용하기에는 애매한 모델이다.

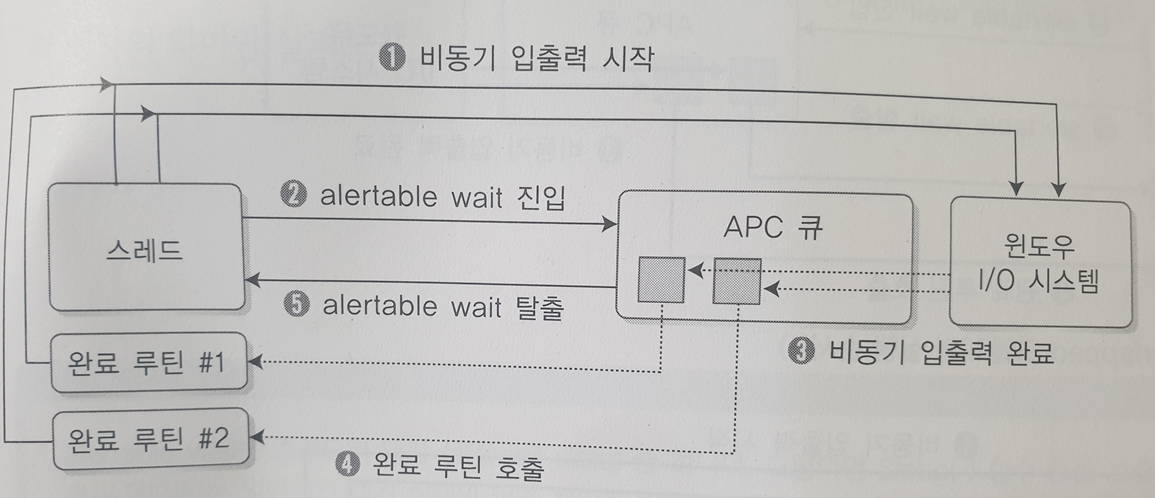
**EVent객체를 Send,Recv로 나눠 추가적으로 관리해야하고 Event Signal왔다고 해도 누구껀지 알수가 없으며,** (그렇다고 모든 소켓당 Event를 만들수도 없음)

**또한 WaitforMultipleEvent는 소켓이 64개 제한되어있기 때문에 사용하기 애매하다.**

(64개씩 스레드를 끊어서 돌리는것도 이상한 구조..)

**Completion Routine**

**Completion rountine은 운영체제가 적절한 시점에 자동으로 호출하는 사용자 정의 함수를 뜻한다**.



어떤 스레드에 작업을 요청한 상황일 때, (ex 파일IO든 소켓IO든) 비동기 입출력은 윈도우I/O시스템에서 백그라운드에서 별도로 돌아간다. (이것에 대한 스레드는 우리가 제어할수없음.)

이후 뭔가 작업이 완료되면 APC큐에다가 넣어주고, APC큐에 들어온것들을 확인하면서 콜백함수를 호출시킨다.

**이때 콜백함수를 등록하고 윈도우가 임의로 이 함수를 호출시키는 것은 말이되지않음**.

**완료통지에 대한 함수가 호출이 되기위해서는 내 스레드 안에서 호출이 되어야 한다.**

(IOCP같은 경우는 이 모든문제 해결한 모델.)

**IO라는단위는 관리자체가 thread이다. File read/write, Socket send/recv 와 같은 IO작업들은 모두 thread에서 관리한다.**

(입출력에 대한 I/O RequstPackt이라는 목록이 스레드에 붙어 내부에서 관리된다.)

((

**I/O RequestPacket**

I/O 요청 패킷은 Windows 드라이버 모델 및 Windows NT 장치 드라이버가 서로 통신하고 운영 체제와 통신하는 데 사용되는 커널 모드 구조이다.

이는 I/O 요청을 설명하는 데이터 구조이며, "I/O 요청 디스크립터"또는 이와 유사한것으로 간주할 수 있다.

))

**따라서 A스레드에 IO를 요청한 것은 A스레드에서 관리당하는 것이므로, A스레드에서 FileIO를 요청했는데 완료되기전에 thread를 파괴시킨다면 모든 File IO는 그대로 끝이난다.**

파일IO는 전지적인 입장에서 요청하는 것을 일괄적으로 관리하는 것이 아니라, thread마다 관리하는 것. 따라서 완료에 대한 것도 이 thread에서 받아야한다.

**APC 큐**

**APC큐는 작업이 완료를 쌓아놓는 역할을 하며, thread마다 존재한다.**

프로세스 단위로 관리당하는 개념자체가 없음.

**IOCP를 임의로 만들어야 사용이 가능한 것과는 달리 APC(asynchronous procedure call)는 원래 존재한다.**

**시스템에서 APC를 많이 사용하며, APC큐는 스택이 아니라 해당 thraed커널객체에 큐가 붙어있는 형태로 존재한다.**

(중간 정리 : Event방식 같은 경우는 Signal을 줄뿐이고, 어디선가 이 Signal을 받는 것. IOCP는 우리가 IOCP큐를 임의로 만들어 큐잉해주는 형태.)

(우리는 사용하지않지만 thread가 만들어질 때 마다 붙어있는 것들이 존재한다. APC, TLS...)

작업완료가 APC큐로 들어오면 우리가 등록한 콜백함수를 우리의 스레드가 호출한다.

**우리가 직접적으로 큐를 확인하고 Deque 하지는 않을것이나, APC큐를 확인하여 함수를 호출할수 있는 시간을 부여해줘야 한다. (alertable waiT)**

마치 인터럽트와 같은 느낌.( 인터럽트는 CPU차원에서 명령어하나씩 처리할때마다 인터럽트 신호를 확인하여 신호가 있었다면 인터럽트 콜로 들어갔다가 다시 CPU명령어로 들어온다.)

**우리의 스레드가 매번 뭔가 일을 할때마다 APC큐에 뭔가 들어왔는지 확인하여, 일이 들어왔다면 일을 처리하고 없다면 자기일을 하는 구조가 되어야 한다.**

**결국엔 Completion Routine이 호출되는 것처럼 보이지만 우리가 호출시켜야 하므로, 구조가 지저분한 형태이다.**

**Alertable wait**

**Alertable wait이란 비동기IO를 위한 특별한 대기상태로, 비동기IO함수를 호출한 스레드가 이 상태에 있어야 완료루틴이 호출될 수 있다.**

APC큐를 확인해서 뭔가 들어왔다면 콜백함수를 호출시킬 시간을 부여하여 우리 스레드가 잠시 기다리도록 해야 한다.

**여기서 기다린다는 것은 코드상에서 기다리는것으로, 해당스레드 자체는 해당 함수를 호출하는것으로 깨어나서 일을 하고있다.**

(Alertable wait도 결국은 block당하러 가는것이다. 실제로 새로운 형태의 wait이 아님)

**Completion Routine 단계별 동작**

**1.비동기 입출력 함수를 호출하여 운영체제 입출력 작업을 요청한다.**

**2.해당스레드는 곧바로 alertable wait상태로 진입한다.**

(꼭 곧바로 할필요는없음. 주기적으로는 해야함. 있나없나는 모르므로 매번 확인해야한다.)

**잠깐 기다리고 APC큐를 확인해서 들어온게 있다면 함수호출을 진행하고, 이후에 alertable wait상태를 빠져나와 우리코드로 돌아온다.**

**이때는 (하나의 스레드지만) 코드의 제어권이 커널쪽으로 넘어가는 것이다.**

(커널 쪽 영역은 우리스레드에서 호출시키는 코드로서 존재하지만 우리가 짠 코드가 아니다.)

**스레드를 alertable wait상태로 만드는 함수는 대부분 ex가 붙으며, 기존의 있던 함수에 + alertablewait상태를 만드는 기능을 추가한 것이다.**

대기중인 상태에서는 APC큐를 계속해서 확인하면서 뭔가 들어왔을 때 그 함수를 호출하는 행위까지도 가능하다.

WaitFor..Ex함수들, SleepEx(), WSAWaitForMultipleEvents()등이 존재한다.

특히 WSAWaitForMultipleEvents()는 WSAEvenetSelect모델과 Ovelapped모델에서 다룬적있는데, 마지막 인자에 TRUE를 사용하면 해당스레는 alertablewait상태가 된다.

우리가 실제로 함수호출 call은 안하겠지만 결국 수행되는단위는 우리스레드이다.

**3. 비동기 입출력 작업이 완료되면, 운영체제는 스레드의 APC큐에 결과를 저장한다.**

(운영체제가 스레드에 할당한 메모리 영역을 말함.)

**4. 비동기IO 함수를 호출한 스레드가 alertable wait상태라면 운영체제는 APC큐에 저장된 정보 함수 포인터를 참조하여 해당 함수를 호출한다.**

(해당 완료루틴 안쪽에서도 다시 비동기 입출력이 가능하다. 호출된 함수도 내 스레드이므로)

**5. APC큐에 저장된 정보를 토대로 모든 완료 루틴이 호출이 끝나면 스레드는 alertable wait상태에서 빠져나온다.**

(이때 만약 SleepEx()로 INFINITE를 걸었다고 가정하자.

이때 APC큐에 뭔가가 와서 완료루틴을 호출시켰다면 스레드가 깨어나서 APC결과를 하나 처리했다고 알려준다. (리턴값도 같이)

만약 SleepEx에 시간을 지정했다다면, 이 시간을 초과하여 Sleep상태일 수도있다.

**(그렇다면 만약에 완료루틴이 한참 걸렸다면 그럼 돌아와서 Sleep된 시간만큼 다시 기다리는가?)**

**스레드가 비동기 입출력 결과를 계속 처리하려면 다시 alertable wait상태에 진입해야 한다.**

**Completion Routine 소켓 입출력 동작**

**1. 비동기 입출력을 지원하는 소켓을 생성한다.(socket())**

**2. 비동기 입출력을 지원하는 소켓함수(..Ex)를 호출한다.**

이때 완료루틴의 시작 주소를 함수인자로 전달한다. 비동기IO 작업이 곧바로 완료되지않으면 소켓함수는 SOCKET\_ERROR - WSA\_IO\_PENDING를 반환한다.

**3. 비동기IO 함수를 호출한 스레드를 WaitForSingleObjectEx(), WaitForMultipleObjectsEx(), SleepEx() 등의 함수를 통해서 alertablewait상태로만든다.**

**(요청 스레드의 APC큐로 들어오기 때문에, 요청한 스레드가 받아야 하는 구조이다.**

**서버입장에서는 병렬처리를 하고싶은 것이므로 하나의 스레드에서 처리를 하고자 하는 이 구조는 애매하다.)**

**4. 비동기 입출력 작업이 완료되면, 운영체제는 완료루틴을 호출한다. 완료 루틴에서는 비동기 입출력 결과를 확인하고 후속처리를 한다.**

**5. 완료루틴 호출이 모두 끝나면 스레드는 alertable wait상태에서 빠져나온다.**

스레드를 alertable wait상태로 만들수 있는 함수는 다양하므로, 필요할때마다 찾아서 사용해야한다.

여기서는 운영체제가 호출하는 완료루틴의 형태와 각 인자의 의미만 살펴본다.

Alertable wait은 스레드의 대기상태, 즉 wait state중 하나이다.

blocked상태와 차이점은 무엇일까?

**Alertable wait상태인데 APC큐에 아무것도 들어오지않아서 아무일도 하지않는다면 CPU는 아무일도 하지않겠지만, 콜백함수가 호출되어 일을한다면 CPU가 사용될 것이다.**

**CompletionRoutine**

void CALLBACK CompletionRoutine

(

DWORD dwError,

DWORD cbTransferred,

LPWSAOVERLAPPED lpOverlapped,

DWORD dwFlags

);

DWORD dwError,

비동기 입출력 결과이다. 오류가 발생하면 0이아닌 값이 된다.

DWORD cbTransferred,

전송바이트 수. 통신상대가 접속을 종료하면 이 값은 0이된다.

LPWSAOVERLAPPED lpOveerlapped,

비동기 입출력함수 호출 시 넘겨준 WSAOVERLAPPED구조체의 주소값이 이 인자를 통해 다시 응용프로그램에 넘어온다.

**Overlapped모델에서는 이벤트 객체를 사용하지않으므로 WSAOVERLAPPED구조체를 완료 루틴내부에서 직접사용할 일은 없다. 완료시 알려주는 개념인 것.**

DWORD dwFlags

NULL. 사용하지않는 값이다.

Completion Routine방식은 OVERLAPPED\* 형태를 뱉어주기 때문에, Signal만을 주던 Event방식보다는 나은 형태이다.

**이 구조체를 WSASend, WSARecv의 마지막인자로 넣을 경우 완료루틴방식으로 돌아간다.**

**WSAOVERLAPPED 확장**

OVERLAPPED I/O에서는 WSAOVERLAPPED라는 구조체 포인터를 작업의 단위로 보기 때문에, 우리가 직접 관리해야할 대상이다.

대부분은 이 구조체를 확장해서 사용한다.

struct MyOverlapped

{

WSAOVERLAPPED Overlapped;

int Mode; -> Send / Recv;

\_SESSION \*Session;

}

거의대부분 이방식을 택한다.

**WSASend WSARecv인자로 NumofByte는 처리결과로 우리쪽으로 DWORD형태의 포인터를 넘기도록 되어있다. 하지만 비동기IO의 경우 당장 결과가 나올수 없으므로, NULL을 전달하여도 상관없다.**

OVERLAPPED구조체를 전달하는 행위자체가 비동기IO로 진행해달라는 요청이다. 이때, 동기입출력이 됐다면 해당 인자는 성공한 값을 뱉어 줄 것.

OVERLAPPED는 무조건 보존해야하며, 전달하는 BUF는 보존할 필요가없다. 리턴되면 제역할을 다한 것.

**446P Compleiton Routine 예제코드 분석. <27:24>**

여기서는 ReadEvent와 WriteEvent라는 객체를 만들고, 자동리셋모드로 셋팅했다.

그리고 하나는 처음부터 signal을 준상태, 하나는 signal이 없는상태.

메인스레드는 보니 accpet만 하고있다.

**OVERLAPPEDIO에서는 항시 모든 소켓을 대상으로 Recv가 대기상태 여야한다.**

이 예제에서는 메인스레드에서 Recv를 걸게되면 메인쪽으로 완료요청이 오기 때문에, 메인스레드에서 recv를 걸수가 없는 상황이다.

때문에 메인스레드에서는 accept만을 받고 실제 Workerthread에게 알려주는 방법을 택했다.

**만약 main thread에서 accept받아 소켓만 뽑아내고 뽑아낸소켓을 워커스레드로 알려주고 싶다면, Queue라는 매개체를 통해 스레드에게 전달요청을 해야한다.**

책에서는 Queue가 아닌 전역변수를 통해 전달하는 방식이다.

결국 이러다보니 병렬로 전달이 되지않고 맞물려가는 형식이다. 절대로 이런방식은 사용되지않는다.

**OVERLAPPED 구조체는 최소한 두개이상은 나와야하고(Recv용, Send용), 받는쪽에서 Send인지 Recv인지 구분해야 한다.**

우리는 OVERLAPPED포인터를 받아서 세션으로 캐스팅할 수가 없다. 위에언급했든 OVERLAPPED구조체를 확장하여 그OVERLAPPED를 두개이상으로가는 것.

Completion Routine의 Transferred가 음수가 나올일은 없다. 예전에는 Recv/Send가 리턴값이 송수신의 결과값과 에러처리 두가지행동이 들어있다보니까 둘다 리턴값을통해 판단했는데, Transferred는 송수신에 대한 결과가 나오는 것이기 때문에 에러가 나오지않는다.

때문에 에러가 났다면 완료하지 못한상황으로, transferred가 셋팅되지않는다.

(종료됐다고 0을 넣어준다는 느낌보다는, 셋팅되지않았기 때문에 0이나왔다고 보는 것이 맞다.)

**OVERLAPPED 구조체자체를 동적할당하여 포인터를 던지고, 완료통지 받은쪽에서 해제하는 형태로 사용한다면 보관하지않아도 되지만 성능적인 문제가 있을 것.**

**Completion Routine 단점**

**작업요청을 한 스레드가 무조건 완료통지를 받아내야하므로, 유저별로 스레드를 나누지않는이상 병렬처리가 불가능하다.**

소규모의 인원을 컨텐츠를 다룬다고한다면 괜찮겠지만 여기서도 좋은구조는 아니다.

**완료루틴의 개념은 사용하지않을 것이나, 하지만 APC Queue의 개념을 알고있어야할 것.**

APC큐에 우리가직접 뭔가를 넣을 수 있는데, 어떤함수를 호출시켜서 APC에 큐잉한다면 alertable wait상태로 만들어놓고 APC 큐에 있는 완료통지를 뽑아서 호출해주는 것.

**APC큐의 활용**

A스레드에서 길찾기 작업이 끝났다고 가정하면,

완료정보를 메인스레드쪽으로 전달하고, 메인스레드는 완료정보를 받아 마무리 작업을 해야한다. 이때 완료정보를 전달해야 하는데, 세가지 방법을 제시한다.

**1. 우리가 직접만든 Queue**

**우리가 직접 완료통지를 담아두는 큐를 만들고, Workerthread (길찾기 스레드)에서 작업이 완료되면 작업완료 정보를 큐잉하고, main스레드는 루프돌때마다 큐를 확인하면서 마무리 작업을 해주는 형태.**

**2. APC Queue의 사용**

**APC큐를 사용하여 main thread Update()쪽에서 alertable wait 상태를 을 한번씩 만들어 완료되는 것이 있다면 진행하는 것. (Ex가 붙은 함수를 통해서 alertable wait상태를 만듬)**

완료통지쪽에서는 어떤함수를 임의로 호출시키는것이다.

**3. IOCP사용**

(생략. 뒤이어 언급됨)

전역데이터든, 클래스멤버든 어떤 메개체를 통해 접근해야 한다.

꼭 APC큐를 사용하지않아도 상관없지만, 언제 이런상황이 나타날지는 모름.

따라서 여러가지 방법을 익혀두는 것이므로, 반드시 기억해둘 것.

APC큐, 직접만든 큐, IOCP…

이셋의 성능평가는 개인적인 판단에 맡긴다. (거의 체감되지않음)

APC큐의 크기는 알수없음. IOCP의 경우는 계속해서 큐잉할수있는데, Non-PagedMemory를 사용할것이므로 로직차원에서 막아야 한다.

**Completion Port모델**

**Completion Port모델은 모든 소켓 입출력 모델 중 가장 뛰어난 성능을 제공한다.**

IOCP를 소켓모델로 봐도 되지만 실체는 하나의 Queue를 뜻한다.

완료된 어떤 정보들을 관리해주는 방법일 뿐이다.

IOCP는 소켓I/O 뿐만 아니라 File I/O, threadPool의 개념으로 사용해도 된다.

**Completion Port모델의 동작원리**

Completion Port모델의 핵심은 입출력 완료 포트라는 윈도우 운영체제가 제공하는 구조를 이해하고 활용하는것이다.

**입출력 완료포트(I/O completion port)는 비동기 입출력 결과와 해당 결과를 처리할 스레드에 관한 정보를 담고있는 구조로, Overlapped모델에서 소개한 APC큐와 비슷한 개념이다.**

**APC Queue VS Completion port**

**생성/파괴**

**APC큐의 경우 스레드마다 자동으로 생성/파괴된다. 개발자가 임의로 제어할수 없음.**

**Completion Port는 생성은 CreateIoCompletionPort(), 파괴는 CloseHandle()를 사용한다.**

**접근제어**

**APC Q에 저장된 결과는 APC Q를 소유한 스레드만 확인할 수 있지만, 입출력 완료포트에는 이런 제약이 없다.**

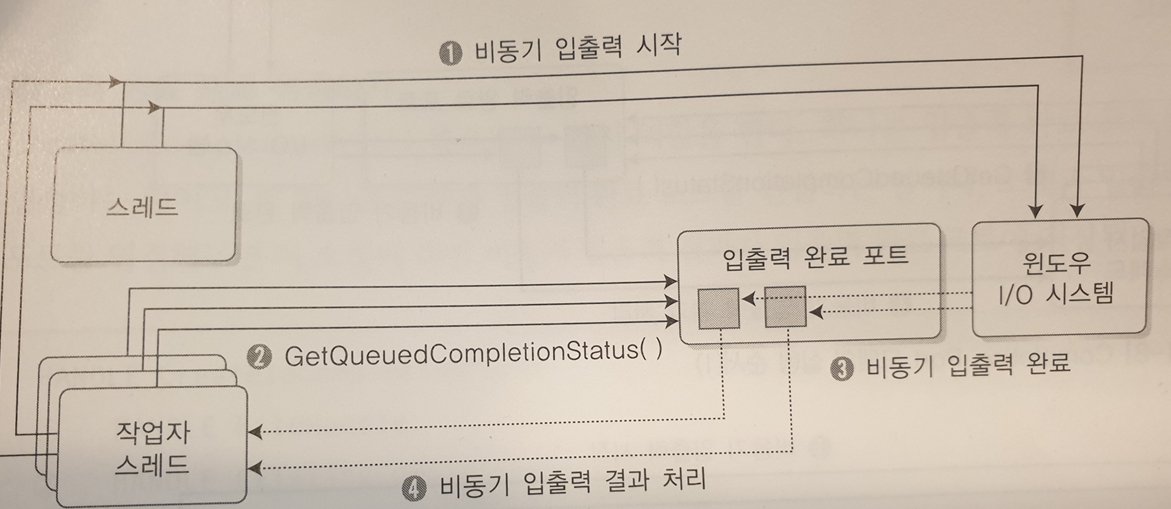
CreateIoCompletionPort() 함수를 호출하면 IOCP핸들을 반환한다. **어떤 스레드에서든 이 핸들에 접근할 수 있다고 한다면 해당스레드가 Workerthread가 되는것이다.**

**비동기 입출력 처리방법**

**APC Q에 저장된 결과를 처리하려면 해당 스레드는 alertable wait상태에 진입해야 한다.**

**입출력 완료 포트에 저장된 결과를 처리하러면 작업자 스레드는 GetQueuedCompletionStatus()함수를 호출하면 된다. (IOCP)**

(IOCP같은 경우 작업요청한 스레드가 아니더라도 전혀 상관이 없다는 뜻이된다.)



453P 그림 11-7.

(위그림은 입출력 완료 포트와 WorkerThread는 미리 생성해 둔것으로 간주한다.)

일반적으로 WorkerThread는 해당 일만을 전담하는 별도의 스레드로, 똑같은 형태의 코드가 되겠다. (하나의 코드를 대상으로 여러 개의 스레드를 만들어서 스레드풀로 돌리겠다는 뜻.)

**Completion Port 동작 원리**

**1.응용 프로그램을 구성하는 임의의스레드에서 비동기 입출력 함수를 호출함으로서 운영체제에 입출력 작업을 요청한다.**

**2. 모든 worker thread는 GetQueuedCompletionStatuts()함수를 호출해 Completion Port(입출력 완료 포트)를 감시한다**. 완료한 비동기 입출력 작업이 아직 없다면 모든 작업자 스레드는 대기상태가 된다.

**당연하게도 대기중인 worker thread들은 모두 completion port 내부에서 관리된다.**

(하나의 Event객체를 대상으로 여러 개의 스레드를 WaitForMultipleObjects() 함수로 signal을 기다리고 있었고, Event객체는 이를 알고 있었다. 그리고 해당 Event Signal이 들어가면서 어떤 thread를 깨울지 결정한다.)

**3. 비동기IO 작업이 완료되면 운영체제는 입출력 완료 포트에 결과를 저장한다. 이때 저장되는 정보를 입출력 완료패킷(I/O completion paket)이라 부른다.**

**4. 운영체제는 입출력 완료 포트에 저장된 작업자 스레드 목록에서 하나를 선택하여 깨운다. 대기 상태에서 깨어난 작업자 스레드는 비동기 입출력 결과를 처리한다.**

이후 작업자 스레드는 필요에 따라 다시 비동기 입출력 함수를 호출 할 수 있다.

여기서는 딱히 Workerthread라는 개념이 없으며, 어떤 thread든 해당 IOCP핸들 대상으로GetQueuedCompletionStatuts()함수를 호출한다면 해당 스레드가 workerthread가 되는것이다.

(GetQueuedCompletionStatuts()는 줄여서 GQCS라 부른다.)

**Completion Port 구체적인 사용**

1. CreatIoCompletionPort()함수를 호출하여 입출력 완료 포트를 생성한다.

2. CPU개수에 비례하여 workerthread를 생성한다. 모든 workerthread는 GQCS를 호출하여 Blocked상태가 된다.

3. 비동기IO를 지원하는 socket을 생성한다. **이 소켓에 대한 비동기IO결과가 입출력 포트에 저장되려면, CreateIOCompletionPort()함수를 호출하여 소켓과 입출력 완료 포트를 연결해야한다.**

4. 비동기IO 함수를 호출한다. 비동기IO작업이 곧바로 완료되지않으면, 소켓함수는 에러를 리턴하고(SOCKET\_ERROR) 오류코드는 (WSA\_IO\_PENDING)으로 설정된다.

5. 비동기 입출력 작업이 완료되면, 운영체제는 입출력 완료 포트에 결과를 저장하고, 대기중인 스레드하나를 깨운다. 대기상태에서 깨어난 작업자 스레드는 비동기 입출력 결괄를 처리한다.

**CreateIoCompletionPort**

HANDLE WINAPI CreateIoCompletionPort(

\_In\_ HANDLE FileHandle,

\_In\_opt\_ HANDLE ExistingCompletionPort,

\_In\_ ULONG\_PTR CompletionKey,

\_In\_ DWORD NumberOfConcurrentThreads

);

**이 함수는 IOCP(완료포트)를 생성시키거나, (반환한 핸들은 저장해야한다.) 이미 만들어진 IOCP에 추가적인 핸들을 등록하는 두가지 기능이 있다.**

어떤 IO대상이 되는 핸들(파일, 소켓..등)이 등록된다는 개념으로 기존방식과는 다른개념이다.

(Event 방식이나 Completion Routine에서는 존재하지않던 개념이다.)

**IOCP에서는 CompletionKey라는 것이 제공되므로, 우리는 셋팅하여 사용하면 된다.**

(Event의 경우 Signal만을 알려줬고, Completion Routine의 APC큐는 OVERLAPPED 구조체의 포인터를 전달했기 때문에 이를 활용했다.)

이는 굉장히 좋은 기능이다.

IOCP는 하나의 핸들당 하나의 키(고유값)을 제공하고 이를 마음대로 사용해도 된다. 권장하는 방법은 세션포인터를 넣는 것.

OVERRAPPED구조체를 확장해서 사용해도 되지만, Key를 사용한다면 활용성이 높아진다.

(키값은 ULONG\_PTR이므로 포인터를 넣어야한다.)

HANDLE FileHandle,

**입출력 완료 포트와 연결할 핸들이다. 새로운 입출력 완료 포트를 생성할 때는 유효한 핸들 대신 INVALID\_HANDLE\_VALUE값을 사용하면된다.**

HANDLE ExistingCompletionPort,

**파일 또는 소켓과 연결할 입출력 완료 포트 핸들이다. 이값이 NLL일 경우 새로운 입출력 완료 포트를 생성한다.**

**정리하자면, 첫번째 인자로 들어온 핸들을 두번째들어온 IOCP에 연결해주는 역할을 한다.**

ULONG\_PTR CompletionKey,

**입출력 완료 패킷 (I/O completion packet)에 들어갈 부가 정보로 64비트 포인트값을 줄수있다.** 여기에 세션포인터를 넣는다면 편리하게 사용이 가능할 것.

DWORD NumberOfConcurrentThreads

**동시에 실행 할 수 있는 작업자 스레드의 개수이다. 0을 사용하면 자동으로 논리 CPU개수와 같은 수로 설정된다.**

**(듀얼코어에 하이퍼스레딩이 지원된다고 한다면 논리 코어개수는 4개가 될 것.)**

운영체제는 실행중인 작업자 스레드 개수가 여기서 설정한 값을 넘지 않도록 관리해준다.

**마지막 인자 NumberOfConcurrentThreads**

HANDLE hcp = CreateIoCompletionPort(INVALID\_HANDLE\_VALUE, NULL, 0, 0);

**NumberOfConcurrrentThreads는 처음에 IOCP를 생성할때만 활성화가 될것이고, 이후에 새로운 소켓이 들어와서 연결시킬때는 이 인자가 무시된다. 한번 만들어 놓은경우 변경이 불가능**

한번에 깨어날수있는 스레드를 제어하는 인자이다.

(기존의 어떤 방식에서도 존재하지않던 방식.)

IOCP가 나오면서 윈도우에서 제공하는 스레드풀의 개념이 획기적으로 발전되었다.

최근나오는 API에는 threadpool이라는 개념이 따로 존재하는데, 이 역시 IOCP기반이다.

(잘 사용하지는 않음)

기존에있던 Event방식은 Signal이 오는대로 block스레드를 깨워버리기 때문에,

**Event방식은 하나의 thread가 깨어나서 일을 할 때 너무 오래걸려 해당일을 끝마치기 전에 다시금 signal이 온다면 이때마다 계속해서 thread가 깨어날 것이다.**

**하지만 IOCP는 NumberOfConcurrrentThreads인자를 2로 넣는다면 2개까지만 깨게 될것이다. (이는 RunningThread 제어)**

IOCP는 자신을 기다리며 block당한 thread를 모두 알고있으며, 내부에서 카운팅중이며,

NumberofCurrnetThreads인자로 n을 넣어 RunningThread를 제어한다.

**이때 RunningThred가 block당하는 경우(Sleep, Lock, I/O작업 등..) 다른 WorkerTherad가 치고 들어온다.**

**이후 block된 Thread가 다시금 Running상태로 돌아온다면 이 경우 우리가 지정한 개수를 초과하게된다. (해당 경우에만 초과할 것)**

**RunningThread 개수조절**

**NumberofCurrentThread 인자를 0으로 할경우 논리 CPU코어 개수에 맞춰준다.**

(다른곳에서 소개되는 CPU코어개수\*n 과 같은 스레드는 RunningThread가 아닌 WorkerThread의 개수를 말하는 것이다.)

**RunningThread는 논리CPU코어 이상의 수를 대입하는 것은 아무의미가 없고, 오히려 역효과가 날수도있다. 어차피 넣을 수 없음**

또한 우리가 만든 서버와 네트워크앱에서만 WorkerThread가 도는 구조가 아니므로, 다른 스레드의 활동성을 고려해서 계산이 이뤄져야 한다.

- 로직스레드가 너무 바쁜경우 내 코어개수에서 -1해야한다. (항시 돌것이기 때문)

- 할일이 거의 없거나 컨트롤 역할만 한다면 없는셈 쳐도 된다.

- 워커스레드 내부에서 블락걸릴 일이 많은경우, WorkerThreaad를 많이 만들면 효과를 볼 것.

**WorkerThread 개수 조절**

WorkerThread 내부에서 block될 일이 많다면..

(로그인 서버와 같이 DB작업이 포함된 경우. DB는 모두 동기Proc이기 때문에 I/O작업시 모두 block된다. 오해금지: DB저장을 비동기로 뺀다는 것은 우리가 별도의 스레드로 빼는것.)

**WorkerThread를 3~40개정도로 많이 만든다면 효과가 확실할 것이다.**

**WorkerThread를 너무 많이 만드는경우, 최악의 경우 모든 WorkerThread가 깨어나서 Running될수도 있을 수 있으나, block시간이 길 경우 다른것도 돌아야 하므로 역효과가 나더라도 이 방법이 낫다.**

단, workerthread내부에서 block걸릴 일이 없다면 많이 만들필요 없을 것.

기본은 코어개수와 부가적으로 만든 스레드를 계산하고, 이후 러닝스레드를 몇 개를 잡을 것인가에 대한 기준으로 +-하는형태.

**일반적인 게임서버의 경우 WorkerThread내부에서 Proc를 돌리지 않기 때문에 Send,Recv를제외한다면 Block걸릴일이 거의 없다.**

**설정파일로 Bindport, IP, 동시접속자 수 등을 별도의 설정파일과 같은 것으로 뺄것인데, 이때 WorkerThread와 RunningThread를 몇 개로 할 것인지도 포함될 것이다.**

**IOCP를 스레드풀로 활용시 Event방식과 비교**

**1. Stack방식으로 돌아가는 IOCP스레드풀**

**스택방식으로 관리한다면 캐시히트를 높일 수 있을 것이므로, Queue 방식의 Event 스레드풀보다 성능이 좋을 것.**

**2. Running스레드를 제어할 수 있다.**

Event방식의 스레드풀일경우 Signal이 올때마다 스레드를 깨워버리는 상황이 올 수도있음

**3. Signal에 대한 카운팅이 불가능한 Event 스레드풀**

**Signal이 여러 번 왔다고 하더라도 Event입장에서는 한번밖에 인식하지 못한다.**

따라서 JopQ에 일이 쌓여있을 때 하나의 스레드만 깨어나서 이를 처리하게 될것이다.

이 경우 제대로된 스레드풀 기능을 하고있다고 말하기 어렵다.

**GQCS**

BOOL GetQueuedCompletionStatus(

HANDLE CompletionPort,

LPDWORD lpNumberOfBytesTransferred,

PULONG\_PTR lpCompletionKey,

LPOVERLAPPED \*lpOverlapped,

DWORD dwMilliseconds

);

**이함수를 호출하여 입출력 완료포트에 입출력 완료 패킷이 들어올 때 까지 대기한다.**

(당연히 IOCP는 내부에서 몇 개의 스레드가 대기중인지 알고 있을 것)

입출력 완료패킷이 입출력 완료포트에 들어오면 운영체제 실행중인 작업자 스레드 개수를 체크하여, **해당값이 CreateIoCompletionPort()함수의 네번째 인자로 설정한 값(RunningCount)보다 작다면, 대기상태인 workerThread를 깨워 완료패킷을 처리한다.**

HANDLE CompletionPort,

입출력 완료 포트핸들

LPDWORD lpNumberOfBytesTransferred,

**비동기 입출력 작업으로 전송된 바이트수가 여기에 저장된다.**

APC Q에 Completion Routine에 들어왔던 애랑 같은 역할이다.

PULONG\_PTR lpCompletionKey,

**CreateIoCompletionPort()함수호출시 전달한 세번째 인자가 여기에 저장된다.**

우리가 마음대로 쓸값.

LPOVERLAPPED \*lpOverlapped,

비동기 입출력 함수 호출시 전달한 OVERLAPPED구조체의 주소값이 여기에 저장된다.

OVERLAPPED의 포인터를 받는형태이기 때문에 이중포인터 형태를 띄고있다.

DWORD dwMilliseconds

작업자 스레드가 대기할 시간을 밀리초단위로 지정한다. INFINITE값을 넣으면 입출력 완료패킷이 완성되어 운영체제가 깨울때까지 무한히 대기한다.

일반적으로 INFINITE가 들어간다.

**GQCS 주의할점**

**WorkerThread내부에서 GQCS를 호출시키기 직전에 지역변수 transferred, CompletionKey값, \*OVERLAPPED는 반드시 초기화 할 것.**

**GQCS가 false를 반환하여 실패를 알려주지만 이것이 IOCP의 실패일지, GQCS의 실패일지, 어떤 다른작업에 대한 실패일지 알 수 없음.**

(일반적으로는 IOCP가 실패할 가능성은 거의없다고 봐도 무방하다. GQCS에서 실패하는 경우 IOCP핸들 자체가 파괴되었거나, 핸들을 잘못넣었거나, TimeOut값을 초과했을 경우)

**만약 TimeOut이 난 경우 transferred, Overlapped값이 셋팅되지않기 때문에, 그전에 사용되던 값을 기준으로 로직을 들어가기 때문에 엉뚱한 결과가 나올것**(쓰레기값 아님.)

**GQCS를 호출 한 뒤 가장 먼저 해야할 일들**

**1. Overlapped, Key, transferred 세개의 값이 모두 NULL인가? (우리가 직접만든 예외)**

**2. Overlapped == NULL 일 때, 작업완료에 대한 결과가 아니므로 나머지 두개의 값을 쓰면안된다.**

**(Overlapped == NULL일때는 IOCP자체의 문제거나, GQCS의 TimeOut초과.)**

**저 세개의 값을 초기화하지않는 경우 (Key에는 보통 Session), GQCS가 리턴이 되고 transferred가 0, Key값에 세션이 존재하므로 종료시키는데, 이 세션은 사실 전프레임 세션 일 수 있다.**

**PostQueuedCompletionStatus()**

BOOL WINAPI PostQueuedCompletionStatus(

\_In\_ HANDLE CompletionPort,

\_In\_ DWORD dwNumberOfBytesTransferred,

\_In\_ ULONG\_PTR dwCompletionKey,

\_In\_opt\_ LPOVERLAPPED lpOverlapped

);

GQCS와 인자가 같으며, 마치 작업이 완료된것처럼 큐에 꽂아준다.

**IOCP를 스레드풀로만 사용할 때, 또는 Queue에 작업을 큐잉하고 이벤트를 기다리는 형태를 사사용하고 싶을 때 사용한다.**

Transfferred에 숫자를, Key와 Overlapped 포인터에 넣고싶은 걸 넣을 수있으므로, 워커스레드에서 GQCS에서 저 셋만 받아서 일을 하면 된다.

주의할점은, 큐자체가 NonPagedPool-Memory를 사용하고 있으므로 염두해둬야할 것.

(실제로 개인적인 threadpool역시 IOCP를 많이 사용한다.)

**또한 워커스레드를 종료하는 목적으로 사용하기도 한다.**

GQCS에 INFINITE를 넣었을 경우 WorkerThread를 종료할 방법이 없기 때문에, PQCS를 이용하여 WorkerThread를 종료한다. 이는 널리 사용하는 정석적인 방법.

**Transferred, Key, Overlapped 세 인자에 모두 NULL을 넣어서 WorkerThread를 종료시키는 기준으로 사용한다.(**우리가 직접 예외처리를해서 종료하는 것)

**때문에 IOCP워커스레드에서 항시 GQCS에서 이 세개의 변수가 0인지부터 확인한다.**

**457P예제**

**mainthread**

IOCP핸들은 전역으로 둬도되고. 보통은 싱글톤 클래스 내부의 멤버로 둔다.

GetSystemInfo(&si)를 해서 NumberOfProsess(CPU코어 개수)를 구한다. 이 개수만큼 워커스레드를 만든다.

**(이 개수를 CreateIoCompletionPort의 마지막인자로 넣는 것은 잘못 이해하여 사용하는 것)**

예제에서는 main thread가 accept thread로, 평상시에 block상태로 있다가 새로운 연결이 들어올 경우 깨어나서 로그를 출력하고 CreateIoCompletionPort로 바로 등록한다.

이후 세션을 만들고, Overlapped구조체를 초기화하여 바로 recv를 건다.

그리고 accept를 호출한다.

(acceptthread에서 Recv까지 걸어도 상관없다. IOCP를 사용했기 때문에 완료통지에서 그쪽으로 올 것. 이제 우리가 연결해놓은 Socket에 대한 IO를 해당 IOCP에 연결시켜놨기 때문에)

**WorkerThread**

**GQCS를 타고 일이 올때까지 무한히 대기한다**.

성공적으로 깨어났다면 transferred와 Key, Overlapped가 셋팅이 될 것.

제일먼저, 이 세개의 인자가 모두 NULL일 경우 WorkerThread를 종료시킨다.

(어차피 mainthread가 종료되면 나머지 thread도 모두 종료될것이지만, 원칙상..)

그다음 Overlapped이 NULL인지(GQCS의 실패) 확인한다.

WSAGetOverlappedResult는 의미없으므로 IOCP에서 사용하지않는다.

OVERLAPPED구조체는 튀어나왔는데 false를 반환했다면 그 작업(Send, Recv)이 실패한 것.

작업이 실패한경우 곧바로 종료루틴으로 들어간다.

Transferred를 확인할 필요도 없음. Transferred가 0이냐로 끝난다.

에러처리를 복잡하게하자면 한도끝도없는데다가 버그만 점점더 생긴다.

7.13\_2//Session

Socket

SendOverlapped

RecvOverlapped

RecvBuffer[]

SendBuffer[]

IOCP

**main문**

while(1)

{

Socket = accept();

CreateIOComplete..

세션생성

세션RecvOverlapped..초기화

WSARecv...

}

**Workerthread**

while(true)

{

ret = GQCS(-); <<여기서 lpOverlaaped가 나옴.

if(종료처리)

if(transferd == 0)

세션끊김처리..

여기서 Overlapped를 확인해서 Send인지 리시브인지 확인,

if(Overlaped확인, Send)

할거업슴

if(Recv)

Recv완료

Recv를 Send에 꽂아넣고, 샌드하세요. 됐네

}

**<7.15>**

**GetQueuedCompletionStatusEx**

BOOL WINAPI GetQueuedCompletionStatusEx(

\_In\_ HANDLE CompletionPort,

\_Out\_ LPOVERLAPPED\_ENTRY lpCompletionPortEntries,

\_In\_ ULONG ulCount,

\_Out\_ PULONG ulNumEntriesRemoved,

\_In\_ DWORD dwMilliseconds,

\_In\_ BOOL fAlertable

);

**(Ex는 보통 기존함수의 추가를 뜻한다.)**

많이 사용하는 함수는 아니다.

특이한점은, **OVERLAPPED\_ENTRY**라는 구조체의 포인터를 인자를 \_Out\_파라미터 형식으로 받고, 뒤이어 Count가 들어간다.

**이는 곧 OVERLAPPED\_ENTRY라는 구조체포인터를 배열로 선언하여 파라미터로 넣어줄 경우 완료통지를 한번에 뽑아준다는 얘기이다.**

(기존에있던 GQCS는 하나씩만 가능했음.)

**따라서 alretblewait상태로 진입하는 기능에 더해서, 여러 개의 완료통지를 한번에 뽑는 것이 가능해진 것이다.**

**그리고 ulNumEntriesRemoved에는 몇 개의 완료가 뽑혔는지 들어온다.**

**OVERLAPPED\_ENTRY**

typedef struct \_OVERLAPPED\_ENTRY {

ULONG\_PTR lpCompletionKey;

LPOVERLAPPED lpOverlapped;

ULONG\_PTR Internal;

DWORD dwNumberOfBytesTransferred;

} OVERLAPPED\_ENTRY, \*LPOVERLAPPED\_ENTRY;

Internal은 쓰지않는다.

위 같은 구조체를 배열로 만들어 GQCSEx로 전달한다.

**GQCSex의 특징**

**GQCSex의 장점으로는, GQCS의 호출량을 줄일 수 있다.**

GQCS라는 함수호출자체가 우리가 반복문을 돌리는것보다 가벼울 수는 없다.

**IOCP를 쓰는이유는 여러 개의 workerthread를 threadpool로서 병렬작업을 하기 위한것인데, 이는 일거리를 한곳에서 모두 뽑아가는 형태이다.**

**GQCSex를 고려할만할 대상**

**1. 또는 송수신의 완료통지 목적이 아닌 Jop에 대한 Queue로서만 활용할 때.(thread에 대한 메시지 전달로만 사용)**

**2. 우리가 준비한 Runningthread 개수 이상으로 완료작업이 쏟아져 들어올 때**

2-1. 사실 WorkerThread가 여러 개있는데도 감당하지 못할 정도로 완료통지가 쏟아지는 상황자체는 거의 존재하지않는다.

2-2 . **하지만 Worketthread가 하나인 경우는 라이브서비스에서도 많이 존재한다.**

**단일 WorkerThread**

(OVERRAPPED I/O는 비동기 입출력의 한 방법이고, IOCP는 threadPool의 개념을 적용해 여러작업을 병렬로 처리하기 위함이다..)

**Worker thread에서 모든일을 처리하자니, 게임이란 특성상 유기적으로 연결된 것이 많아** (동기화 증가) **Update라는 main thread로 단일화 시키거나 제어가능한 설계측면에서 일을 따로 Workerthread로 빼는 형태를 취하는 것이 가장 일반적인 게임서버의 IOCP모델이 된다.**

이때 Workerthread는 크게 담당하는 일이 없으므로 하나만 만들게 되는것이다.

(실제로 라이브 서비스에서 Workerthread를 하나만 쓰는 케이스가 많이 존재한다.)

Workerthread가 하나일 경우 로직이 편해지고, 동기화이슈도 훨씬 적게 발생할 것이다.

**고전적인 게임서버의 IOCP형태**

**(Update – 로직스레드, SendThread - send담당, WorkerThread – I/O담당)**

**동기화의 경합지점을 최소화시켜 workerthread에서 직접적으로 처리하는 것이 IOCP의 이상적인 목적이므로, 위와 같은 경우는 관점에 따라 IOCP를 잘못사용하는 케이스라고 볼 수 있다.**

게임서버가 이러한 형태가 된 것은 크게 두가지 이유가 있다.

**1. IOCP에 대한 오래된 강좌의 영향**

**2. Send를 1회로 제한하기 위함.(뒤이어 자세히 언급)**

**SendThread를 별도로 두는 이유**

우리는 오버랩IO를 쓰는 상황인데 Sendthread는 왜 따로 나왔는가?

**로직에서는 Send버퍼에 Send할 데이터를 쌓아놓고 SendThread에서 Send를 1회로 제한하기 위해서이다.**

**Send를 1회로 제한하는 잘못된 근거들**

**1. Send이후 또다른 Send가 들어가는 경우 Overlapped구조체가 하나 더 생겨야 하고, 이는 Overlapped I/O라는 일반적인 상식선에서는 허용하는 범위가 아니다.**

**2. 실제로 Send를 사용할때마다 내가 요청한 크기보다 작게 반환 되는 경우 돌이킬 수 없고, 서로 덮어쓰거나 동시에 가는 경우, 앞서보낸 결과가 누락되는 경우를 대비한다.**

**3. MS타이틀을 걸고나오는 책에서 IOCP의 작업 완료 순서의 보장이 없다는 문구가 존재.**

**위와 같은 이유들로, Send를 1회만 제한했으나 사실 이는 모두 잘못된 부분이다.**

**1. Send이후 또다른 Send가 들어가는 경우 Overlapped구조체가 하나 더 생겨야 하고, 이는 Overlapped I/O라는 일반적인 상식선에서는 허용하는 범위가 아니다.**

**-> 상식선에서 얘기한것이므로, 제외한다.**

**2. 실제로 Send를 사용할때마다 내가 요청한 크기보다 작게 반환 되는 경우 돌이킬 수 없고, 서로 덮어쓰거나 동시에 가는 경우, 앞서보낸 결과가 누락되는 경우를 대비한다.**

**-> WinSock에서는 FILE I/O와는 구조가 완전히 다르므로, 걱정하는것과는 달리 Send를 중첩시킨다고해도 반드시 순서를 지켜주며, 누락되는 경우도 모두 대비되어있다.**

**Overlapped I/O의 완료기준은 요청한 것을 끝냈을때를 뜻한다. 중간에 빠지는 것이 아니다.**

**(예외로, 연결이 끊길 때 내가 요청한 것보다 작은 크기가 나오는 케이스가 존재하는데 이경우 그냥 끊으면 되므로 더 이상 고민할 필요는 없다.)**

**3. MS타이틀을 걸고나오는 책에서 IOCP의 작업 완료 순서의 보장이 없다는 문구가 존재.**

**-> 이는 오역으로, 결론을 말하자면 IOCP의 작업완료는 보장되지만 큐에 완료통지가 들어가는 순서를 보장해주지 못한다.**

**결론적으로 위와 같은 이유로, Send를 여러 번 하는 것은 기능적으로 아무 문제가 없다.**

따라서 Send를 하고싶을 때 한다고 해도(이Send용 링버퍼가 필요없어질 것) 전혀 상관없다.

**그래도 Send를 1회로 제한하는 이유**

**Send를 마구 보낸다고해서 얻을 수 있는 장점이 없다. 오히려 성능만 떨어질 것.**

때문에 예나 지금이나 Send는 1회로 제한한다.

**결국, 이전이나 지금이나 Send를 1회로 제한하는 것은 같지만 이유가 다르다. 예전에는 기능적으로 안된다고 잘 못알고 있어서, 지금은 구조가 달라지고 성능이 떨어지기 때문에.**

**Recv를 1회로 제한하는 이유**

**Recv역시 중첩으로 걸어놔도 기능적으로는 문제가 없으나, Send와 마찬가지로 장점이없다.**

**순서 보장이 되어도 한thread에 대해 여러 개의 Workerthread에 완료통지가 떨어지면 우리로직에서는 순서보장이 되지않는다.**

**구체적으로, Recv를 3개 걸어서 완료통지가 3개가 왔을 때 거의 동시에 진행된 3개의 thread중 누가 먼저 끝낼지는 우리가 알수가 없다**.

(동기화하면 보장받겠지만 이러면 병렬화의 장점이 아무의미없어지는 꼴)

(http의 경우 하나의 Recv당 하나의 Connection을 사용하기 때문에, 로직을 처리하고 소켓을 끊어버리는 구조이다. 이러한 구조에서는 논의거리조차 나오지않을 것.)

**비동기 IO의 극대화는 결국 병렬처리의 극대화를 뜻한다.**

우리는 에코예제에서 Recv가 오면 다시 Send해서 보낸다음 다시금 Recv를 걸었다.

이 예제에서 Overlapped의 기능을 극대화시켜서 설계한다면,

**If(Recv완료통지.)**

**<받은메시지 처리>**

**WSARecv**

**If(Recv완료통지)**

**WSARecv(…)**

**<받은 메시지처리>**

위와같이 바꿀수 있다. recv를 미리 백그라운드에서 걸어놓고 우리할일이 들어가는 것이므로, 병렬처리의 극대화를 노려본 것.

하지만 이 구조의경우 문제가 존재한다.

(1)

A스레드가 Recv완료통지를 받아 처리하는도중,

B스레드의 Recv완료통지가 와서 먼저 처리될 수있다.

이경우 메시지가 꼬여버리는 것.

(2)

**WSARecv 호출했을 때 바로 IO작업이 들어가면 로직이 처리되기도 전에 Recv완료통지가 먼저오기 때문에, 처리되지 않은 데이터가 내 링버퍼에 계속해서 쌓이게 된다.**(Recv메시지는 링버퍼에 쌓아놨으므로 순서처리가 바뀔일은 없다.)

**이는 내 서버의 한계치를 넘도록 Recv를 받는다는 얘기가 된다.**

**때문에 우리는 WSARecv를 1회로 제한한다. 기능적으로는 가능하나 우리가 제대로 사용할 수 없는것.**

굳이 게임이 아니더라도 위문제는 발생하는 구조.

**1.Page-locking**

**WSASend, WSARecv를 할 때 결국은 완료통지가 오기전까지 버퍼자체(힙), OVERLAPPED구조체는 보존되어야 한다.**

OVERLAPPID I/O는 백그라운드(커널쪽)에서 카피를 하는 행위이고, Recv를 걸어놓으면 커널쪽에서 계속 대기하고 있는 상태이다.

**이때 대기하고있던 힙 메모리영역이 PageOut이 되는 것을 방지하기위해 PageOut락을 건다.**

**(이 락은 PageOut락이므로 해당메모리는 언제든지 우리가 접근하여 사용할 수 있음.)**

**IO역시 버퍼를 대상으로 포인터를 잡고있는 형태이므로, 이 메모리가 PageOut된다는 것은 블루스크린감이다.**

이때 내부적으로 비동기IO로 돌 때 페이지락을 걸어야 하는 상황이 된다.

따라서 언제 정확히 비동기IO로 도는지 알아야 한다.

**비동기IO가 작동되는 상황**

우리는 예제를 진행하면서 비동기IO를 사용했기 때문에 내부적으로 비동기IO가 돌것이라고 기대했다. 하지만 Send의 경우는 단 한건도 OverlappedIO로 작동하지않았다.

(언급했듯 FILE I/O와 소켓쪽은 작동방식이 완전히 다름)

**반환값이 Error – WSA\_IO\_PENDING 인경우에만 OVERLAPPED I/O로 들어간 것이고, 반환이 성공했다면 동기입출력을 통해 해당 Send나 Recv의 요청이 성공했다는 뜻이다.**

**소켓IO는 비동기IO가 되는순간이 굉장히 제한적이며, 지금당장 일을 할수있다면 동기로 작동되고, 일을 할 수 없을 때만 오버랩IO로 전환된다.**

**비동기IO로 작동되는 순간은 WSASend의 경우 소켓버퍼가 가득찼을 때, WSARecv의 경우 소켓버퍼로부터 받을 데이터가 없는 경우일 것이다.**

**우리가 일부러 비동기가 되게끔 의도하고자 한다면, 송신버퍼를 0으로 만들면 될것이다.**

**WSANOBUFS Error**

모든 메모리에 락을 걸면 가상메모리 구조자체가 의미가 없어진다.

**일정시간이상 용량이 넘어가는 경우는 그 이상 락을 걸지못하도록 한계치가 정해져있다.** (여기에 대한 정확한 수치는 아직 찾지못함)

**이 한계치를 넘어서는 락을 걸게되면 WSANOBUFS 라는 메모리구조 에러가 뜨게되고, 이는 정말 최악의 상황이다.**

최대한 어떤상황에서 이에러가 발생했는지 정확히 파악하여 로그를 남기고 서버를 어떻게 할것인지 판단해야 한다.

**실제로 해당에러를 경험한 사례는 극히 드물지만 msdn및 모든서적에서 언급이 되는 내용이므로 반드시 숙지하고 있어야 한다.**

**이 에러가 난경우 페이지 락을 걸지 못했기 때문에 당연히 OverlappedIO는 실패된다.**

**WSANOBUFF 에러가 발생하면, 검은화면이 뜨게되고 아무것도 제어할 수 없는 상태가 된다.**

(더미테스트에 IOCP에 PQCS만 넣어도 Nonpaged-Pool Memory는 상승곡선을 그리게됨)

일반적인 상황에서 recv/send 1회씩 제한하고, TPS자체도 안정적이고, IOCP를 잘 사용했다면 괜찮을 것.

(TCP에서 TIME\_WAIT이 남는 것 자체도 리소스를 계속해서 남기는(NonPaged Pool의 사용) 현상이므로 위문제와 비슷하다.)

**WSANOBUFS Error 방지**

우리가 1byte의 메모리를 할당받았다 하더라도 페이지 단위로 락이 걸리게 된다.

**1. 버퍼를 페이지 단위로 간다.**

하지만 우리는 링버퍼를 사용하기 때문에, 불가능한 방법이다.

**링버퍼를 4KB로 잡는다고해도 시작점을 기준으로 온전히 4KB인지는 알수없고, 이를 맞추기위해 힙의 구조를 포기하는 것은 바람직하지 못함**

**2. 동접자를 낮추거나, 버퍼크기를 줄여 사용되는 메모리양 자체를 줄이도록 한다.**

링버퍼의 크기가 10000byte라면 페이지를 3개나 사용하고 있을 것이다.

**3. 물리메모리 자체를 늘려 메모리양 자체를 늘려보는 것.**

이외에는 딱히 뾰족한 방법이 없을 것.

**우리가 단지 GQCS에서 뽑지않는다고 하여 락이 안걸리는 개념이 아니다.**

**어쨌든 오버랩IO작동이끝나서 완료통지로 넘어가야 락이 풀리는 것이다.**

구체적으로 본다면,

**Recv는 항시 걸려있어야 하므로 OVERLAPPED IO로 5000명을 받았다면 5000\* RecvRingBuffer만큼 차지한다.**

**추가로 send까지 OverlappedIO로 중첩되는 경우를 생각해야 할 것.**

**2. non-paged pool**

**Non-paged pool은 애초부터 물리 메모리에 상주하며 페이징 되지 않도록 만들어 놓은 영역이다.** Fage-fault가 없으므로 접근속도가 빠르며, 고레벨 IRQL(Interupt Request Level)접근이 오류 없이 가능해진다.

그렇다보니 드라이버나 프로그램들이 non-paged pool을 남용하여 사용하게 되고, 제한된 리소스는 손쉽게 고갈되어 문제가 발생하게 된다.

**non-paged pool 사용하는 경우**  
  
① **드라이버와 같은 커널모드 컴포넌트에서 사용.** 윈속과 tcpip.sys와 같은 프로토콜 드라이버도 이에 속한다.  
② **소켓을 생성시 상태 정보를 저장하기 위한 용도로 작은 양의 non-paged pool이 소비.**  
③ **소켓이 특정 주소로 바인딩되면 TCP/IP 스택은 로컬 주소 정보를 저장하기 위한 용도로도 non-paged pool을 할당.**④ Overlapped I/O연산시 IRP (I/O request packet)를 발생시키면서 약 500바이트의 non-paged pool 할당.  
  
  
보통 non-paged pool은 windows 2000 이상의 버전에서는 물리메모리의 4분의 1이 한계이다.  
**비동기 작업시 page-locking되는 메모리에 비해 사용되는 곳이 더 많으며 한계치에 다다르는 경우 문제 해결이 어려워진다.**

**Non-PagedPool이 부족할 때 나타나는 증상**

**1. 운이 좋을경우 윈속함수가 WSAENOBUFS에러를 발생시킨다.**

**2. 시스템 에러로 손상되어 우리가 아무것도 제어하지못하는 검은화면이 뜨게된다.**

**우리는 항상 Non-PagedPool Memory를 모니터링 클래스로 모니터링 할 것.**

(여기서 Non-PagedPool메모리가 물리메모리 양에 비례해서 늘어나는지는 정확하지 않다.)

**Non-PagedPool의 특징**

Non-PagedPool의 정리는 C#의 Garbage Collecter와 비슷하다.

**C#의 경우 포인터도 없고 동적할당 한 메모리는 별도의 스레드가 알아서 정리해 주는 구조이다.**

**이때 어떤 행위를 중단한다고해서 그 즉시 반환되는 것이 아니라 한박자 늦게 정리하므로, 계속하여 사용한다면 메모리가 늘어나기만하고 줄어들지 않을 것.**

((

Non-PagedPool에 대해 우리가 알수있는 정보들

Sysinternel, ClockRes, TCP view 등..

**이중에 PoolMon이라는 Nonpaged를 모니터링하는 프로그램이 존재하는데, 이것으로 확인한다.**

PoolMon은 어떤드라이버 코드, TCP, IOCP 몇몇 파트에서 Non-PagedPool이 변화량이 어느정도있지 보여주며, 이 Non-PagedPool의 변화량은 바로바로 반환이 안된다.

우리에게 공개된 정보는 이정도 밖에 없음

))

**3. zero byte recv**

우리가 쓸대상은아닌데 MS의 windows Network Programming에서 언급되므로 기억해둔다.

(이렇게 할바에는 그냥 Select를 쓰는게 낫지않을까.. 하는 의문이 남음)

**언급했든 Overlapped I/O 작업의 경우 page-locking + non-paged pool의 소비가 일어난 다는것을 알 수 있다.**

**이때 MMORPG서버에서 이 리소스의 소비를 줄이는 방법 중 하나로 Zero Byte Recv가 있다.**

**수신버퍼를 0으로 만드는것이아니라, Recv의 WSABUF의 Len을 0으로 넣는것이다.**

**일부러 recv의 인자인 WSABUF의 len을 0으로 넣어서, 완료통지에서 transferred가 0으로 떨어지게 한다.(recv한 크기가 0)**

**(WSABUF를 0으로 만든것으로, 수신버퍼의 크기는 비동기 환경이라 하더라도 0으로 만들지 못한다.)**

**OVERAPPED I/O를 쓰는 입장에서는 언제 Recv를 해야하는지 모른다.** (계속 Recv를 걸어놨음)

(Select 경우 수시로 falling하여 Recv를 해야하는지 판단했음)

**이경우 완료통지가 온 것은 수신버퍼까지만 들어오고 우리쪽으로 복사하지 않은 형태이다**

**이때 recv완료 통지가 올 때 recv를 걸게되면 수신버퍼에 데이터가 있기 때문에 IOPENDING이 나지않고 동기로 처리가 될 것.**

**(이는 마치 AsyncSelect에서 Read를 걸어두면 Read로 오는것과 같다.)**

((

**WSAOVERLAPPED라는 구조체 포인터를 넣고 오버랩IO로 희망하는 상황으로 요청을 내부에서 동기로 처리됐든 비동기로 처리됐든 결과에 대한 완료통지는 온다.**

IOCP에서는 내부에서 동기로 작업됐는지 비동기로 작업됐는지 구분하지않는다. 모든 마무리작업을 workerthread에서 하도록 코드를 단일화 시키기때문이다.

이때, **성능적인 입장으로 본다면, 만약 동기로 작업이 됐다고 치면 굳이 이걸 워커스레드 IOCP완료통지가지 넣고 워커스레드 깨워서 할필요가 없다고 생각할 수도있다.**

이를 위해 동기로 작동됐을 때 완료통지가 오지않는 옵션이 존재한다.(via에서 언급.)

**대신 이 경우에는 성능을 조금더 극대화시킬수있지만 구조적으로 좋지않아서 사용하지는 않는다.**

(써도 상관없지만 굳이 이렇게까지 사용할 필요는 없음)

))

**Zero byte Recv를 쓰는 이유**

결국은 ProActor였던 OVERLAPIO를 버퍼를 0을전달해 Reactor로 바꾼 형태이다.

**이와 같은 형태를 취하는 이유는,**

**1. 오버랩IO의 IOCP구조는 좋지만 오버랩IO의 Page-Locking, NonPaged-Pool등의 리소스 할당 문제 때문에 쓰기가 싫고, Recv의 상황은 바로바로 파악하고 싶은 경우**

**(당장 받을 데이터가 없는데도 불구하고 OVERLAPPED I/O에서 Recv가 비동기로 걸려있는 모양새가 싫은 경우, 버퍼를 너무 많이, 크게 잡아야 할 때 리소스 문제가 생길 가능성 등 모두 리소스를 사용하고 있는 문제이다.)**

**2. IOCP의 구조를 그대로 쓰되, 동기IO를쓰고싶을 때.**

어차피 비동기든 동기든 CPU소모량은 같고, 일반적인 게임서버의 모양새는 WorkerThread가 바쁘지않으므로 동기화를 하겠다는 뜻이다.

(비동기 IOCP와 Zero byteRecv 어느것이 더 좋을지는 테스트를 거쳐 검증하여 판단해 볼 것)

**단, Zero byte Recv는 로직상에서 Recv에 대한 반환값이 0이 떨어진다고 하여 종료처리를 할 수 없기 때문에 코드가 매우 복잡해지고, 잘못 만드는 경우 엉망진창이 되기 때문에 충분히 고민하고 시도해야 한다.**

**ZeroCopy**

제로카피는 Copy를 하지말자는 뜻이다.

**TCP스택 | 송신버퍼(커널) | 서버SendQ | 로직Packet**

위구조에서는 각 계층마다 카피(Memory Copy)가 계속해서 일어날 것이다.

**컨텐츠쪽에서 (구조체or직렬화버퍼)에 데이터 셋팅 -> SendQ링버퍼에 Copy -> Send를 통해 송신버퍼에 Copy -> TCP스택은 이를 뽑아 실질적인 Send**

**ZeroCopy는 로직Packet에 있는 데이터를 TCP스택으로 한번에 보내자는 의미이다.**

(링버퍼까지는 아니더라도 기본적으로 제로카피는 송신버퍼를 뛰어넘자는 의미)

송/수신버퍼는 우리가 조절가능 했으나, 우리가 요청하는 모든 것을 그대로 들어주지는 않았다.

**수신버퍼를 0으로 만들경우 윈도우사이즈가 0이 되므로 알아서 늘렸고, 송신버퍼의 경우도 다시금 늘어났다.**

**하지만 OVERRAPED I/O에서는 송신버퍼를 0으로 요청할 경우 그대로 반영된다.**

송신버퍼를 0으로 만든다면 WSASend는 100% WSA\_IO\_PENDING상태에 빠지게된다.

**이후 준비된 버퍼에 해당 포인터를 넣고 스레드를 호출시키면 송신버퍼에 공간이 없기 때문에 TCP스택이 서버SendQ에서 데이터를 뽑아간다.**

**이때 Page-Lock이 일어나서 서버SendQ가 속한 메모리에 락을 걸고 TCP스택에서 직접 뽑아간다.**

**이는 우리의 메모리가 송신버퍼역할을 하게된 것.**

**Zero-Copy는 결국 100% OVERLAPPED I/O를 유도시키는 방법으로, 리소스를 아끼기 위해 어떻게든 비동기를 피하려는 Zero-ByteRecv랑은 완전히 반대되는 상황이다.**

(동기로 진행하는 ZeroByte-Recv는 리소스를 아낄수있음. 비동기인 경우 페이지를 물고있어야함)

로직Packet 컨텐츠쪽에서 메시지가 만들어졌고, 이를 바로 WSASend에 꽂아버리기 위해서는 힙에 존재하는 메모리에 Locking이 걸린다는 얘기이고, SendQ도 링버퍼가 아닌 직렬화버퍼 포인터를 받는 모양새가 되어야 한다.

**구체적으로는 WSASend할 때 직렬화버퍼를 모두 뽑아 버퍼 포인터들을 WSABUF에 모두 셋팅하고,**

**WSASend를 호출시키면 로직Packet에서 만든 버퍼를 TCP스택까지 보내는 구조.**

**우리의 메모리가 page-out 되지않도록 Lock을 걸기 때문에, 여기저기널려있는 힙메모리가 페이지단위로 Lock이 걸릴 것이다.**

**TCP스택 | 송신버퍼 | 서버SendQ(링버퍼) | 로직Packet(Msg)**

로직Packet에서 서버SendQ의 카피는 없앨건데,(SendQ자체를 Rock-free로 바꿔 직렬화버퍼에 꽂을 예정) 서버SendQ에서 송신버퍼로의 카피는 테스트를 해보고 판단을 해볼 것.

그리고 어쩔 때 쓸만한지. 이건 여러분이 테스트를 꼭해보셔야 해요. 그리고 테스트가 끝나고 나면 결론이 날것이다.

나중에는 링버퍼도 없앤다. 근데 이거는 난이도가 너무높아지기 떄문에 여기서는 진행하지않을것.

**2022추가**

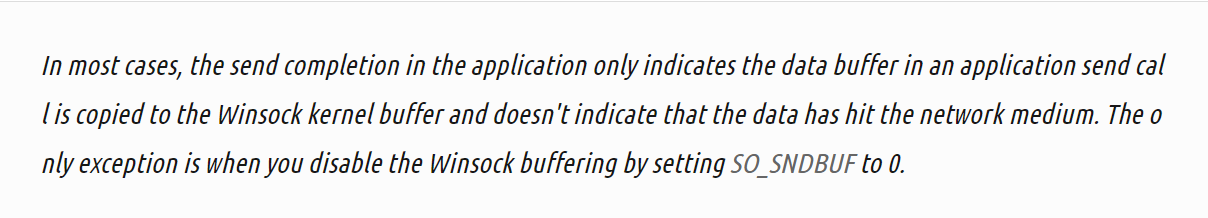
**ZeroCopy에 대해..**

윈도우소켓함수 Send()는 단지 유저버퍼에 있는 데이터를 소켓송신버퍼(커널버퍼)에 복사 해줄 뿐이다. 데이터 전송은 OS가 알아서 해줌.

TCP는 혼잡제어, 흐름제어(윈도우)같은 전송량 제어 프로토콜을 따라야 하기 때문이다.

콜백함수나 IOCP큐로 통지받는것은 유저버퍼의 데이터가 소켓의 송신버퍼에 복사된것을 의미한다.

<https://docs.microsoft.com/en-us/troubleshoot/windows/win32/data-segment-tcp-winsock>



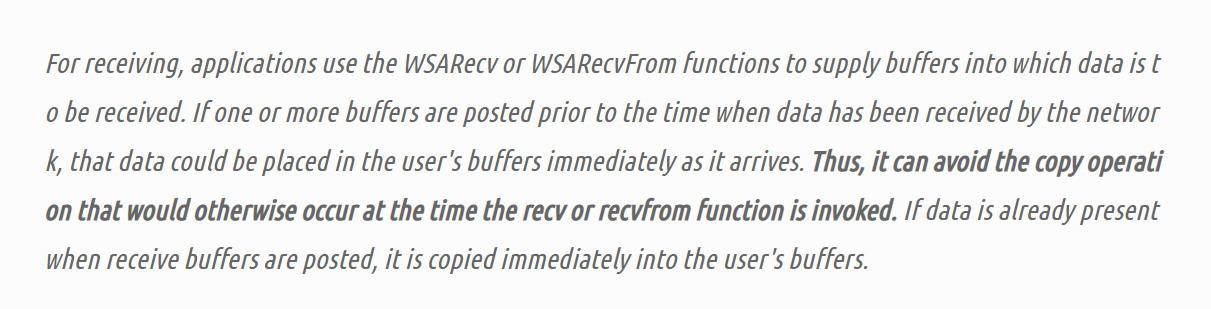
만약 송신버퍼(커널)를 0으로 만든경우, Send완료(IO\_PENDING이 아닌 복사완료를 뜻함)는 네트워크 패킷이 복사되었음을 의미. 상대방이 받은거랑은 다른 얘기다.

OS가 인식하는 수준에서 케이블또는 와이파이로 목적지를 가기위한 다음 경유지로 패킷이 나갔음을 의미한다.

(이전에는 송신버퍼에 복사하는것이 Send의 완료였음)

zero copy에 대한 내용이 IOCP와 OverlappedI/O를 거론한 글에서 많이 나옴.

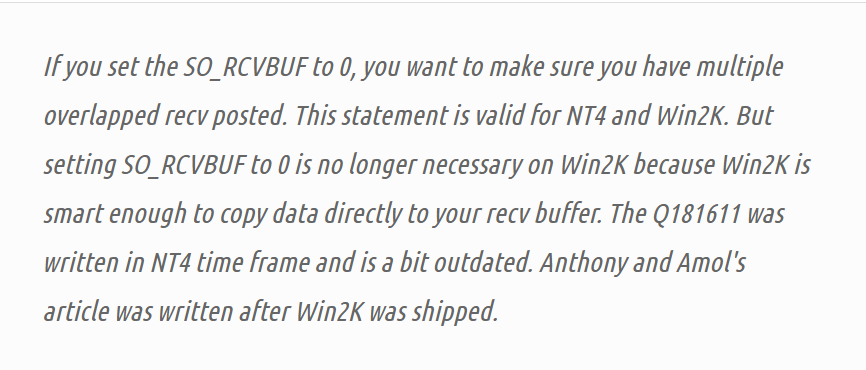
<https://docs.microsoft.com/en-us/windows/win32/winsock/overlapped-i-o-and-event-objects-2>



수신의 경우 수신버퍼의 사이즈가 어떻게되든 상관없이 WSARecv를 통하여 제공한 버퍼에 복사.

그럼 수신버퍼 사이즈0으로 만들어 카피를 줄이라는건 틀린말? ㄴㄴ.

<https://groups.google.com/g/microsoft.public.platformsdk.networking/c/0RZYJpWLKnw>



수신버퍼(커널)를 0으로 만드는것은 과거의 커널에서 의미있는 이야기였지만, 현재 기준으로는 커널에서 똑똑하게 처리해준다.

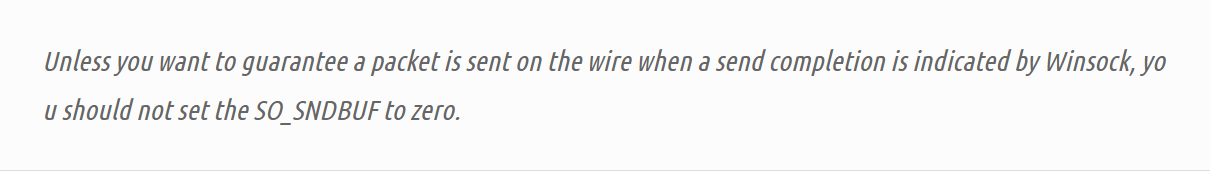
단 소켓수신버퍼를 0으로 만드는것이 아닌, WSABUF len을 0으로 전달하면 논페이지드 풀 메모리를 아낄 수 있다. (메모리 효율성증가)

(위에서도 언급했듯이 이럴바에는 그냥 Select를 쓰는게낫지않나?)

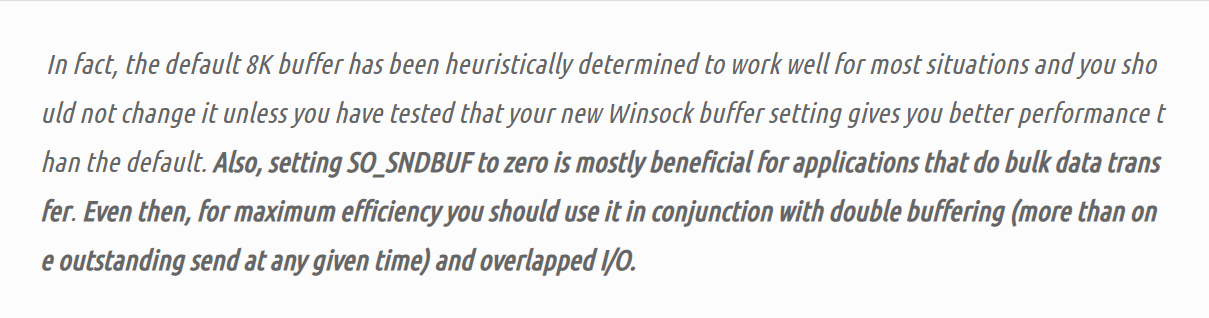
그럼 송신버퍼를 0으로 하는건?

일단 논페이지드 메모리를 많이 쓰는건 제쳐놓고, 성능향상이 있는가?

이 경우 SEND의 완료통지는 송신패킷이 네트워크로 나갔음을 의미한다.



이 한번의 카피를 줄여주는게(줄지 않을수도있음. 2008 서버쯤부터 소켓버퍼 사이즈를 동적으로 조절하는 내용이 추가된 것이 있기 때문). 퍼포먼스 향상이 있으면 있다고 얘기를 할텐데, 그 부분은 빼고 예외적으로 성능향상이 일어날 수도있는 경우를 게시했기 때문.



***"more than one outstanding send at any given time"*** 이 의미하는 것은 송신 완료 통지를 기다리지 말고, 일정 숫자이상(테스트를 통해 최적의 횟수를 구해야 함) 송신 요청(WSASend 호출..)을 계속 걸어두라는 것이라고 추측한다.

결론은 버퍼 사이즈 변경하지 말고, 변경하고 싶으면 성능 테스트를 할 수 있는 환경을 갖추고 테스트를 하라는 것

(리눅스는 아예 zero copy를 지원하는 API가 있다고함)

++추가

송신버퍼를 0으로하면 송신버퍼 복사후 완료통지 하던것에서, TCP레이어쪽으로 복사완료되면 완료통지 되게된다.

이 경우 TCP레이어는 송신이 완료됨을 보장해야 하기 때문에 대기해야된다. (느려짐)

수신인 경우 커널버퍼가 있으면 수신된 패킷을 커널버퍼에 보관하면됨.

수신커널 버퍼를 0으로 하면 보관할 버퍼가 없으므로 해당 패킷을 드롭해버림.

이때 recv를 통해 유저버퍼를 세팅했으면 드롭하지않아도 되니까 유저버퍼에 기록.

++추가

<https://developer.ibm.com/articles/j-zerocopy/>

송신버퍼를 0으로 했다. -> 논블락인 경우 (동기든 비동기든간에) 요청한 데이터가 전부다 카피되었는지 알수없는 경우가 생김.

때문에 0으로 잡게되면 해당데이터를 nic에 다 실어 보낸경우에 송신이 완료도었다고 말할수 있다.

근데 송신버퍼가 0이아니면 일단 커널에서 유저버퍼내용을 긁어갔으면 나중에 보내줄거니까 일단 됐다고 알려줄수 있음.

zerocopy가 이득을 보려면 GQCSEx를 사용할때만 이득. (테스트해봐야함).

하나씩 보내야 하는경우는 매우 비효율적

++뇌피셜

여기저기 걸려있는 CMsg를 만약에 Copy한다..이것은 캐시미스.

**<7.17>**

지난시간에 대한 내용 복습

**WSASend/WSARecv를 1회로 제한하게된 잘못된 지식**

**1. 중첩해서 걸었을 때 우리가 요청한 값보다 적게 나오지않을까?**

Recv/Send의 반환값이 우리가 요청한 값보다 적게 반환이 될 경우는 절대 없다.

**완료통지는 요청한 데이터 크기를 다 하는 것이 완료.**

**단한가지 예외로, 소켓에 연결이 끊겼을 경우는 요청한값 미만으로 올수있지만 이 경우는 어차피 끊을 대상이기 때문에 고려할 대상이 아니다.**

때문에 어쨌든 중첩해서 Send하는 것은 기능적으로 전혀 문제가 없다.

**2. 완료에 대한 순서보장이 안되지 않을까?**

**완료에 대한 순서는 보장되지만 완료통지에 대한 순서보장이 되지않는다.**

소켓을 대상으로한 IO작업과 FILE I/O작업은 작동방식이 완전히다르므로, 혼동할 이유가없다.

**WSASend/WSARecv를 1회로 제한하는 이유**

**1. 또한 WSAOVERLAPPED 구조체 여러 개를 관리해야하고, 중첩할때마다 지우는 것은 바람직하지않은 형태.**

**2. 결정적으로 WSASend같은 IO작업을 여러 번 호출하는것에 대한 장점이 전혀 없다.**

**결론은 예전에는 잘못된 지식때문에, 지금은 장점이 없기 때문에 1회로 제한한다.**

**실제로 내부에서 처리된 IO방식**

**WSARecv / WSASend의 결과로 WSA\_IO\_PENDING이 아닌 ERROR가 떴다면 연결이 끊긴것으로 처리한다.**

**리턴이 성공한 경우라면 (지금당장 요청한 일을 수행할 수 있을때)내부적으로 동기IO로 작동된 것이다.**

**WSARecv – 수신버퍼에 이미 데이터가 있을 경우 동기로 COPY.**

**WSASend – 송신버퍼에 공간이 있다면 동기로 COPY.**

**위와 같은 이유들로 우리는 Overlapped I/O를 사용하고 있긴하지만 WSA\_IO\_PENDING이 자주 발생하지않는다.**

이는 오버랩IO의 장점을 딱히 사용하지 못하고 있다고도 말할 수 있다.

이렇게 동기가 빈번한 경우 훨씬 가벼운 send/recv를 사용하는편이 낫다.

**(언제는 Send/Recv그냥 쓸때도 WSABUF로 몰아주니까 이거쓰라매..)**

**Overlapped I/O 오버헤드**

**WSASend, WSARecv가 내부에서 비동기로 작동이 됐을 때, Copy중인 메모리는 UserMemory이고, Page-Out될 수 있기 때문에 Page-Locking된다.**

(여기서말하는 Page-Locking은, read/Write가 아닌 Page-out되지못하게 lock을 거는 것)

**1. WSASend함수는 send작업을 위해 Page-Lock, 정보셋팅, 내부에서 Io작업등의 오버헤드가 발생하고, 이 작업이 Copy하는 것보다 더 오랜시간이 걸릴 수도있다.**

(여기서 메모리를 Page단위로 관리를 하고자한다면 우리가 사용하는 heap을 포기하고 virtualAlloc을 통해 직접 메모리를 관리해야하므로 이는 매우 난해한 작업이 된다.)

**또한 더 이상 락을 걸수없는 메모리 한계에 도달한 경우 이는 WSANOBUFFERROR라는 심각한 에러를 초래하는 결과가 나올수도있다.**

**Zero Copy**

**일단 IOPENDING이 발생되서 오버랩IO로 전환 됐다는 건 소켓의 경우 우리가넣은 버퍼를 송신대용으로 사용하는 것이다.**

**이를 유도하기위해 억지로 송신버퍼를 0으로 만들어 IO\_PENDING을 발생시키는 것.**

**Send같은 경우 대부분 wirte가 가능한 상황이므로, 거의 대부분 WSA\_IO\_PENDING이 발생하지않는다.**

(원래는 내부에서 마음대로 늘어났지만 OVERLAPPED IO의 경우에는 유지시켜준다.)

**Send는 ZeroCopy를 유도했을 때 백그라운드에서 Copy가 되는 Overlapped I/O의 이점을 얻을 수 있을것이다.**

**Recv는 항시 걸어놓는데 수신버퍼에 데이터가 없을것이므로, 대부분 IO\_PENDING이 들어간다.**

**Recv는 수신버퍼를 0으로 만들 이유도 없고 기능적으로도 불가능하다. 이렇게 하지않아도 ZeroCopy가 지원된다.**

(Recv가 와서 로직을 처리하고 다시금 Recv를 걸어 항시 Recv를 유지하게끔하는데, Recv를 1m/s의 틈도없이 걸어놓을 수는 없다.)

수신버퍼는 0으로 크기를 조절하는 것이 OVERLAPPED I/O를 사용한다고 해도 알아서 늘어나며, Zero byte Recv같은 경우에는 WSABUF의 Len길이를 0으로 넣는 것.

(실제로 Setsokopt()를 사용해 RecvBuffer를 0으로만들고 getsokopt()를하면 0으로 나오기는하지만, Recv받은 다음 WSARecv를 걸면 IOPENDING이 뜨지않고 Copy가 진행된다.)

**Session**

**WSASend, WSARecv를 할 때 링버퍼를 사용하므로, 이는 보존되어야 할 대상이다.**

Session

{

RecvQ

SendQ

OverlappedSend,

OverlappedRecv

}

하나의 세션을 대상으로 1회의 WSARecv, 1회의 WSASend가 들어가므로 총 2회의 IO가 들어간다.

**GQCS호출 후 종료처리**

(GQCS호출 후))

If(transferred == 0 || ERROR)

{

CloseSocket()(세션->소켓);

SessionRelease()(자료구조에서 지운다)

}

이때, 하나의 세션을 대상으로 1회이상의 OVERLAPPED I/O가 요청들어가는 상황이므로, 하나의 세션이 이상이 있다고하여 지울수 있는 상황이 아니다.

하나의 Session에 Recv가 걸려있다고 가정한다.

이때, WSASend()후에 CloseSocket()를 호출한다면 WSASend에 대한 완료통지는 어떻게 올것인가? 이는 상황에 따라 다르겠지만,

**Send의 성공이후 Recv의 연결끊김이 올수도있지만, Recv에서 연결끊김이 뜨고나서 Send의 성공이 오는 상황이 나올수도 있다.**

**IO가 두개이기 때문에 완료통지역시 두개가 걸린다.** (완료통지가 없어지는 경우 구조적 결함)

이때 완료통지가 왔다고 세션을 지워버렸다면 로직이 엉망진창이 될 것.

사실 어떤소켓이 끊겼다면 내부에서도 끊긴것이다. closeSoket하는 순간 모든 IO는 중단된다.

**따라서 작업 자체에 뭔가 문제가 나는 것은 아니지만, 우리의 컨텐츠 코드에서 문제가 발생한다.**

**컨텐츠 코드차원에서의 문제**

**1. 세션자체가 삭제되서 메모리 참조 오류가 날 가능성이 존재한다.**

**2. CloseSocket하는 순간 accept에서 해당값을 재사용하기 때문에 다른 소켓을 사용해서 엉뚱한 애가 응답을 하게 될 수도있다.**

**따라서 우리가 해야할일은 모든IO가 끝났는지 판단하여 세션을 지우는 여부를 판단하는 것이다.**

**세션을 지우기 위한 방법**

**1. 완료통지로 종료가 오더라도 삭제를 하지않는다.**

**이렇게 진행할 경우 처리가 모두 끝나고 다른쪽에서 별도로 정리하는 행위를 할 때, Accept쪽에서 바로 재사용할 수있는 위험성이 존재한다. 이는 사실 막기 좀 애매한 부분이다.**

**이 방법에 남아있는 단점**

1-1. 연결종료 완료통지 이후 거의 찰나에 Send완료통지가 올것이므로, 턴을 가지고서 그 다음 완료통지가 뜰 가능성은 거의 없다. **따라서 길게 물고갈 이유가 없음.**

1-2. 또한 여러 개의 워커스레드중 우연찮게 동시에 두개가 깰수도있음.

**이 방법을 사용하는 이유**

**위 문제를 해결하기 위해서는 세션을 재사용하는 시간자체를 조금 벌어주는것으로, 문제를 완전히 없애주지는 않지만 가능성을 0에가깝게 만들어 준다.**

(단순한 방법으로는 일부러 소켓 재사용시 큐방식을 사용하는 것)

이는 실제 라이브서비스에서도 많이 사용하는 방법이나, 100%가 보장되지않기 때문에 권장하기는 어렵다.

**2. IO가 모두 끝났는지 우리의 코드로 직접 확인한다.**

**가장 확실하고 고전적인 방법으로, 세션내부에 변수를 둬서 IO자체를 카운팅 하는 것이다.**

IO를 걸때마다 IOCount를 증가시키고 IO가 완료되면 차감시킨다.

**IOCount가 0이됐을때가 Release의 시점이 되는 것.**

정상적인 상황인 경우 항시 Recv를 걸고가기 때문에, IO가 0이된다면 재등록하지 못한상태이다.

따라서 이를 기반으로 간다.

(이변수는 Interlock을 사용할 것)

**RecvPost() / SendPost**

WSARecv, WSASend를 랩핑한 것으로, 여기서는 WSARecv로 설명한다.

우리가 WSARecv를 걸던 작업을 따로 뺀것이다.

**WSAOVERLAPPED구조체 Recv, WSABUF를 만들어 RecvQ 포인터를 두개꼽아(링버퍼는 진짜 원형인 형태가 아니므로..) 셋팅하고 WSARecv를 건다.**

Ovelapped초기화

WSABUF 생성

Ret = WSARecv()

If(ret == SOCKET\_ERROR)

에러가 IO\_PENDING아닐떄..실패. WSARecv를 걸지 못했을 때.

(IO\_PENDING인지 아닌지 우리는 구분하지않는다. 동기든 비동기는 마무리작업을 통일시킨다.)

만약 WSARecv가 실패하면 완료통지가 오지않음.

**IO\_PENDING, 또는 성공으로 인해 작업이 들어가서 완료통지가 아주빠르게 온다면 WSARecv가 채 리턴되기도전에 워커스레드가 깨어날 수도있다.**

**따라서 IOCount는 등록하기 전에 증가시키고 들어간다.**

물론 실패(IO\_PENDING이 아닌 ERROR일 때 는 IOCount는 차감된다)

SendPost도 똑같은 형태로 진행된다.

**선증가 후, 등록이 실패하는 경우에는 차감시킨다.**

Session자료구조는 리스트로 관리한다. 최종적인 모델은 배열을 쓸것이지만 그전에는 배열이 아닌 리스트나 컨테이너를 대상으로 하는 구조를 사용한다.

RecvPost SendPost은 세션의 멤버를 인자로 넣어도되고, 아니면 외부에서 줘도된다. 편한대로 할것

이제 모든 WSASend, WSARecv는 RecvPost SendPost로 통일시켜서, 예외처리도 한번에 간다.

**MainThread**

메인스레드는 현재 딱히 할일이없으므로, 루프돌면서 기다린다. 키보드 입력을 추가해도 좋다.

**AcceptThread**

AcceptThread에서는 블락소켓으로 accept하고, 세션을 생성해서 자료구조에 추가한다.

만약 AcceptThread를 닫고싶다면, ListenSocket을 Close해버리면될 것.

(스레드는 우리가 임의로 강제종료 해서는 안되기 때문)

**WorkerThread**

GQCS

종료, GQCS,에러처리(세션에 대한 에러처리가 아니라는것임)

**If(세션 종료처리) { if(IOCount == 0) }**

**If(세션 Recv완료통지)**

**If(세션 Send완료통지)**

**이때 정석은 완료통지가 왔을 때 해당 Queue에 포인터를 이동시켜야 한다.**

Recv는 빈 버퍼를 모두 잡아줄 것이기 때문에 딱히 문제가 없음.

(1) Send의 경우 Send완료통지가 왔을 때 front를 이동시키지않고,

(2) Send를 WSABUF에 담고나서 그 즉시 front를 이동시킨다면?

**Send에 새로이 Enque될 때, 해당 데이터가 덮어씌워질 수 있으므로 이는 위험한 방식이다.**

**CloseSocket**

SessionRelease와 연결끊기는 다른 개념이다.

연결종료가 왔다면 내부에서 연결이 끊긴 것.

**주의해야 할점은, 두개의 IO에서 두개의 연결종료가 나오는 경우 CloseSocket을 사용한다면 한번만 하도록 로직을 처리해야 할것.**

위 경우가 대비되지않는다면 곧바로 재사용된 Handle에 의해 엉뚱한 세션이 끊길 수 있다.

**Shoutdown이용**

CloseSocket을 한번만 처리하도록 로직을 짜는 것이 싫다면, 다른대안으로 연결종료 절차를 shoutdown으로 해도된다.

Shoutdown은 리소스반환이 되지않아서 소켓이 재할당되지않으므로, 평소에는 Shoutdown으로 연결을 끊고 리소스는 Release할 때 closeSocket하겠다는 뜻이다.

(Recv 0을 받는 것이 FIN을 받는상황이다.)

**Shoutdown을 이용했을 때 문제**

클라에서 먼저 끊은 경우는 상관이 없으나,

서버쪽에서 먼저 끊은 경우

**1. TIME\_WAIT이 남는 문제.**

4Way-handshaking으로 인한 graceful shutdown이 진행되어, 서버쪽에 TIME\_WAIT이 남게된다.

**2. 서버쪽에서 FIN을 보냈는데 상대방이 FIN을 주지않는경우**

**결국 서버에서 먼저 끊어내려는 소켓들은 문제가있는 클라이므로 완전히 끊어내야 하는데 FIN을 주지않아서 끊어지지않는다.**

**Shoutdown으로 연결끊기를 유도하는 경우 closesocket을 1회로 제한하는 것은 간단하게 해결되지만 클라쪽이 FIN을 주지않는다면 끊어낼 방법이 없어지므로, 사용할수없음**

**우리가 간단하게만든 서버/클라에서는 돌겠지만 라이브에서는 끊기지 않는애들이 쏟아져 나올 것이다.**

결국은 CloseSocket을 1회로 하는 방향으로 해야할 것.

내부적으로,

서버측에서 FIN을 쐈는데도 아무런 응답이없다면(ACK를 받지못함) 점차 시간을 늘려가며 재전송하다가 결국엔 끊어버릴것이다.

**CancelIO**

BOOL WINAPI CancelIo

(

\_In\_ HANDLE hFile

);

지정된 핸들에 대해, 해당 스레드가 보류중인 모든 IO작업을 취소한다.

(IO작업이 Overlapped I/O인 경우)

만약 다른스레드의 I/O작업을 취소하려면 CancelIoEx()함수를 사용해야한다.

함수가 성공한다면 반환값은 0이 아니다.

함수가 실패한다면 반환값은 0.

**종료방법 - IOCount차감**

If(세션 종료처리) { if(IOCount == 0) }

**If(세션 Recv완료)**

**{ 받은메시지 처리Proc, 다시 Recv를 걸기위한 RecvPost }**

**If(세션 Send완료)**

**{ Send마무리 후 SendPost() }**

**Session Recv가 완료됐을때는 받은메시지에 대한 Proc가 돌아갈 것.**

**이후 RecvPost가 들어간다.(Recv는 항상걸려있어야 하므로)**

Session Send가 완료됐을때는

보내기 마무리하고 SendPost. (이후 언급)

If(세션 종료처리) { IOCount == 0; }

If(세션 Recv완료)

(ㄱ)

If(세션 Send완료)

**이 때 (ㄱ)구간에서 IOCount가 차감후에 증가가 들어가므로, 정상적인 세션임에도 불구하고 IOCount가 0이 되는 순간이 나올 수 있다.**

**연결 종료와 SessionRelease**

**연결을 끊는것과 IOCount가 0이됐을 때 Session을 Release하는 행동은 완전히 별개이다.**

**따라서 소켓에 IO Error가 발생했든, 연결이 끊겼든 Session Release는 IOCount가 0일 때 진행한다.**

**IOCount차감 기준**

**만약 선차감 후 로직으로 들어갈 경우, Send,Recv 두개의 작업이 걸려 워커스레드 두개가 동시에 깼을때는 IOCount는 0이되기 때문에 판단기준이 서지않는다.**

**결국 내IO작업을 물고가야 되기 때문에, 차감자체를 마지막에 하고, 차감과 동시에 IOCount가 0인지 체크한다.**

**SendThread**

일단 우리는 SendThread를 만들지않고, 보내고싶을때 아무데서나 Send를 시도한다.

지금이 어느로직이든 보낼게있다면 WSASend를 호출할 것.

이 경우 IOCount에 관련해서 문제가 발생하게된다.

**SendThread가 없을 때(아무데서나 WSASend할 때) 생기는 문제**

**지금처럼SendThread가 없는 상황이라면 Workerthread가 아닌 전혀다른 외부에서 호출하여 IOCount를 증가시킬 수 있는 상황이 된다.**

**만약 세션이 끊겼고 완료통지가 모두 도착하여 IOCount가 0이되면, IOCount == 0이라는 로직을 타면서 Release를 타야하는데 이 찰나에 다른스레드에서 Send를 하게된다면 IOCount는 다시 1이되고 로직을 통과하게 된다.**

**따라서 IOCount가 0이될수있는 코드는 차감하는 곳이므로, 모든 차감하는 부분에는 (IOCount == 0)가 들어가야한다.**

**RecvPost의 경우는?**

**RecvPost의 경우는 호출할 수 있는 스레드자체가 한정적이므로 위치자체가 제약이 있다. RecvPost를 호출하는 시점에서는 IOCount가 0이되는 일은 없음.**

**IOCP와 Select**

IOCP는 이런안전장치가 무조건 들어가야하고 많은 고민이 필요하기 때문에 어설프게 만든다면 Select보다 성능이 더 나오지않는다.

**세션자체의 안전성 확보**

최종적으로는 자료구조를 배열로 만들고, 락프리구조를 넣어 동기화가 없는 형태로 만든다.

지금은 리스트가 들어가므로 동기화가 걸릴수밖에 없다.

또한 세션자체의 안전성을 확보해야 한다.

**외부에서 다른스레드가 샌드할수있다는 전제는 우리가 워커스레드가 아닌 전혀 다른곳에서 건드릴수도 있다는 것이고, 0떨어지는 순간 Release로직을 탈것이기 때문에 세션자체를 락을 걸어야 한다.**

**SendThread를 쓰는경우**

SendThread가 없는경우 너무 복잡해지기 때문에 사용하는 경우도 있다.

간단하게 만들고자 한다면 동기화부분을 간단하게 만들수있다.

**삽입/삭제 동기화 문제를 해결하는 방법**

지금은 삽입과 삭제가 문제.

삽입삭제가 문제일 때 가장 간단하게 해결하는방법은 자료구조를 배열을 사용하거나, 삽입과 삭제를 같은곳에서 처리하면 된다.

이 같은 형태로 구조를 바꾼다면 accept구조도 바뀌어야 한다.

지금은 모든게 워커스레드 안에서 IOCount를 물린상태에서 진행도기 때문에 Release를 쓸일이 없다. 따라서 Session에 대한 동기화를 넣지않아도 문제가 일어나지않음.

**<7.20>**

소켓종료는 LINGER로 진행한다. Shutdown을 통한 graceful shutdown은 우리한테 해당사항이아니다.

LINGER ling;

ling.l\_onoff = 1;

ling.l\_linger = 0;

setsockopt(Socket1, SOL\_SOCKET, SO\_LINGER, (const char\*)&ling, sizeof(ling));

우리가 만들어볼 형태

OnClientJoin(SessionID)

OnClientLeave(SessionID)

--------------------------------

OnAccept()

OnLeave()

--------------------------------

OnRecv(SessionID, \*msg)

OnSend

--------------------------------

SendPacket(SessionID, \*Packet)

Disconnect(SessionID)

--------------------------------

SendPost

RecvPost

**On이 붙은경우 일반적으로 어떤역할을 다했을 때를 말한다. 무언가를 핸들링하기 위한 함수를 말한다. (네트워크 라이브러리 내부에서 호출될 함수들이다.)**

(OnRecv -> Recv가 끝났을 때 호출시킬 대상)

**위 함수들을 전역 선언해놓고, 우리의 워커스레드와 메인스레드에서 호출만하려는 목적이다.**

**추가적으로, 우리가만들 형태는 프라우드넷과 유사하게 만들 것이다.**

**SessionID추가**

우리가 최종적으로 만들어볼 형태는, 크게 세개의 파트로 나누는 것이다.

**[ 네트워크 소켓 API / 우리가만든 네트워크 라이브러리 / 컨텐츠파트 ]**

**SessionID는 컨텐츠 파트에서 소켓과 같은 역할을 하게 될것이다.**

**우리가 만든 네트워크 라이브러리에서 데이터확인/데이터보내기 등을 수행하고, 컨텐츠는 파트를 나누기 때문에 소켓을 사용하지 않아야 한다.**

**네트워크에 대한 모든 전담작업은 네트워크라이브러리에서 끝난다.**

(우리가 진행하는 에코서버의 경우에는 사실 SessionID가 필요없다. Recv한걸 바로 Send하면 그만이므로.)

SessionID는 세션이 들어올때마다 ++해주는 형태로 간다.

소켓값은 재사용되므로 이를 SessionID로 사용해서는 안된다. SessionID를 대상으로 뭔가 보내고 받기 작업을 할것이므로 로직이 꼬일 것.

일반적인 네트워크 환경에서는 host는 네트워크에 연결된 장치,

P2p게임에서의 host는 서버를 의미한다.

따라서 host란 용어는 애매하므로 SessionID로 간다.

**세션찾기 실패**

FindSession에서 세션ID로 세션을 찾지 못하는경우, 그냥 return false한다.

컨텐츠쪽에서 실수할수 있으므로, 이경우 서버를 죽여서는 안된다.

**OnClientJoin()**

Accpet가 반응이왔을때 기존에 했어야하는것들을 처리한다.

캐릭터 생성 등의 로직이 들어간다.

지금상황에서는 딱히 할일이 없다.

**OnClientLeave()**

모든것이 마무리되는 단계이다.

연결이 끊기고, 모든 리소스가 해제된다음, 최종적인 로직이 들어간다.

지금상황에서는 할일이 없다.

**OnRecv()**

OnRecv (SessionID, CMsg\*)

**세션 링버퍼 내부에 직렬화버퍼 자체의 길이가 들어가있다.**

**[ 길이(2), 데이터(n) ]**

**완성된 하나의 메시지를 판단하여 헤더를 제외하고 PayLoad만을 OnRecv에 전달한다.**

(우리는 지금까지 헤더와 컨텐츠헤더를 하나로 합쳐서 봤지만, 이제는 계층을 나눠야 하기때문에 네트워크 라이브러리인 헤더를 분리한다. 지금우리가 만드는 것은 5,6계층이며 7계층에 컨텐츠가 들어간다.)

지금은 에코에 대한 데이터가 정의되지않았으나, 실제 테스트하는것에는 에코를 정의하여 8byte값을 사용한다.

(그대로 받은패킷을 Send로 보내도 상관없긴함..)

**8byte, value;**

**Pacekt>>value;**

위와같이 마샬링해서 뽑아낸다음, SendPacket생성해서 다시넣는다.

**SendPacket;**

**Packet << Value;**

**SendPacket(SessionID, &SendPacket);**

**OnSend()**

Send의 경우는 짝을 맞추기위해 만든것으로, 지금시점에서는 할일이 없다.

**OnAccept()**

OnAccept는 accept함수와 비슷한데, 마치 SocketAPI에서 accept를 호출시키면 Socket뱉어내면서 셋팅되는형태이다.

**OnAccept에서는 세션을 마련하고, 새로운SessionID를 부여하여 리스트에 넣어줌으로서 컨텐츠 파트쪽에 새로운 Session이 들어왔음을 알려준다.**

(지금은 에코예제이므로 컨텐츠쪽에서 아무런 일을 하지않는다. 로그정도가 있을 것.)

결국 세션도 네트워크파트로서 관리하고있고, 컨텐츠쪽에서는 메시지만 전달하는것이다.

**OnLeave()**

단순히 소켓이 끊긴 것이 아니라 Release까지 모두 완료된 상태를 말한다.

마찬가지로 크게 할일이없다.

**On..함수들의 호출위치확인**

**OnRecv, OnSend, OnAccept는 모두 네트워크 라이브러리 내부쪽에서 호출되는 함수이다.**

**OnAccept**

Session확보가 끝난다음 AcceptThread안쪽에서 호출한다.

(PORT는 없으므로 생략한다.)

**OnLeave**

실질적으로 Session을 해지하기위해서 들어가는 함수인 SessionRelease()를 추가한다.

OnLeave는 SessionRelease()내부에서 호출한다.

OnLeave는 연결이 끊기고, IOCount가 0으로 떨어져서 Release해야하는 순간에 호출한다.

**Workerthraed()내부**

If(Recv완료통지가 왔을 때)

While

{

RecvQ, 완성된 메시지확인. (2byte헤더),

(헤더를 확인해서 페이로드 길이가 충분한지 확인하고 충분하다면 Cpacket, Packet, RecvQ->Packet메시지 카피)

OnRecv(SessionID, &Packet)

}

**SendPacket(), Disconnect()**

지금껏 On을 붙여 소개한 함수들은 핸들링을 하고자하는 어떤 이벤트의 발생을 알려줄 역할의 함수들이었다.

**이와 반대로 이 두함수는 컨텐츠쪽에서 요청이 들어오는 함수이다**

**(컨텐츠쪽에서는 소 켓,링버퍼 등 라이브러리 관련 리소스에게는 절대 접근불가)**

**SendPacket(SessionID, Cpacket\*); //컨텐츠쪽에서 Send를 요청하는 것**

**Disconnect(SessionID); //컨텐츠쪽에서 에서 연결을 끊는 것.(요청)**

Session을 생성하는곳은 리스트에 삽입할때.(이후 여기서 삭제할 일도 생길것이다.)

Workerthread는 Session을 사용, 삭제.

**Session 동기화이슈**

**1. 컨테이너 자체의 동기화 이슈.**

**컨테이너 자체의 동기화 문제는 락을 걸면 해결된다.**

(lock을걸고 삽입, 검색 시 lock걸고 검색, lock걸고 삭제.)

**2. Session 낱개의 동기화 이슈**

**특정 Session 하나를 접근해서 사용해야하는 상황이므로 컨테이너 자체를 lock을 걸고 사용하면 동기화 이슈는 발생하지않는다.**

**단, 이 경우는 병렬처리가 전혀되지않으므로 멀티스레드로 로직을 짜는 의미자체를 잃는다.**

**검색을 하는 경우에는 별다른 방법이없기 때문에 락을 건다. 검색 후 특정Session을 써야하는 상황에서는 다른쪽에서도 접근하여 사용할수있어야 하므로 락을 풀어야할 것.**

((

**에코서버에서는 일어나지 않는 문제**

지금 진행하는 형태(에코서버) 일 경우 이러한 문제는 일어나지않는다.

에코서버의 구조는 뭔가를 보내는 행동, 받는행동이 모두 Workerthread안의 GQCS완료통지를 통해서만 이루어지고 있기때문인데,

**완료통지가 왔을 때 우리는 할일을 다 하고난다음 IOCount를 차감시키고 0인지 확인하기 때문에 어떤 Session한테 데이터를 보내서 뭔가를 하는 동안에는 절대 IOCount가 0이되지않는다.**

만약 Recv,Send완료통지 도중 Release가 진행 된다면 잘못만든 것.

))

**결국 Workerthread안쪽에서 뭔가의 세션에 접근하는 코드들은 문제없고, 다른 스레드가 나왔을 때 새로운 스레드가 치고 들어갈때가 문제이다.**

다른 스레드가 언제 호출이 될지 우리는 알수없으므로 우리의 권한이 아니다.

**바깥에서 SendPacket을 시도하는경우 내부에서 SendPost(WSASend)가 호출되므로 IOCount가 차감되는 코드가 들어가고, 이때 IOCount가 차감되는 부분에는 if(IOCount == 0) 로직이 반드시 들어가기 때문에** **세션이 바깥에서도 지워질수 있는 상황이다.**

WorkerThread에서 특정Session에대해 IOCount가 0인 것을 확인하고 Release하려고 하는데, 다른쪽에서 검색중이거나, 검색을하고 사용하려고헀는데 Release하려고 할수도 있다..

**이중락으로 동기화 이슈 해결**

지금 시도하고자 하는방법은 우리가 최종적으로 완성하고자 하는 방법은 아니지만, 스레드의 개념/동기화를 이해하기위해 시도한다.

**먼저 list Lock을 걸고 검색하고, 세션을 사용하려고 할 때 list Lock을 걸어둔채로 세션을 잠근다음 listLock을 해제한다.**

(이는 교차로 lock을 거는 것이기 때문에 잘못 짤경우 데드락에 빠질 가능성이 높다.)

**삭제할때도 락을 걸어야 한다.**

listLock을 걸고 검색에 성공해서 SessionLock을 걸려고했는데 누군가 쓰고있다면..?

(동기화객체는 용도에 맞춰 써야하므로, 링버퍼용이랑 Listlock이랑은 구분해서 써야할 것)

이 문제는 에코서버 환경에서는 제대로 테스트되지않는다.

**WorkerThread내부에서의 Session요소 락**

**1. WorkerThread 에서 Session동시접근 불가.**

**특정 Session이 사용중이라면 어떤 스레드에서도 접근하지 못하게 Lock을 걸어버린다.**

(매우 간단하게 구현이 될 것)

WorkerThread안쪽에서 Session완료통지가 온다는 것은 무조건 IOCount가 1이상인 상황으로, 절대 Release될수 없는 상황이다.

**따라서 락을 걸기전에 WorkerThread가 지워지는 경우는 생각하지않아도 된다.**

**1-1. 재귀적으로 락이 걸리는 상황이 등장**

이때 WorkerThread에 진입하자마자 Lock을 거는경우,

내부에서 Send를 진행한다면 또다시 Lock이 걸리게 될것이다.

또한 Release시에도 Session에대한 요소 Lock이 걸리게 된다.

**따라서 SRWLock을 사용하는경우 데드락에 빠지게 되므로, CriticalSection을 사용해야한다.**

((

**이후 비슷한 문제의 등장**

이건 나중에 우리가 실제 최종본에서도 비슷한 문제가 나올것이다.

(SendPacket과 ReleaseSession은 라이브러리 코드이다. 따라서 이문제는 우리 라이브러리 코드안쪽의 문제니까 좀 애매하긴 함)

나중에 최종적으로 나오는 문제는 컨텐츠쪽에서의 문제이다.

컨텐츠쪽에서 SRWLock을 잘못사용할때 데드락이 걸리는데, 이 문제는 해결방안이 없으므로 구조 자체를 바꿔야만 한다.

**결국 IOCount가 0이되는순간에 SessionRelease가 호출되므로, SessionRelease가 아무데서나 되는게 문제이다. (WorkerThread에서도 Release가 된다)**

**따라서 재귀가 가능한 CriticalSection으로 간다.**

만약에 이 문제를 지금 해결하고싶다면 외부에서 호출되는 쪽에서 별도의 인자를 넣거나, 다른이름의 함수를 만들어 사용해야 한다.

**이는 Lock을 걸고서 들어왔다는 상황이기 때문이다.**

Release자체한테 어떤값을 전달하여 Lock상태인 것을 알게끔 표현하여 접근할 수 있는 상황으로 유도를 시키던가, Release라는 것을 단독스레드로 빼는 수밖에 없다.

지금 우리는 Release하는 순간이오면 직접 Release하는 구조이다. 어떤스레드던 IOCount가 0이되면 Release를 직접해버리는 구조이기 때문에 여러스레드에서 Release를 하는 구조이기 때문에 방법이없다.

))

**2. WorkerThread 에서의 동시접근 허용. (Release만 Lock)**

**WorkerThread에서 완료통지를 받아 Session에 대해 처리할때는 Lock을 걸 필요가 없다.**

**다른쪽에서 특정Session에 요청할 수 있는 것은 Send(보내기 요청)/Disconnect(끊기요청) 두개밖에 없다.**(Recv(받기요청) 은 늘 걸려있음)

**WorkerThread안쪽에서는 어차피 IOCount가 0일 때 WorkerThread내부에서 Lock을 걸고 처리할 것이기 때문이다.**

(WorkerThread가 아닌쪽에서 Session을 사용하고자한다면, Lock을 걸고 가야할 것.)

(지금은 Session내부의 데이터동기화는 고려하지않는다.)

**이는 어떤 세션하나를 대상으로 하나의 스레드만 접근해서 쓰자는 것이 아니다. 하나의 세션을 대상으로 Recv하고있는데 Send를 하는경우 문제될게 없으므로 가능해야 한다.**

이때 만약 WorkerThread에 Lock을 걸면 다른 문제가 생기지않으므로 엉망으로 만들고 잘만들었다 착각할 수도있다.

**단, WorkerThread내부에서 IOCount가 0이 결정나 삭제해야 할 경우에는 Lock을 걸어야 한다**. **위에 언급한 것은 WorkerThread에서 해당 Session을 사용하는 동안에 Lock을 걸필요가 없다고 한 것.**

IOCount가 0이되지않으면 지울수 없기 때문에 Lock을 걸지않았지만, IOCount가 0이된다면 지워야 하기 때문에 Release안에서 Lock을 걸고 지워야 한다

WorkerThread에서는 Session\*가 나오므로 SessionID를 사용할 필요가 없음.

**이중 락의 구조와 개선점**

세션 컨테이너에 대한 Lock을걸고, 그안에서 또다시 Lock을 걸어 이중으로 락을 건 현재 상황은 아무도 쓰지않는 방법이다.

이방법이 얼마나 이상한지 느끼기위하여 시도하는 것.

**이러한 구조는 생성, 삭제(new / delete)가 나오는 구조라면 반드시 등장하는 구조이다.**

이러한 생성삭제가 등장하는 구조에서는 동기화를 확실하게 잡을 방법이 없다.

**IOCP에서 생성/삭제 방식을 사용하지않는 이유**

그래서 IOCP에서는 이러한 생성삭제 방법을 사용하지않는다.

1. New/delete에 대한 성능이슈

2. Session요소에 대해 확실하고 깔끔하게 동기화를 잡을 방법이 애매하다.

(대안으로는 한쪽에서 삭제할 세션을 모았다가 한쪽에서 한방에 삭제하는 방법도 존재는 할것)

수시로 생성/삭제하는 방식은 너무 난해하다.

**따라서 우리는 세션자체를 지우지않게끔 진행하며, 여기서 더 나아간다면 SessionLock까지 없앤다면 더 좋을 것이다.**

(세션을 지우지않겠다는 것이 배열로 가자는 것은 아니다. 굳이 배열로 가지않더라도 지우지않는 방법도 있을 것)

**DisConnect()**

Disconnect는 컨텐츠쪽에서 연결끊기를 요청하는 함수이다.

1. 비정상적인 메시지가왔거나,

2. 중복로그인, 패스워드 에러 등의 이유

**DisConnect안에서는 당연히 Session을 검색해서 찾아내야 하는 상황이므로 Session을 검색해야 하는 상황이므로, List 덩어리에 Lock을 걸고 검색성공한 특정 Session에 대한 Lock을 걸어놓고서 빠져야 한다.**

**Disconnect는 연결을 끊는것으로, Release와는 다른 얘기.**

**IOCount가 0떨어지도록 하여 Relase를 유도해야 할것이다.**

**OverlappedIO에서는 Release는 우리가 호출하는 것이 아니라 IOCount가 0이되면 자동으로 이루어진다.**

**Disconnect에서 할일은 한번만 끊게 하는 것으로, CloseSocket이 한번만 되게끔 해야할 것.**

(LINGER셋팅 후 CloseSocket호출. LINGER는 지금시점이나 처음에 걸어도 상관없다.)

어차피 Session락걸고하니까 문제될건 없다.

((

기본적인 소켓옵션들은 대부분이 다 listen에 걸어두면 상속이 되는데, LINGER의 경우는 정확하지 않으므로 직접 테스트해보고 사용할 것.

))

**이후 Session을 모두 사용했다면 Lock을 풀고 빠져나간다.**

Session = FindSession();

setsockopt(SO\_LINGER); // LINGER Setting

CloseSocket();

**SendPacket()**

Session->SendQEnque(Pacekt)

Session->SendPost(); //내부에서 WSASend호출

이때 SendPost안쪽까지 생각해본다면 문제가 있다.

**Send가 중첩되는 문제**

**지금이 SendPacket은 외부에서 마음대로 호출 될 수 있다는 기준으로 만들었다.**

**1. SendPost(WSASend)의 중첩**

**Send는 1회로 제한해야 하므로 SendPost에서 Send가 진행중이라면 WSASend하지않고 그냥 빠져야한다.**

**Send가 진행중인 경우, 해당 Send에 대해 완료통지가 오지않은 상황이기 때문이다.**

**(SendOverlapped 사용중)**

**따라서 Send가 진행중인지 알수있게끔 flag를 추가하여, Send를 1회로 제한한다.**

**2. SendPacket(SendQ.Enqueue)의 중첩**

**SendPost에서 Send가 진행중인 경우(SendIO가 걸려 SendQ버퍼를 대상으로 Send)라도, SendQ.Enqueue의 경우 뒤에다 데이터를 추가하는것에는 문제가 없다.**

하지만 데이터가 중첩되어 쌓이는 것은 막아야 하므로(crash) 이를 막아야한다.

(지금은 세션을 찾아서 사용할 때 세션단위로 락을 걸기 때문에 상관 없음)

(이후 락프리 구조를 적용할 경우 락을 걸지않고 진행한다)

**중간정리**

**SendPacket(SendQ.Enqueue)의 중첩을 막는 이유는 (메모리 참조오류/예외) 때문이다.**

**SendQ 락을 믿고 하나의 Session을 대상으로 여러스레드가 Send해서는 안된다.**

**억지로 만든다면 만들수있겠지만, 이러한 구조는 쓰이지않는다.**

**위 같은 경우 여러스레드에서 Send가 어떤 순서로 올지 전혀 보장되지않음.**

**한스레드에서 Send를 중첩해서 보내는 것은 정확히 순서가 보장되지만, 여러스레드에서 Send를 보낸다면 누가먼저 처리되어 도착할지는 아무도 알수없음**

또한 이 구조는 코드차원에서 동시접근이 발생했을 때 예외를 발생하지않게끔 하자는 것으로, 이렇게 권장하는 것이 아니다.

**SendPost()**

일단 SendPost들어왔다면 SendQ에있는걸 보내기작업이 들어가는것이니까 바로 들어오자마자 Sendflag를 확인하면 된다.

SendPost();

{

If(SendFlag == true)

Return;

SendFlag = 1;

…

}

**이렇게하면 작업이 엉망진창이되기때문에, (한줄만 내려가도 값이 보장되지않기 때문) Interexchange를 이용해서 0으로 1됐을때만 체크해서 빠질수있도록 간다.**

SendPacket()을 호출하여 큐잉에 성공했는데 SendPost(WSASend)중이라면, 우리는 넣고 빠져야 한다.

이후 Send완료통지가 다시왔을때는 SendFlag를 다시 바꿔주고, SendQ에있는 모든데이터를 Send해야 한다.

**따라서 Send완료통지가 왔을 때 SendFlag를 바꿔주면서, SendPost가 들어가야한다.**

(지금문제는 IOCP에 대한 문제가 아니라 스레드 문제이다.)

이때 동기로 처리될 경우 함수가 채 리턴되기도전에 완료통지가 올 수 있다.

**이러한 상황이 마음에 들지않는다면, 송신버퍼를 0으로 만들어 비동기처리를 유도할수 있겠다.**

**이코드에서 위험한 부분**

SessionMap에대해 교차로 락을 거는부분이 데드락의 위험을 가지고 있다.

**(한곳에서 모든 것을 처리한다면 이슈가 발생하지않음.)**

**앞으로 적용해볼 Lock-Free언급**

Session 맵에 대한 Lock, Session낱개에 대한 Lock.

이러한 Lock을 없애고자 우리는 Lock-Free을 적용해볼것인데, 이경우 락프리로 도배될것이다.

**락프리는 InterLocked기반으로 존재하므로, Lock을 없애겠다는 것은 최대한 인터락을 사용해서 우회하겠다는 뜻이다.**

이러한 유저객체 동기화가 가장 처음 하는일은 해당 임계영역이 사용중인지 InterLocked으로 비트체크 한다.

동기화객체대신 Lock-Free를 적용한다고하여 성능이 좋아지는 것은 아니다.

경합이 발생되지않는다면 (유저모드 객체라면)성능차이가 전혀 없기 때문에, 결국 경합의 유무가 중요하다.

WorkerThread는 보통 8개, Session은 5000~10000개 정도 돌아가므로 사실 거의 경합이 발생되지않는다.

(지금 이 구조에서 경합이 발생되는 경우는 같은 Map에대해 접근하거나, Session낱개 요소를 같은곳에서 사용하거나..)

포트폴리오에 Lock-Free알고리즘을 넣을것이나, 동기화객체를 쓰지않는 IOCP보다 빨라진다는 언급은 하지않는다.

**<7.22>**

**SessionID**

컨텐츠 쪽에서는 해당 Session을 대상으로 SessionID가 보관되어있을 수 있다.

**이를 바탕으로 Send를 시도한다면 의도치않는 Session에게 Send를 시도하게되므로 이를막기위해 유니크한 값으로 간다.**

**파트분리**

현재 구조정리

(여기서 말하는 컨텐츠쪽 파트는 단순 게임로직을 일컫는다.)

**네트워크 라이브러리내의 스레드**

**[ WorkerThread, AcceptThread ]**

**네트워크 라이브러리 -> 컨텐츠쪽 요청**

**[OnAccept, OnRecv, OnLeav, OnSend]**

**컨텐츠 -> 네트워크 라이브쪽 요청**

**[ SendPacekt(), DisConnect() ]**

**IOCP 구조에 대해**

**IOCP에서 스레드를 나누는 구조로**

**컨텐츠 로직을 별도의 UpdateThread로 두기,**

**WorkerThread에서 모든 로직을 처리.**

등을 소개했었다.

IOCP를 기반으로 짜는 구조는 굉장히 다양하다.

**우리가 시도하는 구조는 범용적인 네트워크 라이브러리의 성격을 띄고있다.**

**따라서 어떤상황, 어떤스레드에서 동시에 호출하더라도 문제없이 돌아가는 상황을 목표로 하기 때문에 SendPacekt이나 DisConnect가 어디에서 호출되든 문제가 없어야한다.**

**(단, 실무의 경우 한가지 상황에 맞게끔 구조를 짤수도 있음.**

**만약 사용자가 연결되자마자 뭔가를 보내야 한다면 OnAccept에 SendPacket까지 포함될 것)**

하지만 우리가 시도할 방식이 일반적인 IOCP(IOCP는 정석이없다. 범용적인지 아닌지만 있을뿐)방식은 아니다.

단순히 일반적인 IOCP만을 익힌다면 왜 이렇게 돌아야하는지에 대한 이해도가 부족하기 때문에, 우리는 이런식으로 하면 얼마나 고달픈지 이해하기 위해 시도한다.

(왜 SendThread를 사용하는지, 왜 Session을 이렇게 관리하지않는지 등..)

**지금 구조에 대한 정리 (복습)**

**Overlapped I/O의 가장 기본적인 사용**

**IO작업당 Overlapped작업이 필요하다.**

**하나의 세션을 대상으로 두개의 완료통지가 오기 때문에, 함부로 삭제해서는 안되는 상황이다.**

가장 기본적인 IOCP는, Send/Recv마다 Overlapped 구조체를 동적할당하여 전달하는 방식이다.

(관리가 불가능한 방식. 샘플에서 자주 등장한다.)

완료통지가 오기전까지 해당 구조체가 삭제될 일은 없다.

**따라서 Overlapped구조체를 확장하여 버퍼포인터 등 넣고싶은 것을 넣어 WSASend를 요청한다.**

**이후 완료통지가 온다면 넣었던 데이터들이 모두 담겨져 있을 것. (받은데이터, 보낸데이터..)**

에러가 날경우 Overlapped구조체와 버퍼를 지우고 CloseSocket을하면 끝이난다.

**하나의 소켓대상으로 2개의 IO가 걸려있다고 하더라도 CloseSocket시 등록된 작업이 그 즉시 중단되고 완료통지가 떨어지게된다.**

**완료통지 시 Overlapped구조체, Buffer를 지우면 마무리된다.**

이 경우 IOCount와 같은 안전장치가 없기 때문에 CloseSocket한 소켓을 또 지울수있는 상황이 발생될 수 있으나, 이 구조의 경우 관리자체를 포기했기 때문에 서버측에서는 끊는 경우가 없다.

만약에 이 오버랩자체가 여러 개가 생겼다.하나의 소켓을 대상으로 여러 개의 작업을 해서..

그럼 여기에 어떤 소켓들이 다 담겨져 있을건데..

다 똑 같은 소켓들일것이다. 같은 소켓들

근데 만약에 어떤애가 에러가났다 그럼 어떻게 해야하나? Transferred 0떨어지던가 결과가 에러가 났다.

((

**완료통지**

내가 작업을 건것에 대한 완료통지는 어떠한 상황에서도(에러, 중단 등) 오게된다.

따라서 최초 실패에 대한 완료통지에 이어오는 완료통지가 성공이든, 에러든 오는 것은 보장된다.

**Closesocket이 2번 되는경우**

단, 이때 closesocket이 하나의 소켓에 대해 두번 될수있는 상황이 발생한다.

각기 다른 완료통지를 서로다른 스레드에서 받은 경우가 있을 수 있겠다.

거의대부분의 IOCP구조에서 이러한 문제(소켓 재사용)를 대비하기위해 대기하는 코드를 짜놓지 않는다.

거의 대부분은 모르거나 그냥 받아들이고, 특별한 문제로 보지않는다.

))

**우리가 사용할 Overlapped I/O방식**

위와같이 설명한 Overlapped I/O의 가장 기본적인 방식은, 일단 관리자체가 안될뿐더러 Overlapped를 매번 동적할당 해야하는 등의 문제로 사용 할수 없다.

**따라서 우리는 Overlapped I/O구조체 2개를 고정으로 간다.**

Overlapped를 3개이상 넣어 여러 개의 IO를 관리하는것도 문제없지만, 딱히 이득이 없기때문에 하지않는다.

**send() 속도**

**send()라는 함수자체는 엄청나게 느리다. 만약 5000개의 소켓을 대상으로 루프를 통해 send를 호출한다면 거의 2~300m/s단위로 멈추는게 체감이 될 정도로 느리다.**

만약 5000개의 소켓을 대상으로 루프를통해 send를 호출한다면, 거의 2~300m/s단위로 멈추는게 체감이 될정도로 느리다.

(send는 가볍기 때문에 WSASend보다 빠름)

따라서 서버코드에서는 많은인원에게 send하는 코드는 존재해서는 안된다.

공지를 띄우는 개념이라고 하더라도 채널/방 으로 분산으로 send하며 이는 많아도 몇백명 수준이다.

**IOCount의 필요성**

우리는 하나의 세션에대해 두개의 I/O를 요청하고, 두개의 완료통지를 받기 때문에 하나의 Overlapped만 지울수있는 상황이 존재할 수 없다.

**따라서 세션의 삭제를 통제하기위해 IOCount를 통해 IO를 카운트한다.**

**IOCount는 IOCP를 사용하는 코드가 대부분 취하는 방식이다.**

**IOCount를 사용하지않는 구조**

**단, 일부의 경우 IOCount방식을 쓰지않고 연결이 끊기면 끊어버리는데, 이 방식은 Session에대해 할당과 Release가 없어야한다.**

**이는 에러가나면 Session을 반환하되,삭제의 개념이 아닌 확보해놓고 재사용하겠다는 개념이다.**

**이 방식의 문제는 첫번째IO로 연결끊기를 판단하여 미사용으로 전환했는데, 두번째IO가 오지않았음에도 불구하고 다른쪽에서 이 세션을 할당하여 즉시 재사용해버릴수 있다.**

**보통 이러한 경우 남은IO가 n m/s정도로 거의 즉시 완료통지가 오기 때문에 이 세션에 대해 재활용의 시간을 벌어준다면 문제는 나지않을 것이다.**

이론상으로는 안전하지만 위험성을 100% 막아주지는 못하므로 불안한 구조라고 볼수있겠다.

**비동기로 IOCP를 사용하는 경우**

우리는 Send라는 행동자체를 OVERLAPPED IO로 작동이된다는 전제로 개발한다.

(비동기가 되는경우는 송신버퍼를 임의로 0으로 설정하거나, 동기로 돌수없을 때)

따라서

Overlapped 구조체,

메모리버퍼 인자로 들어가는 데이터(Session). -> Page Lock발생

**동기IOCP의 사용 - ZeroByteRecv**

이때 관리가 귀찮은 등의 이유로 Overlapped 구조체를 쓰고싶지만 동기입출력만을 사용하여 만드는 경우가 있다.

**이는 비동기로 돌지않기 때문에 IOCP를 사용하지만 Overlapped I/O는 사용하지않겠다는 뜻이다. (마치 ZerobyteRecv처럼)**

ProActor(WorkerThread에서 완료통지만 전달)가 아닌 ReActor(WorkerThread에서 마무리)를 유도시키는 것이다.

**이경우는 Overlapped I/O를 보존시킬 이유가 없기 때문에, send의 경우 아예 완료통지를 꺼버리는 경우도 있다.**

(이미 동기로 작업이 끝났기 때문에, 작업의 단위인 Overlapped I/O가 필요 X)

이는결국 기존의 send와 똑같이 돌아가고, 관리하기 훨씬 수월해진다.

우리는 이 방식으로 하지않는다

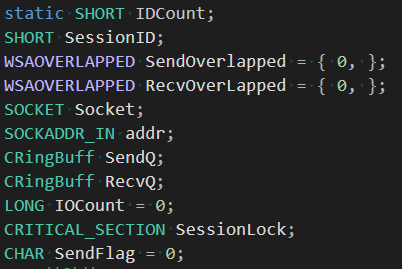
**적용할 IOCP 방식**

우리는 어떤 환경에서도 문제없이 작동하는 네트워크 라이브러리를 만드는 것이 목적이다.

따라서 복잡하더라도 OverLapped I/O까지 염두한다.

WorkerThread한 개만 사용하는 경우 동기화이슈가 매우 간단해지기 때문에 생각할꺼리가 없어지나, (실제로 이렇게 상용화된 게임서버도 다수 존재함) 이렇게 가지않음.

**Session구조체 인자**



SendFlag와 IOCount는 서로 목적이 다르므로 혼동하지 말 것.

만약 Sendthread를 따로 두고 Send의 IOPENDING을 원하지않는다면 SendFlag는 존재하지않아도 된다. (만약 IO\_PENDING이 나는경우 동기로 돌수 없다는 것이므로 끊어버릴 것)

흔히 볼수있는 코드들 중에 Transferred 0 이 나오는경우 closesocket하고 지워버리는 경우가 있다. 이는 우리가 염두한 예외처리를 하지않은 상태이므로, 다른 코드를 볼 때 유의할 것

**스마트 포인터**

SmartPtr<int>p = new int;

(예시로 작성한 코드이므로 실제 스마트 포인터 형태와는 다르다)

**스마트 포인터의 제일 기본형태는 포인터가 아닌 스마트포인터 객체를 사용하는것으로,클래스 내부에 \*와 ->로 오버로딩하여 기존 포인터처럼 사용하게끔 되어있다**

이때 동적할당한 포인터를 다른 포인터 변수로 가리키고 싶다면 스마트포인터를 또 선언한다.

SmartPtr<int>p2 = p;

내부에 복사생성자가 호출되어 대입될 것이다.

함수의 인자를 포인터로 던지는 경우에도 스마트포인터객체로 변경된다.

스마트 포인터는 따로 delete하지않아도 지워진다.

포인터를 별도의 클래스로 랩핑하여 C#과 같은 언매니지드 느낌으로 구현한 것이다.

**스마트포인터 내부**

Smartptr<T>

\_ptr;

RefCount;

생성자 : 실제 포인터를 받고 Count가 증가하는 형식

생성자 : Count가 증가할것이고, 소멸자에서 감소할 것이다.

복사생성자/대입연산자 : 단순히 포인터를 받아들인다. (카운트 증가)

이 래퍼런스카운트를 받아서 카운트가 증가해야하는데, 이를 전역변수로 갈수는 없다.

**최초로 할당받는 객체가 래퍼런스카운터도 동적할당하여 이 포인터를 공유하고, 소멸자에서 카운트를 차감하다 0이되는경우 delete하는 방식**

(이 방식으로 직렬화버퍼를 커버할 것이기 때문에 잘 기억해 둘 것)

**스마트포인터는 카운트값이 들어가므로 IOCount와 비슷하다고 생각할 수 있다.**

**스마트 포인터 방식으로 Session요소 동기화?**

**결론부터 말하자면, 스마트포인터 방식의 RefCount로 요소를 동기화 하는 것은 불가능하다.**

**Session포인터는 내가 사용하면 잠그고, 사용이 끝나면 풀어야한다.**

**반면 스마트포인터는 복사될때마다 내부 RefCount를 증가시키는 형태이기 때문에 누군가는 반드시 물고있어야 한다.**

**우리상황은 WSASend를 하는 경우 Session포인터는 우리손에서 떠나게되고,**

**완료통지가왔을 때 Key값으로 해당 Session을 받는 형태이다.**

**따라서 중간에 Session이 사라지는 구간이 존재하므로, RefCount가 0이되므로 이러한 방식은 적용이 불가능하다**

**SessionMap 동기화**

SessionMap에 삽입/삭제시 자료구조 차원에서의 동기화이슈를 막기위해 동기화객체가 필요하다.

**SessionMap 자료구조 동기화**

Session동적 할당

SessionMap.LOCK()

Push()

SessionMap.UNLOCK()

Session검색의 목적은 사용/삭제 두 경우 밖에 없다.

**SessionMap 요소 동기화예시 1**

SessionLOCK()

SessionFind()

..Session 사용..

SessionUNLOCK()

**이 경우 특정 Session이 사용중일경우 로직상 문제는 없을 것이나,**

**어떤 스레드도 SessionMap요소 사용하지못하므로 멀티스레드 구조가 의미없어진다.**

**SessionMap 요소 동기화예시 2**

SessionLOCK()

SessionFind()

SessionUNLOCK()

..세션사용..

**이 경우 UNLOCK이후 동시에 스레드가 치고들어오는 경우를 막을 수 없으므로 로직상 문제가 생기게 된다.**

**결국은, SessionFind이후 해당Session만을 Lock을걸고 SessionMap에 대해 UNLOCK해야한다.**

**SessionMap 요소 동기화예시 3 (최종 모습)**

(여기서는 삭제를 예시로 들어본다)

SessionMap.LOCK()

SessionMap.Find()

Session.LOCK() - (1)

SessionMap.Delete() //세션에 대한 해제는 IOCount쪽에서 한다.

SessionMap.UNLOCK()

Session.USE();

Session.UNLOCK()

(1) SessionLock (SessionMap.Lock() 를 풀기전에 세션자체를 안전하게 보호해야한다. 그래서 세션마다 존재하는 이 세션의 락을 활용하자는 것이다.세션내부에 락.)

**IOCount방식의 Session요소 동기화**

만약 세션내부에 동기화객체를 넣지않고, 마치 IOCount처럼 접근스레드 카운팅 방식으로 사용하는것도 가능하다.

**동기화 객체를 사용하는것보다 빨라지긴 하겠지만 결국 경합이 발생되지않는다면 똑 같은 구조이므로 큰 이득은 없다.**

**추가로 유의할점**

검색-사용이라는 부분이 어디에서 호출되느냐에 따른 고민이 필요할 것이다.

세션을 사용하는 파트가 중첩으로 락이 걸리는 상황이고, WorkerThread안에서 진행이 되기 때문에, 여러가지 이슈가 나오게된다.

이를 해결하지못한다면 동기화객체 사용이 미숙한 것.

**데드락 위험성**

위 구조는 교차로 락을 걸고 있기 때문에 데드락위험성을 가지고있다.

이는 지금구조에서는 뾰족한 방법이 없다.

**CloseSocket 한번만 하기**

나중에 동기화객체를 안쓰는상황이오면 약간 까탈스러울수도있는데 지금은 동기화객체가 있기 때문에 큰 문제는 없을 것.

**SendPacket에 대해**

Workerthread 내부에서 완료통지를 통해 Session에 접근하는 상황에서는 락을 걸지않는다.

WorkerThread 내부 OnRecv내부에서는 SendPacket이 호출되는데, SenPacket에서는 동기화를 걸고 검색하는 코드가 들어가게된다.

이때 Find하기위해 동기화를 걸고 가게된다.

경우에따라 종료까지 들어갈수있음.

이부분을 어떻게 할것이냐인데.. 이거는.. 그냥 어떻게보면 워커스레드에서 호출을 하든지.

아니면 다른 업데이트 스레드에서 호출을 하든지, 저 SendPacket안쪽에서는 동일하게는 안만들어도된다. 왜냐면 이친구자체는 외부에서 호출할 목적으로 나온 친구이다.

일단은 SendPacket에서는 락을걸고 WorkerThread에서 SendPacket이 호출되서 들어가는데 이안(?)에서 Release까지 존재할수가없음. 그건말이안됨..

그건애초에 고려대상이 아닌거고.

이안에서 얘를 호출하는데 그럼 락을 안걸어도되는거 아니냐 고 하는데 지금이 SEndPacket이라는 걸 호출하는 부분자체가 컨텐츠쪽의 코드이다. 컨텐츠쪽의 코드는 분리가 된거기대문에 실제 호출하는 것은 워커스레드지만, 그걸 구분할수 있는 상황은아니다.

그래서 워커스레드.. 결과적으로 호출하는스레드는 워커스레드지만. 그래도 컨텐츠 코드에서 해당하는 부분이기 때문에 그냥 컨텐츠쪽에서 호출하는것과 동일하게 가겠다는 것이다.

근데 만약에 이 안에서 직접적으로 뭔가.. 우리가지금 컨텐츠쪽에서 호출할수있는게 Disconnect랑 SendPacekt두개밖에 없는건데..

이워커스레드 코드안에서 직접적으로 얘네들이 혹시라도 호출될 일이 있다라면 그거는 선택적으로 하셔도되는거에요

또는 RecvPost SendPost도 마찬가지이다. 저도 그렇게되어있고. 두개씩돼요 사실. Send는 아니고 Recv같은 경우는. Recv는 지금 호출위치가 Accept하고 WorkerThread쪽이에요. 걔네두개가 다르게 나와도 괜찮아요.

조금미묘하게 달라질수있거든요. 어셉트쓰레드쪽에서 호출하는 최초의 리시브가 워커스레드에서 호출하는 어떤 리시브.. 뭐가다른지는 기억이안나는데.. 암튼 나눠져있음.

이거는 여러분들이 해보시면서 스타일대로가세요 이거까지 답을 드릴수는없으니까.

**더미 사용법**

EchoDummy Port : 6000

# 사용법

ServerIP : 알아서 입력

Disconnec Test : Yes – 무작위 재접속 시도. NO - 에코테스트.

Client Count : 테스트 클라이언트 수 입니다. 1 명 / 50 명 / 100 명 선택,

이제부터 발생하는 문제는 스레드동기화 문제이므로 어떤스레드에서 어떤 코드가 돌았는지 정확하게 파악해야하고, 이는 5천개 배열을 사용했다고 한다면 5천개 배열을 모두 뒤져야 한다.

OverSendCount : Echo 응답이 오지 않아도 보내는 수치 입니다.

기본은 1로 테스트 해주시고 문제가 없다면 늘립니다.

1 - 핑퐁. (문제X)

2 - 100개 쏟아붓는다. Send가 늘어나는경우 Recv/Send가 두개씩 걸리고 문제가 나기시작할 것이다.

Loop Delay ms : 더미 클라이언트의 컨텐츠 루프 딜레이 값 입니다.

적으면 적을 수록 에코를 빨리빨리 보냅니다.

도저히 감이안잡힐떄 느리게해서 디버깅해볼것.

스레드 루프

딱히 볼필요없음.

나중에 IOCP모듈을 만들면 통신해야 하므로 이에 대응하는 클라도 만들어야한다.

이렇게 만든 클라도 테스트를 해야하기 때문에 IOCP로 들어가고, 사양이 굉장히 높아지게 된다.

(클라n명분의 IOCP와, n\*2(Send, Accept)만큼의 WorkerThread가 만들어 진다.)

**Wait Count**

응답이 오지않는 카운트의 개수.

샌드하지않았는데 오지않는다면 카운팅이 올라가게 된다.

송수신과정에서는 들쭉날쭉하지만, (S)키를 눌러 스탑했을때 WaitCount는 무조건0이와야한다.

0이아닌경우 유실되었거나 보내주지않은 값이 있는것.

있어서는 안되는 상황이다.

**레이턴시**

지금은 고려하지않는다.

Connect Try Success는 큰의미가 없으므로 일단은 남겨둔다.

Connect fail - 커넥트 에러. (신경쓰지않는다.)

Disconnect테스트할때 간혹 백로그큐가 다차서 fail나는 경우가 생기는데, 이는 문제삼지않음

# 실행화면에서

Error - Connect Fail : 클라에서 connect 함수가 실패한 경우 입니다.

서버가 accept를 하지않거나, 빠르게 accept 하지않을 경우 나타날 수 있다.

(신경쓰지 않는다.)

100명가지고는 잘안된다

Disconnect테스트할때 간혹 백로그큐가 다차서 fail나는 경우가 생기는데, 이는 문제삼지않음

\* Error - Disconnect from Server : 서버가 일방적으로 끊는 상황.

**만약 서버의 AcceptTotal과 더미의 ConnectToal수치가 같다면 문제가 없는 경우이나, 아닌 경우 서버가 일방적으로 끊는 경우이므로 동기화 오류문제로 볼수 있다.**

지금 더미에도 Connect Total값이 카운팅되고있다. (이는 재연결 테스트시에만 유효하다.)

((

**backlog Q에 자리가 없는데도 3way handshaking을 받아 연결성공이 뜬 후에 그 즉시 바로 끊기는 현상을 이전에 언급한 적이 있다.**

**TCP스택 내부에서도 3Way handshaking을 진행하는것과 완료된 클라를 보관하는 backlog Q 버퍼가 따로 존재**하는데, 이때 3Way handshaking에 성공해서 backlog Q에 넣으려는순간 backlog Q가 닫혀버린 경우이다.

**일단 3way handshaking을 성공했기 때문에 클라에는 마지막 ACK까지 전달이 되어 Connect가 성공되었으나, backlog Q가 다찼기 때문에 그 즉시 끊기게 된다.**

따라서 연결한 다음 끊겼기 때문에 더미(클라)측에서는 Disconnect From Server 수치를 증가시킨다. (이 수치가 발생하면 ConnectTotal값을 차감시키게 된다)

서버쪽에서도 Accpet가 나오지않기 때문에 달리 판단할 방법이 없음.

**따라서 더미(클라)의 ConnectTotal과 서버의 Acccept Total가 같다면 백로그 큐의 이슈문제로 판단한다.**

))

WaitEchoCount

S누르면 0떨어져야한다. Disconnect From 도 절대 올라가면안된다.

Error - LoginPacket .. 관련항목 2개 : 무시합니다.

차후 다시설명한다.

\* Error - Echo Not Recv : 에코를 보냈는데 500 ms 이상 응답이 없는 카운팅 입니다. 얘가 올라간다면 어마어마하게 느린것.

이 수치가 누적으로 계속 올라가거나 줄어들지 않는다면 서버에서 패킷 응답이 없거나 손실된 경우 입니다.

일시적으로 수치가 있어도 상관 없으나 계속 커지거나 줄어들지 않으면 문제 있음.

\* Error - Packet Error : Echo 보낸것과 응답 받은것이 다름.

8바이트짜리 카운터를 주고받는다. 내가보내지않은게 왔을때 에러낸다.

당연히 존재해서는 안된다. 누적치임.

위 목록에서 \* 이 붙은 항목은 나타나선 안됩니다.

헤더 2Byte (길이)

데이터 8Byte (에코)

# 스트레스 클라이언트

1. 클라이언트 수량만큼 접속 해제를 반복.

2. 각 클라이언트 세션마다 고유하게 증가되는 값 8 Byte 를 서버로 송신

3. 서버로부터 받은 값을 자신이 보냈던 값과 비교.

테스트 항목.

- 접속을 거부하는 경우는 없는가 ?

- 서버가 접속을 끊는 경우가 있는가 ?

- 보낸 값과 되돌아온 값이 일치하는가 ?

**Server Debug**

TotalAccept수치

Accept TPS(Transaction Per Second) – 초당 몇회의 Accpet가 발생하는지

Accept TPS는 Accept시도에 따라 증가할 것이므로 성능측정의 대상이 되지는 않는다.

추가적으로 RecvQ, SendQ, 직렬화버퍼는 끝까지 들고가면서 기능을 확장시킬것이므로 확실하게 테스트하고 가야한다. (오류가 0이라는 것을 검증해야함)

**Send/Recv 추가적으로 언급**

OverSend 수치를 200으로 준다면 2000(10\*20)byte가 나올것이다.

Send는 한번으로 제한하기때문에, 첫번째 Send는 WSASend로 들어가겠지만, 나머지는 링버퍼에 저장되어 이후에 갈것이다.

**리시브의 경우 리시브를 하나씩 뽑아서 OnRecv를 호출해야한다.**

**이번에 받은 전체 데이터를 모두 OnRecv에 던지는 것이 아니다.**

**OnRecv받은 쪽에서는 직렬화버퍼를 새로 만들어 마샬링하여 옮겨담아야 한다.**

(받은거 그대로 돌려보내지 말 것)

지금은 성능측정 의미가 없음. (단, 500m/s 를 넘어가는 것은 안될 것)

오류잡는것에 집중할 것.

계속하여 언급했듯, 정확하게 원인을 파악하고 시뮬레이션 하여 디버깅해야한다.

감으로 디버깅하여 돌아간다면, 영원히 불안감이 사라지지않을 것.

추가적으로 로컬에서 잘 돌아간다 하더라도 서버에 올려놓고 돌린다면 또다른 상황이 펼쳐질수 있다. (PC대 PC의 연결이라고 하더라도 조금이나마 레이턴시가 생기기 때문)

최종적으로 L2, L3를 모두 거치는 방식으로 테스트해도 문제가 날 것이다.

이러한 이유로 현업에서 사내테스트, 더미테스트, 클로즈테스트, 오픈테스트를 모두 거쳤음에도 불구하고 문제가 발생하게 된다.

문제가 생기면 덤프를 확인(지금은 없으므로 로그정도)

면접에 자주 등장하는 질문으로 디버깅을 어떻게 하냐에 대한 질문이 자주나온다.

**멀티스레드 환경에서의 링버퍼**

앞서 링버퍼를 쓸 때, 한쪽에서는 Enque하고 다른쪽에서 Deque하는 경우,

이론적으로는 문제가 없어야 함에도 불구하고 내가사용하는 변수가 바뀌었기 때문에 특정부분에서 문제가 날 것이다.

**이 같은 문제를 해결하기위해, 사본을 활용하는 방법(내가 알고있는 값을 활용)을 적용할 수 있겠다.**

**사본값의 활용**

멀티스레드 환경의 링버퍼처럼, 과거의 값이 이 행위에 어떤 오류를 내는값이 아니라면 실시간의 값을 얻어올 필요는 없다.

따라서 사본의 값을 과거의 값을 기준으로 작업한다면 아무런 문제가 일어나지않는다.