**IOCP 정리**

**2015182025**

**게임공학과**

**오현호**

**2021 01 09 ~ ing**

**P3 객체 생성 CreateIoCompletionPort**

**P4 소켓 등록 CreateIoCompletionPort 임시 Worker Thread**

**P5 GetQueuedCompletionStatus,**

**P6 PostQueuedCompletionStatus**

**P7 IOCP 객체 구조 모양**

**P8 IOCP 서버 설계, Accept 처리**

**P9 클라이언트 객체, 오버랩 구조체**

**P10 오버랩IO 포인터의 확장, 완료 처리**

**P11 버퍼 처리 – Recv**

**P12 버퍼 처리 – Send**

**P12 버퍼 처리 – Send**

**P12 버퍼 처리 – Send**

**P13 오버랩IO에서 IOCP로 추가할 것, 프로토콜 정의 + 패킷 처리 루틴**

**P14 Recv 구현 + 패킷 재조립**

**P15 Recv 패킷 재조립 구현, AcceptEx**

**P16 WSASend + Send 완료 구현**

**P17 멀티플레이어 구현, 병렬 처리**

**P18 프로세스와 쓰레드**

**P19 멀티쓰레드 프로그래밍스타일 2가지**

**P21 멀티쓰레드 사용할 때 주의점**

**P22 쓰레드 생성 종료**

**P23 Data Race**

**P24 IOCP 멀티쓰레드, Worker\_Thread**

**P26 lock\_guard**

**<Mutex 없이 동기화 할 때 문제점>**

**P27 volatile**

**P28 메모리 일관성 문제**

**P29 Cache Line Size Boundary**

**HANDLE CreateIoCompletionPort (**

**HANDLE FileHandle,**

**HANDLE ExistingCompletionPort,**

**ULONG\_PTR CompletionKey,**

**DWORD NumberOfConcurrentThreads );**

**HANDLE hIOCP = CreateIoCompletionPort(INVALID\_HANDLE\_VALUE, NULL, NULL, 0};**

커널 객체 생성 API

FileHandle에는 무조건 INVALID\_HANDLE\_VALUE가 들어가야함

그래야 IOCP 객체를 생성함

NumberOfConcurrentThreads - IOCP 객체가 동시에 컨트롤 할 수 있는 쓰레드 수

0을 넣으면 컴퓨터에 돌아가는 코어의 갯수만큼 사용하겠다는 뜻

**HANDLE CreateIoCompletionPort(socket, hIOCP, key, 0};**

IOCP 객체를 만들었으니 이제 객체를 이용해 send recv를 해야함

데이터 통신을 하려면 소켓을 IOCP 객체에 등록을 해줘야 함

등록을 해줘야 소켓의 send recv가 IOCP 객체를 통해서 이루어짐

자세히 보면 위에 객체 생성API랑 이름이 같음. 이래서 IOCP가 어려움

socket을 hIOCP 객체에 연결을 하겠다는 의미

key는 socket을 어떤 숫자로 부를까 정하는 것

IOCP를 IO할 때 소켓을 주지 않고 키를 줌, 그래서 소캣을 등록시킨 다음엔 키를 통해서access함,

key 값은 개발자 마음이므로 주로 플레이어의 ID나 클라이언트 자료를 저장하는 자료구조의 주소값으로 설정함, 중요한건 값이 유니크해서 중복이 없어야함

==========================================================

IOCP에서 관리하는 쓰레드를 생성하는 API는 따로 없음

멀티스레드로 동작할 때 IOCP 객체에 스레드들을 어떻게 연동을 시키느냐?

스레드 생성은 프로그래머 알아서 생성하고 나중에 등록을 시킴

IOCP는 굳이 멀티스레드 아니어도 동작 가능

메인스레드를 IOCP 스레드로 취급을 해서 가능

그러나 고성능 프로그래밍 하려면 멀티스레드 필수임

IOCP로 오버랩IO할 때 완료 검사는 어떻게 하느냐?

**while(true) { //서버 메인 루프**

**GetQueuedCompletionStatus(**

**hIOCP,**

**&dwIOSize,**

**&key,**

**&lpOverlapped,**

**INFINITE;**

**);**

**lpOverlapped에 따른 처리;**

**}**

**GetQueuedCompletionStatus** 함수 호출을 하면 멈춰있다가 오버랩IO가 완료되면 함수 종료되면서 정보들을 파라미터에 넣어줌, 우리는 이걸 가지고 (**lpOverlapped에 따른 처리)** 여기서 오버랩IO CallBack에서 했던걸 해주면 된다

그래서 사실상 얘가 스레드를 IOCP 객체에 등록해주는 함수임

그래서 지금은 다른 스레드를 선언하지 않았지만 얘를 메인 스레드에 호출하면 메인 스레드를 IOCP에 넘겨주는거임, 하위는 자세한 파라미터

**BOOL GetQueuedCompletionStatus(**

**HANDLE CompletionPort,**

**LPDWORD lpNumberOfByte,**

**PULONG\_PTR lpCompletionKey,**

**LPOVERLAPPED \*lpOverlapped,**

**DWORD dwMillisocnds );**

**CompletionPort –** IOCP 객체 핸들 넣어주면 된다

**lpNumberOfByte –** 총 전송된 데이터 양

**lpCompletionKey –** 오버랩IO니까 여러 개의 IO가 중첩되서 진행되고 있고 그중 하나의 IO가 완료됐을 때 누가 완료됐는지 확인용, 아까 위에서 말했던 키 값을 알려줌

**\*lpOverlapped –** 오버랩IO할땐 오버랩IO 구조체 반드시 사용해야 하는데 IOCP도 오버랩IO이니까 사용해야함, 오버랩 구조체 리턴해줌

뒤에 진도 배우면 여길 통해서 여러가지 정보를 구할 수 있음

**dwMillisocnds –** 이 함수의 타임아웃, 얼마나 기다릴건지 설정

INFINITE 쓰면 IO가 하나라도 종료하지 않으면 무한루프 상태

적절한 숫자를 설정하면 그 시간에 리턴함, 타임아웃이므로 타임아웃 에러코드가 뜸

**BOOL PostQueuedCompletionStatus(**

**HANDLE CompletionPort,**

**DWORD NumberOfByte,**

**ULONG\_PTR dwCompletionKey,**

**LPOVERLAPPED lpOverlapped);**

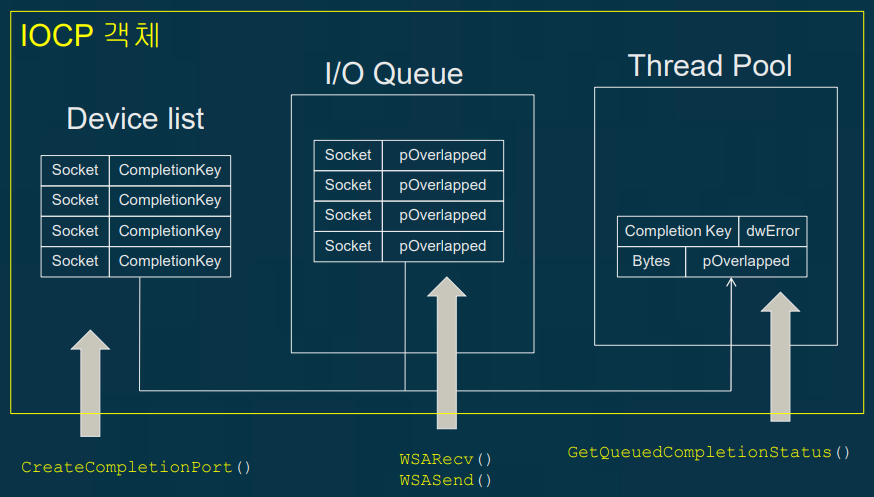
IOCP 객체에서 완료를 한 개 전달해주는 함수

이 함수의 뜻은 파라미터들을 **GetQueuedCompletionStatus**에 넘겨주라는 뜻

**PostQueuedCompletionStatus**를 호출하면 **GetQueuedCompletionStatus**가 전달받은 파라미터들로 그대로 리턴함, 검사 하나도 안함 진짜 넘겨준 그대로 리턴함

그래서 이건 프로그래머가 임의의 값을 넘겨주는 용도로 사용

스레드 관리에 유용하게 사용됨, 나중에 NPC 할 때 쓸거임



그래서 이 IOCP 함수들을 사용하면 이런 구조가 형성된다.

**<IOCP 서버 설계> (싱글스레드 기준)**

1. 초기화

바인드 리슨 등등 + IOCP 핸들 생성

1. 리슨 소켓 IOCP에 등록

Accept 해야되는데 그러면 클라가 연결될때까지 멈춰있어야함

물론 겜서버는 accept 기다려야되는데 그 와중에 send recv 끊임없이 해야함

멀티스레드라면 Accept 전용 스레드 하나 열면 되는데 싱글스레드이니까

오버랩IO로 accept 받아야함, 그러기 위해선 리슨 소켓을 IOCP에 등록하고 AcceptEX (오버랩IO 버전의 accept) 호출해야함

1. 서버 메인 루프

새로운 소켓 접속하면 IOCP 연결 후 WSARecv 호출

WSARecv 완료되면 클라이언트에서 준 데이터로 패킷 처리

WSASend 완료되면 자료구조 정리(메모리 재사용)

**<Accept 처리>**

Accept는 블로킹 처리되면 안 되므로 AcceptEx 사용해서 비동기 처리

리슨 소켓을 IOCP 등록하고 AcceptEx 호출하면 IOCP 객체에서 완료 알려줌

완료되면 무한 루프에서 클라이언트 정보를 저장하는 클라이언트 객체를 만들고

접속된 소켓은 IOCP에 등록, 그래야 클라이언트가 IOCP로 동작을 함

그다음 WSARecv 호출하고 오버랩IO recv callback을 만들어서 동작시켜야함

그리고 다시 AcceptEx() 호출

**<클라이언트 객체>**

서버는 클라 정보 가지고 있는 객체가 필요함

ID, 네트워크 접속 정보 (소켓, 네트워크 버퍼), 연결 상태, 게임 정보 (hp, 위치)

**GetQueuedCompletionStatus** 리턴 값 받았을 때 클라이언트 객체를 찾을 수 있어야함

IOCP에서 오고 가는건 key, 총 데이터 크기, 오버랩IO 포인터 이므로 key를 클라이언트 객체의 포인터로 하거나 클라이언트 객체의 ID 혹은 Index로 하면 된다.

주로 클라이언트 ID로 많이 함

**<오버랩 구조체>**

모든 send recv는 오버랩 구조체가 필요함, send recv는 순서대로 아닌 동시다발적으로 일어남, 그래서 구조체가 스크립 당 하나만 있으면 되는게 아니라 여러 개가 필요함

왜냐면 send recv가 중첩되서 실행되고 있는데 하나의 구조체를 동시에 여러 호출에서 사용하는 것은 불가능하다

왜 불가능하냐, 데이터가 에러가 남, 호출할땐 안나는데 동작하면서 에러 및 데이터 깨짐

여러 개의 구조체를 어떻게 관리하고 할당할지 잘 생각해야함

소켓당 recv 호출은 무조건 한 개여야함, 여러 개일 경우 어떤 데이터가 어떤순서로 들어올지 모름, 두개 이상 호출해도 두번째부터 에러가 날거임

소켓이 100개면 100개의 호출이 중첩되야 하고 다 각각의 다른 소켓에 대한 recv호출

그래서 클라이언트 recv 호출용 오버랩 구조체 하나 만들어놓고 돌려쓰는 게 좋음

왜냐? New/delete 오버헤드 줄이기 위해

소켓당 send 호출은 여러 개가 될 수 있음 -> send 버퍼도 같은 개수만큼 필요함

그리고 개수를 알 수 없으므로 new/delete 사용해야함

Send 할때 new, send가 complete되었을 때 delete -> 근데 new/delete 자체가 별로 좋은 방법은 아니니까 Free list나 available list 사용해서 따로 관리하는 게 성능 더 좋음, 만약 따로 관리할 시 send 버퍼뿐만 아니라 오버랩 구조체도 같이 관리해줘야함

그래서 그냥 send 버퍼랑 오버랩 구조체를 같이 묶어서 관리할거임(제일 흔한 방법)

**<오버랩IO 포인터의 확장>**

오버랩IO 구조체는 반드시 필요함, 그러나 구조체 자체에서 쓸만한 정보는 없음

커널이 사용하는거지 게임서버 내에선 안씀, 그래서 오버랩IO 구조체 자체에다가 우리가 필요한 정보를 넣을 공간은 없음, 실습이랑 과제에서 hEvent 안에다가 소켓을 넣는데 표준도 아니고 안전하다는 보장도 없음, 그래서 구조체를 사용을 해야되는데 마땅히 방법이 없음 -> 꼼수 써야함, 우리는 정보가 필요하기 때문에 구조체를 써먹어야함

IOCP 등록 완료 후 **GetQueuedCompletionStatus**에서 주는 정보만으로 완료처리 해야하는데 num of byte, key를 알려주므로 key를 통해 어떤 소켓인지는 알 수 있는데 가장 중요한 send 인지 recv 인지 accept인지 알 수가 없음 그래서 정보를 추가 해야함

오버랩IO 구조체 안에는 추가할 수 없음, 구조체에 정보를 덧붙여서 전달 해줘야함(확장)

**GetQueuedCompletionStatus**에서는 IO가 완료되었다, 어느 소켓이다, 몇 바이트가 전송되었다 -> 그러나 그 바이트가 어느 버퍼인지(버퍼 위치) 안 알려줌

recv같은 경우는 버퍼가 한 개니까 거기서 확인 가능한데 send인 경우 어떤 버퍼에서 완료되었는지 알 수 없음. 버퍼의 내용은 알 필요 없지만 버퍼의 위치는 알아야함

왜? Send에 사용된 버퍼는 재사용 해야되므로, new했다하면 위치를 알아야 delete 가능

메모리 pool로 관리한다치면 위치를 알아야 반환 가능함

**<완료 처리>**

**GetQueuedCompletionStatus**를 통해 완료 정보를 얻고 에러 났을 경우 에러처리

소켓이 close 되면 접속종료처리 해줘야함

확장 오버랩IO 구조체 통해서 알아낸 send recv accept를 처리해줘야함

**Accept인 경우**

새 클라이언트 객체를 등록하고 recv 호출함

**Recv인 경우**

패킷이 다 왔는지 검사 후 다 왔으면 패킷 처리

여러 개의 패킷이 한 번에 왔을 때 하나씩 처리

패킷이 중간에 잘릴 수 있으니 다음 recv에서 합쳐서 처리-> 패킷 재조립

Recv 처리 끝나면 recv 호출을 더 해서 오버랩IO 상태 유지

**Send인 경우**

Send는 누가 호출하냐? Recv가 호출을 함

클라이언트가 패킷을 보내면 recv에서 처리를 하고 다른 클라이언트한테 정보를 알려줘야 하거나 본인 소켓에 응답을 해야할 때 send하고

Send 완료 후 send할 때 사용한 오버랩 구조체랑 버퍼를 free or 재사용 해야함

**<버퍼관리>**

**RECV** 버퍼 관리 잘 신경써야함

하나의 소켓에 대한 recv 호출은 언제나 하나이기 때문에 하나의 버퍼만 계속 사용

근데 패킷들이 중간에 짤린 상태로 도착할 수 있어서 recv 버퍼를 함부로 비우면 X

그럼 읽은곳 뒤에 이어서 읽어와야 되는데 그러면 recv 버퍼 관리가 어려워짐

그래서 일단 읽은 거를 저장해두는 버퍼가 추가로 필요함

그게 귀찮으면 ring buffer 사용->복잡해지지만 데이터 복사가 줄어서 성능이 올라감

우리는 링버퍼 안쓸건데 본인이 IOCP 이해도가 높아지면 링버퍼 시도해볼 것

**SEND** 하나의 소켓에 여러 개의 send 동시에 할 수 있음

클라이언트는 여러 개임, 클라 하나가 상태가 바뀌었을 때 다른 모든 클라이언트에서 브로드캐스팅 된다. 그래서 클라들은 본인에 관련된 send와 다른 클라 관련된 send를 동시다발적으로 받음.

가장 간단한 방법은 send 할때마다 오버랩 구조체와 버퍼를 new/delete로 할당

아니면 클라마다 send형 구조체를 만들어서 send 할때마다 queue에 쌓아놨다가 하나씩 send를 한다치면 프로그래밍도 복잡해지고 여러 개의 send를 동시에 보낼 수 없어서 성능저하 발생함-> 오버랩 구조체와 버퍼를 다이나믹하게 할당을 받아서 비동기 식으로 그때그때 send 해줘야함

근데 소켓이 하나 있는데 여러 개의 클라이언트가 동시에 send를 하면 데이터가 섞이지 않냐? -> 그렇지 않음, send를 실행한 순서대로 버퍼에 넣어서 전송하기 때문, 버퍼에서 어떤 순서로 보내질지는 모르지만 확실한건 데이터가 섞이진 않음

내부 버퍼가 차서 send가 중간에 잘렸는지는 number of byte로 확인 가능

아까 위에서 순서는 모르지만 데이터는 안섞인다 했지만 내부 버퍼가 차서 잘린 경우에는 섞여버림 (100바이트 중 70바이트 보내고 다른패킷 보내고 남은 30보낼 경우) 그래서 위처럼 모아서 차례차례 보내야함, 버퍼에 모아놨다가 차례차례로 보내고 패킷이 하나가 완료가 되면 다음꺼 보내고 하는식으로 -> 결국 성능 저하

두번째는 중간에 데이터가 짤렸다? 그러면 그냥 클라 접속 종료시키는 것도 방법

그런데 충격적이게도 두번째 방법을 써야함 왜냐?

커널의 내부 버퍼가 다 차서 내부 버퍼로 send 데이터를 다 복사 못했다는 뜻임

즉, 운영체제에 할당된 데이터를 다 썼다는 얘기고 그 말은 new 나 말록조차 관리를 못해줌, 결국 게임서버가 메모리를 너무 많이 사용한거임 -> 게임서버가 오동작을 해서 제대로 실행될 가능성이 없다는 뜻임 -> 결국 서버 셧다운이 답임

**<오버랩IO에서 IOCP로 추가할 것>**

다중 접속 관리할 방법 추가 및 패킷 포맷, 프로토콜 정의 해야함

기본 패킷 : 길이 + 타입 + 데이터 순서

그래서 자주 주고받아야 하는 패킷은 사이즈를 작게하고 길드 정보, 파티원 정보 같이 정보량 많은건 패킷 사이즈 크고 한 번씩 주고받게 설정

클라 -> 서버: 로그인 요청, 이동 패킷, (스킬 사용, 아이템 사용은 추가 구현)

서버 -> 클라: 로그인 수락(캐릭터 초기화 같이), 위치 지정, ID 접속 알림, 로그아웃 알림

(추가 요소: 상태 이상, 레벨 업, 마법 스킬 등), 실제 게임은 패킷이 100종류 정도 있음

**<프로토콜 정의>**

어차피 서버랑 클라 둘다 PC이므로 구조체가 편함, 모든 패킷은 chat size랑 char type이 맨앞에 있음, int id는 누가 접속을 했다, 누가 어디로 이동을했다 할 때 누가에 해당됨

근데 만약 숫자로 관리 안하고 캐릭터 이름으로 id를 저장한다 -> 문제가 많아짐

문자열로 되면 길이가 길어져 네트워크 대역폭(bandwidth)이 낭비된다

여기서 대역폭이란 일정 시간 내에 데이터 연결을 통과할 수 있는 정보량의 척도임

Char o\_type을 통해 이게 플레이어인지 npc인지 구분, char name을 통해 이름 표시

#pragma pack(push, 1) 패킷에 빈공간 다 지워버리는 역할, 이걸 안하면 패킷 내부에 변수들이 int형에 맞춰서 데이터 크기 설정해버림

**<패킷 처리 루틴>**

**bool PacketProcess(const unsigned char\* pBuf, int client\_id);**

클라->서버, 서버->클라 패킷이 왔을 때 어떻게 처리할지 정해주는 함수

**<Recv 구현>**

**OverEx** 오버랩 구조체 안에 추가할 것

**unsigned char m\_IOCPbuf[MAX\_BUF\_SIZE];** -> IOCP recv/send 버퍼 추가

**enumOperation m\_Operation;** -> send/recv/accept 구별용

**class ClientInfo** 안에 추가할 것

**unsigned char m\_recv\_packet\_buf[MAX\_PACKET\_SIZE];**

recv 할때 데이터 크기가 커서 중간에 끊길 때 위에 오버랩 구조체 안에 있는 버퍼에 있는 데이터를 보관하기 위한 버퍼

**int m\_prev\_size;** ->그리고 그 버퍼의 크기 저장, 위에랑 얘는 사실상 패킷 재조립용

**<패킷 재조립>**

리시브가 완료되었다 -> 루프 돌면서 패킷 재조립을 함

Start:

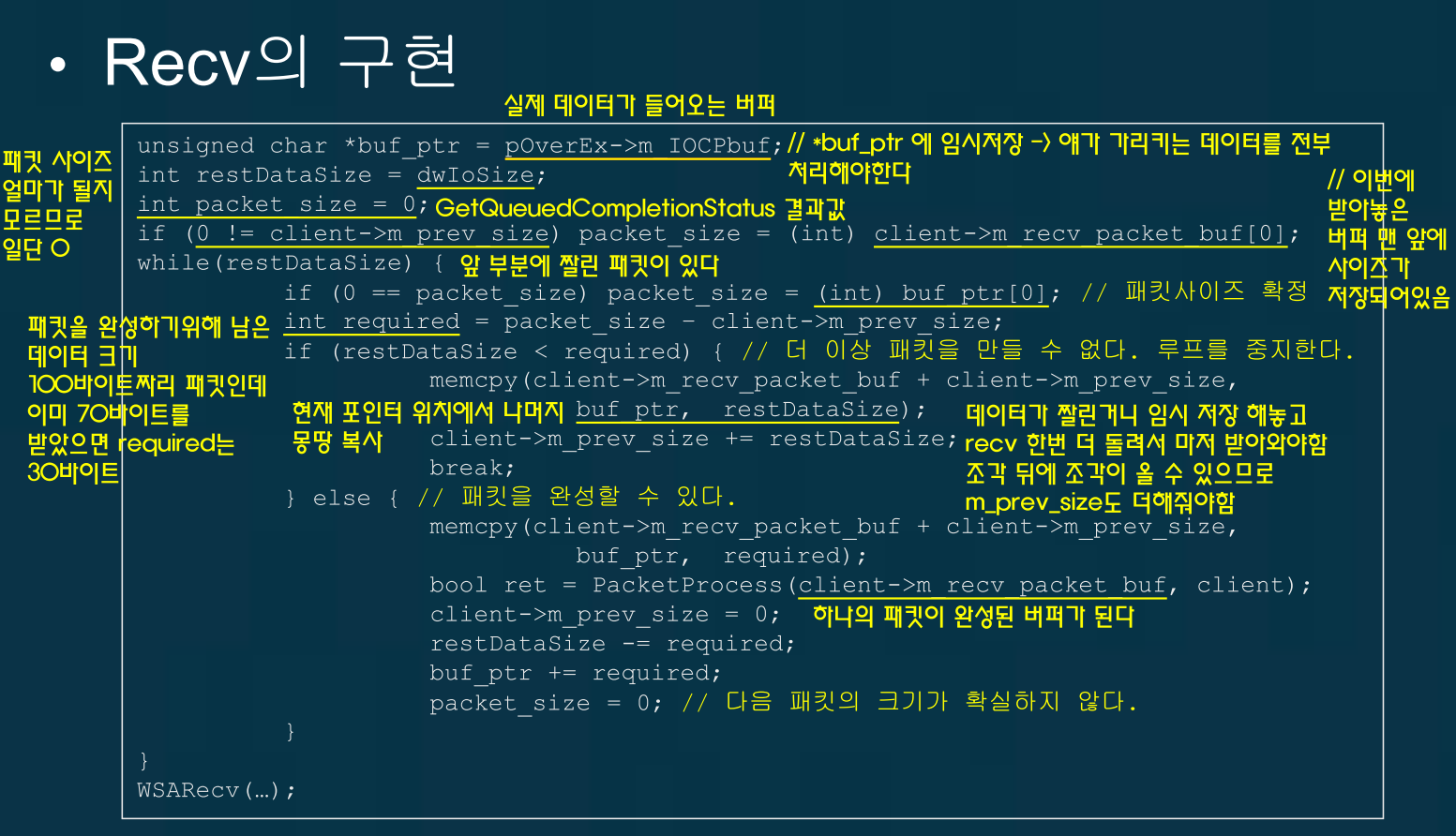
1. 모든 데이터를 처리했으면 End
2. 모든 데이터를 처리하지 않았다 -> 처리하지 않은 남은 데이터로 패킷 완성 가능?

예 : 패킷 버퍼 완성시키고 패킷 처리 함수 (process packet) 함수 호출, 이후 Start

아니요 : 이 경우 남은 데이터가 있는지 확인, 남은 데이터가 있다면 이건 짤린 패킷의 앞부분이라는 뜻, 그럼 패킷 버퍼로 모두 보내버리면 된다, 이후 Start

End:

Recv 호출

**<Recv 패킷 재조립 구현 + AcceptEx 함수>**

**SOCKET clientSocket = accept(listenSocket, (sockaddr\*)&clientAddr, &c\_AddrSize);**

**AcceptEx(listenSocket, clientSocket, buffer, NULL, sizeof(sockaddr\_in) + 16, sizeof(sockaddr\_in) + 16, NULL, over);**

기존의 accept는 리슨 소켓에서 소켓을 받아서 새로운 소켓을 하나 만들어서 리턴

매개변수1 : 리슨소켓

매개변수2 : 그러나 AcceptEx는 새로 만들어서 리턴 안하고 미리 소켓을 만들어서 거기다가 클라를 연결함

매개변수3 : accept를 받으면 연결된 클라 주소를 넣어줄 버퍼

접속을 할때 클라 주소도 필요하지만 클라가 가지고 있는 여러 카드중 어떤 네트워크 카드로 접속했는지 알아야해서 수신, 송신 주소 둘다 필요함

매개변수5,6 : 수신주소, 송신주소 (+16은 여유공간 마련)

매개변수7 : 버퍼에 주소가 몇바이트 들어갔냐, 근데 우리는 이거 안씀, 받는다 치더라도 여기서 처리 안할거임

**<WSASend>**

Send 하는 오버랩 구조체와 버퍼는 끝날 때까지 유지되어야 하고

오버랩 구조체와 버퍼 개수는 미리 할 수 없어서 다이내믹하게 관리해야 함

Send할 데이터가 100바이트인데 70바이트만 보내지고 30바이트가 남는 부분 send가 일어난 경우는, 즉 버퍼가 비워지지 않은 경우는 에러 처리하고 끊어버려야 함

이건 운영체제의 메모리가 꽉 찬 경우라는 거임, 이런 일이 일어나면 이미 답이 없는 경우이므로 이런 일이 안 일어나게끔 예방 해야함 -> send 데이터량 조절해야함

Send를 너무 많이 하거나, 클라로부터 받는 recv 속도가 너무 느리거나

-> 클라에서 보낸 데이터 빨리 처리, send 될 수 있으면 적게

-> 패킷을 자주 안보내고, 보내더라고 데이터량 적게 설계

근데 아무리 그렇게해도 최소한으로 필요한 양의 패킷이 있을거임

아무리 줄여도 recv가 쌓이고 send 공간이 없어서 메모리가 모자란다 -> 너무많은 동접

프로그램이 감당 가능한 동접을 넘어섰다는거임 -> 동접을 낮춰야함

**<Send 완료 구현>**

**if (dwIoSize < pOverEx->m\_IOCPbuf[0])**

**Disconnect(client);**

**delete pOverlappedEx;**

send는 무조건 패킷 통으로 보내야하는데 **dwIoSize < pOverEx->m\_IOCPbuf[0]**는 넘버 오브 바이트가 패킷 크기보다 작은 경우이므로 문제가 있다는거임 -> 연결 끊기

**<멀티 플레이어 구현>**

OP\_ACCEPT에서 새 플레이어의 위치를 다른 플레이어들에게 전송

다른 플레이어들의 위치를 새 플레이어에게 전송

do\_move에서 이동할 때마다 새 좌표를 모든 플레이어에게 전송

GetQueuedCompletionStatus() 에서 접속 종료 시 다른 모든 플레이어에게 알림

**<병렬처리>**

하나의 작업을 여러 개의 콘텍스트에서 수행

콘텍스트 = 병렬로 수행하는 단위

콘텍스트 -> CPU의 실행 상태

콘텍스트가 여러 개다 -> 여러 개의 CPU가 동시에 실행된다

CPU가 한 개 있는데 프로세스가 여러 개일 경우 CPU가 옮겨다니면서 실행하고

CPU가 실행될 때 지금 실행 중인 상태를 저장 해야되고 그게 바로 콘텍스트

어디에 저장되냐, PCB에 저장된다, 프로그램 카운터 및 모든 레지스터 값을 저장

이 콘텍스트 원래 하나인데 프로세스에 콘텍스트가 여러 개 있는게 병렬처리

PC (Program Counter) (명령어 포인터): 마이크로프로세서 내부에 있는 레지스터 중 하나

다음에 실행될 명령어의 주소를 가지고 있어 실행할 기계어 코드의 위치를 지정

PCB (Process Control Block) : 운영체제가 프로세스를 제어하기 위해 정보를 저장하는곳

프로세스 상태 관리와 문맥교환을 위해 필요

프로세스 생성시 만들어지고 주기억장치에 유지

벙렬처리 하는 이유 : CPU의 처리속도가 느려서 성능을 높이기 위해, 프로그램 구조가 깔끔해진다 또는 읽기 쉬워진다라는 말이나 이유로 멀티스레드 만들지 말 것

클럭 속도가 2002년 대략 2.4g, 2020년 대략 3.0g임, 하드웨어 용량 증가에 비하면 턱없이 증가량이 낮음, 그리고 라이젠이 경쟁에 뛰어블면서 16코어 32코어 등 멀티코어 CPU 보급이 뻥튀기 됐다, 핸드폰마저 멀티코어임

**<프로세스와 쓰레드>**

프로세스 : 실행 중인 프로그램, 프로그램은 하드디스크에 있는 파일, 그걸 실행하면 운영체제 안에서 프로세스가 되어서 실행이 된다, 실행 파일은 하나여도 여러 번 실행시키면 프로세스가 여러 개, 파일 실행-> 메모리 할당-> CPU할당-> 프로세스

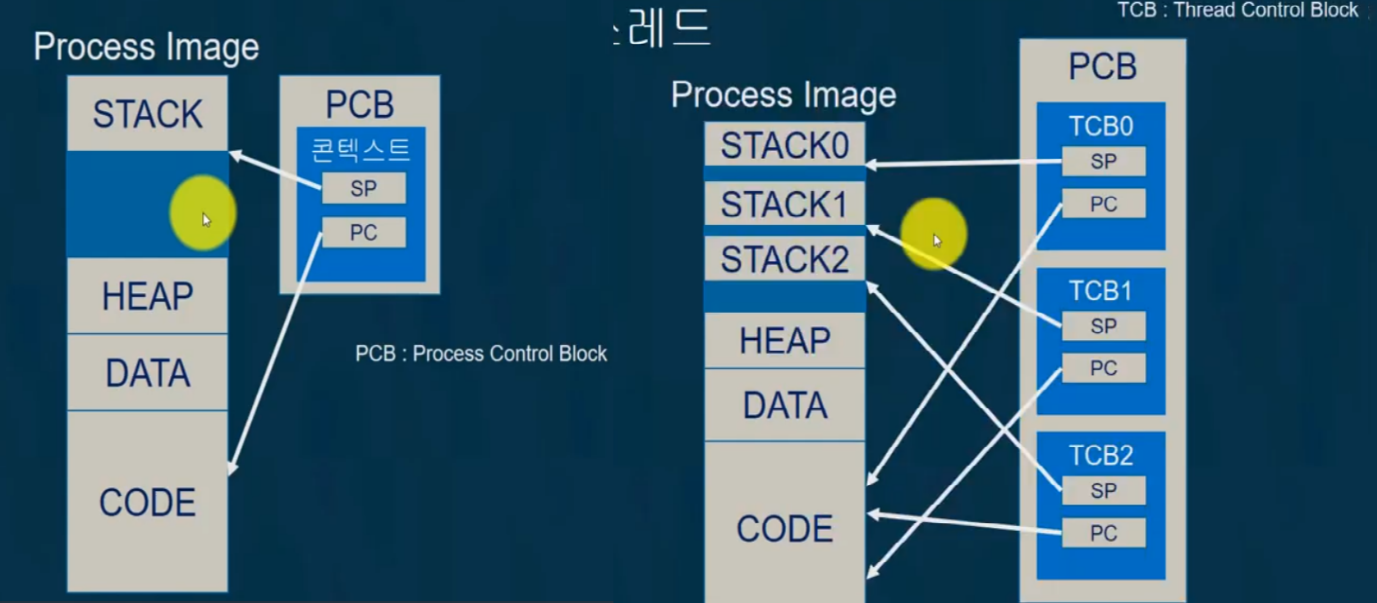
쓰레드 : 프로그램 실행의 흐름 (메인부터 시작해서 프로그램 종료까지)

프로세스가 프로그램을 실행시킨다->프로세스 안에 쓰레드 하나 있는거임

멀티쓰레드 : 프로세스 실행-> 프로그램 실행-> 프로그램 안에 쓰레스 생성 명령이 있음

그리고 쓰레드마다 자신의 스택을 가지고 있어서 스택은 여러 개지만 Data(전역변수), Code(실행 프로그램), HEAP은 하나밖에 없어서 같은 프로세스의 모든 쓰레드가 공유함

Sp : 스택 포인터, pc : 명령어 포인터



전역 변수는 모든 쓰레드가 공유하고 지역변수는 쓰레드마다 따로 존재하는데

지역변수도 강제로 공유하려면 가능함(전역 변수에 지역변수 주소 저장해서) 그러나 그러지 말 것, 그렇게 하면 메모리 leak 및 디버깅이 상당히 어려워진다

모든 자원(메모리, 파일 핸들, 윈도우 핸들 등등)은 공유된다

멀티쓰레드 프로그래밍

장점 : 성능 향상(핵심), 빠른 응답 속도, 더 나은 자원 활용(CPU Core), 프로세스보다 효율적인 통신 (프로세스끼리 데이터 주고받는거 보다 쓰레드끼리 주고받는게 더 빠름, 문맥 전환도 프로세스끼리 보다 쓰레드끼리 하는데 오버헤드가 적음)

단점 : 통신 코드가 추가되면서 프로그램 복잡도 증가, 디버깅이 어려움

게임서버에서는 동접을 늘리기 위해, 빠른 응답 속도를 위해 멀티쓰레드 사용

**<멀티쓰레드 프로그래밍의 스타일 2가지>**

1. 헤테로 제니오스 (Heterogeneous)

작업을 쪼개서 종류별로 다른 쓰레드에게 맞기는 스타일 (언리얼 엔진)

로직(AI)쓰레드, 렌더링 쓰레드,사운드 쓰레드,특수효과 쓰레드, 물리엔진 쓰레드 등등

장점 : 싱글스레드에서 멀티쓰레드로 바꾸기 힘듬, 그러나 헤테로제니오스는 따로 함수처럼 구현 해놓았기 때문에 멀티쓰레드로 전환이 그나마 쉬움, 다른 쓰레드들간의 메쏘드 호출만 잘 시키면 된다

단점 : 쓰레드마다 하는 일이 달라서 bottle neck 쓰레드가 있을 수 있음, 딴 스레드가 아무리 일을 잘해도 병목 스레드가 속도가 안나오면 다같이 느려짐, 즉 성능 향상에 한계가 존재함, 코어가 많을 때 다 활용하기 힘들다, 그래서 처음부터 멀티쓰레드로 구현하겠다 하면 이 방법보단 호모 지니오스 방법을 더 많이 사용함

1. 호모 지니오스 (Homogenous) (event driven) (data driven)

작업을 쪼개서 정렬해놓고 작업 처리해야할 순서대로 쓰레드들에 일을 하나씩 가져가서 처리하고 끝나면 남은 다음일을 처리하고 해서 모든 쓰레드들이 놀지않고 모든 작업을 풀로 실행하는 방식

게임 서버는 호모지니오스 방식이다, 단위를 쪼개는 단위가 패킷임

클라에서 패킷이 오면 쓰레드 붙여서 처리하고 다음 패킷오면 붙여서 처리하고

동접이 5000이면 최대 5000개의 조각이 날라옴, 그걸 쓰레드가 하나씩 잡아서 처리

패킷이 어느 쓰레드에서 처리할지 안 정함, 그냥 노는 쓰레드 있으면 걔 가져다가 처리

게임 클라는 헤테로지니어스 일 경우가 높음, 렌더링 쓰레드가 주로 병목 쓰레드임

그래서 fps가 달라지지 않음, 이걸 해결하려고 다렉X12 같은 경우는 벌칸이라고 렌더링 쓰레드 자체를 여러 개의 멀티쓰레드로 나누는 식으로 해결함

**<멀티쓰레드 사용할 때 주의점>**

쓰레드가 많다고 좋은게 아님, 쓰레드가 많으면-> 스텍이 차지하는 메모리가 많아지고-> PCB안에 TCB가 많아짐 -> 문맥 전환할 때 비교해야할 대상이 많아지면서 운영체제에 부담이 많아짐

그리고 쓰레드가 많아도 코어 개수만큼만 실행됨, 코어가 8갠데 쓰레드가 100개면 나머지 92개는 waiting 상태임 -> 문맥 전환 오버헤드 때문에 더 느려짐

교수님 회사 다닐땐 코어수 x 1.5배 한 개수가 제일 성능이 잘나왔음

그리고 운영체제, 하드웨어에 부담가고 자료구조 중복되면서 캐스 미스가 남

내가 사용하는 메모리의 내용이 내가 아닌 다른 쓰레드에 의해서 변경될 수 있음

이걸 데이터 레이스(경쟁 상태)라고 부름, 싱글 코어도 마찬가지임, 디버깅도 어려움

**<Windows에서의 멀티쓰레드 프로그래밍>** 얘는 그냥 피피티 참고,.

**<쓰레드 생성 종료>**

#include <thread>

Std::thread t1 { mythread }; 선언 하는 순간 mythread라는 쓰레드가 생성

그러고 그 이후 문장이 계속 진행되면서 void mythread() 내부에 내용도 같이 시작된다

t1. join() 함수를 쓰면 쓰레드가 종료될 때까지 기다렸다가 끝나면 join하고 이후 진행

그래서 프로그램 종료할 때 모든 쓰레드가 join 된걸 확인하고 꺼야함

New / delete랑 같은 개념

멀티쓰레드는 무조건 속도 때문에 사용, 최적화를 진행하고 한단계 진행될 때마다 성능측정 해봐야함

화면에 쓰레드 넘버 출력하는거는 벤치마크 프로그램 불가능

IO가 들어가면 cpu 실행 시간보다 io동작속도가 훨신 느림 -> 전체 속도의 대부분은 IO가 차지 -> 멀티쓰레드 프로그래밍으로 cpu속도를 빠르게 했다고 해도 측정 불가 ->

IO없이 cpu가 계산하는 프로그램을 벤치마킹 해서 그것의 성능을 측정해야 멀티쓰레드로 빨라졌나 안빨라졌나 판단 가능

모든 성능 측정은 릴리스 모드에서 한다

**<Data Race>**

같은 메모리를 한 개 이상의 쓰레드가 동시에 읽고 쓸 때 발생

그 중 적어도 한 개는 반드시 쓰기 일 때

Mutex 객체는 무조건 전역변수 : 쓰레드들이 같은 자물쇠를 써야함

Lock의 개수를 최소화하고, 병렬수행을 최대화 해야하는데 이건 서로 트레이드 오프임, 하나 올리면 다른거 줄어들고.. 그래도 이거를 맞춰줘야 성능이 좋아짐

이유 : 배열이 충돌을 일으킴

메모리를 읽을떄 int를 읽었다고해서 cpu가 메모리에서 4바이트를 읽어가는게 아님

cpu엔 캐쉬가 있고 cpu가 메모리 읽을땐 캐쉬 라인 단위로 읽음

intel의 캐쉬 라인은 64바이트임

그래서 4바이트 데이터들을 코어가 읽을때 4바이트만 읽으면 되는데 64바이트 캐쉬라인을 다 읽음

똑같은 메모리의 주소가 캐쉬 두군대 이상 존재하면 안됨 한명이 업데이트하면 그 값을 다른 한명이 읽어야되는데

각자 캐쉬라인이 있어서 자기껏만 업데이트하고 상대가 업데이트한게 안보임

그래서 intel에선 같인 캐쉬라인을 여러 코어가 공유 못하게 만들어 놨음

그래서 한 코어가 캐쉬라인을 업데이트 하려고 할때 옆에 다른 캐쉬라인을 접근 못하게 막음

그래서 한명 업데이트했다가 내보내고 그걸 다른 코어가 받아와서 업데이트 하고 왔다갔다함

그걸 캐쉬 쓰래슁이라고 함, 그래서 속도가 멀티코어보다 느리게 나옴

alignas(64)를 쓰면 컴파일러에서 메모리에 할당을 할때 캐쉬라인 단위로 떨어트림

**<IOCP 멀티쓰레드화 = Worker\_Thread>**

멀티쓰레드 IOCP서버는 워커 쓰레드가 있어서 **GetQueuedCompletionStatus(완료검사)** 를 여러 쓰레드에서 동시 호출하는 것, 즉 IOCP는 멀티쓰레드에서 돌아가고 멀티쓰레드로 돌아가기 위해서 쓰레드 여러 개를 쓰레드 풀에 넣어야된다

그 역할을 하는게 바로 **GetQueuedCompletionStatus** 함수

이 함수는 이 함수를 실행한 쓰레드를 쓰레드 풀에 넣는 기능과 완료된 IO를 꺼내서 프로그램에 넘겨주는 일 2가지를 동시에 한다.

그래서 **GetQueuedCompletionStatus** 받아서 처리하는 부분을 워커 쓰레드에 넣으면 끝 문제는 데이터 레이스임, 모든 전역 변수가 데이터 레이스의 대상임

ClientInfo g\_clients[MAX\_USER]; = 다른 클라도 브로드캐스팅 하면서 접근

int current\_User\_ID = 0; = 동시에 accept될수도, 겹쳐서 ++ 했는데 +만 될수도 있음

HANDLE g\_iocp; = 얘는 데이터레이스가 아님! Worker thread 이전에 선언한 이후로 값을 바꿀일이 없음, 어떤 스레드도 write를 안하기 때문

ClientInfo g\_clients[MAX\_USER] 내부 변수들을 따지고 보면

SOCKET m\_socket = worker thread에서 값을 받으면 여기다 넣음 = 데이터 레이스

int m\_id = 처음 g\_clients 생성할때 초기화 하므로 데이터 레이스 아님

OverEx m\_recv\_over = 한 쓰레드 내부에서만 처리하고 여러 쓰레드에서 동시 접근 안함

int m\_prev\_size, char m\_packet\_buf[MAX\_PACKET\_SIZE] = 얘네도 recv 처리할때만 접근하므로 위에 3개 다 데이터 레이스 아님

bool m\_isConnected = 데이터 레이스임, 여러 쓰레드에서 true false 처리함

short m\_x, m\_y = 데이터 레이스임, 본인 클라 다른 클라에서 다 접근함

char m\_name[MAX\_ID\_LEN + 1] = 데이터 레이스임, 로그인 패킷을 받을 때 수정되므로

m\_socket, m\_isConnected, m\_name은 mutex로 보호할거임

x,y는 보호 안할거임, 어차피 바로바로 업데이트되고 오류 났을때의 오차가 작음 그러나 실제 게임에서는 텔포, 대륙 건너뛰므로 락 걸어야함, 우리는 좌표 1칸씩 이동해서 ㄱㅊ

current\_User\_ID 값도 한 쓰레드가 accept 끝나면 끝에 AcceptEx를 호출하게 해놔서 여러 쓰레드가 동시 접근 안하지만 이렇게 되면 여러 클라가 동시에 접속할 때 한명한명 접속해서 엄청 오래걸림, 나중에 AcceptEx를 여러 곳에서 호출하게 확장했을 때를 대비해 미리 전역 mutex 락을 걸어둠

ClientInfo g\_clients[MAX\_USER]; 이게 의미하는건 동시 접속자 수임, 그런데 유저들은 나갔다가 들어왔다 할 수도 있음, 그래서 동접자가 5000명이다 하면 주로 5배를 잡아서 왔다갔다 하는 유저는 25000명 정도로 잡는다. 서버도 매일매일 새로 여는 게 아니라 최소 5,6일 단위로 점검하므로 객체를 매번 늘릴 수 없음, 나간 객체 자리에 재사용을 해야함, 그래서 connected를 확인해야함, true인 애는 재사용 하면 안 된다

그런데 connected가 true되는 과정이 1. Initclient 에서 false로 초기화 -> 2. AcceptEx로 완료 -> 3. ID할당 -> 4. CS\_PACKET\_LOGIN 패킷 수신 (이걸 수신한다는 거 자체가 클라이언트 측이 로그인 할 준비가 되있다는 의미임, accept했다고 정보 보내면 날라감) -> 5. 이후 connected =true (CS\_PACKET\_LOGIN 패킷 수신을 해야 플레이어 정보를 알 수 있고, 그래야 다른 클라에 표시를 할 수 있음

여기서 생기는 문제가 뭐냐, 만약 클라 한 개가 accept하고 connected가 false인 id를 할당받고 cs\_login 패킷을 아직 안보내고 있는 상태이면 아직 connected가 true가 아님, 그 상태에서 다른 클라가 accept되고 id를 할당 받을 때 이전에 할당해준 클라가 아직 connected가 true가 아니므로 같은 id를 새로 accept된 클라한테 할당해서 2클라가 같은 객체에 저장되는 문제가 발생 가능

그럼 ID 할당해줄 때 connected를 true로 바꾸고 cs\_login패킷 수신하면 되지 않냐? 그러면 클라가 다른 클라 정보 읽어올 때 connected true로 판단하는데 cs\_login 패킷 수신을 안해서 정보가 없는 클라의 정보를 읽어오려고 할꺼임, 그러면 또 다른 클라 정보 읽어오는 부분에서 예외 처리해줘야 하는데, 그럴 바엔 차라리 connected 말고 추가로 flag가 하나 더 있던가 클라 객체 상태를 구분 짓던가 해야함.

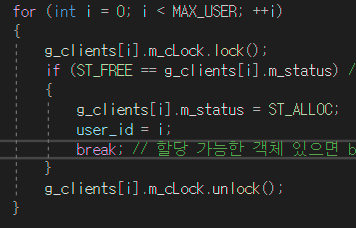
우리는 후자 방법으로

FREE: 아무도 사용하지 않음, 할당 가능, 객체(객체 정보)에 접근 금지임

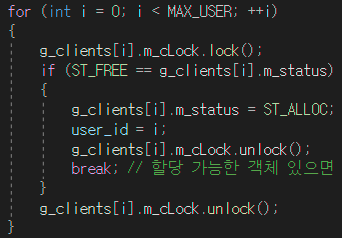
ALLOCATED: 예약되었음(accept 되있음), accept 되어서 아이디가 이미 할당됐으므로 할당 불가능, 접근 금지(cs\_login 처리가 안끝났으니 데이터를 보내지도 말고 객체 내 정보도 읽어가지 마라)

ACTIVE: 할당 불가능, 접근 가능(lock 건 후에)

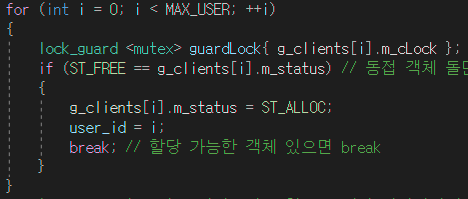
**<lock\_guard <mutex> 객체명 {뮤텍스 락};>**



For문에 락 걸 때 이런 경우 발생 가능함, break 타버리면 unlock이 안된다



그래서 이게 정석임, 조건부 만족할 때 unlock해주고 만족 안할 때 unlock해서 총 2번



C++11 기능 중에 lock\_guard라고 써주면 이 cLock를 락을 걸고 락가드가 속한 블록에서 빠져나갈때 unlock해주고 루프 돌때마다 unlock-lock 해줌

템플릿 클래스라서 생성자에서 lock, 소멸자에서 unlock 한다/ break, continue도 가능

**< volatile** **>**

데이터 레이스 없애는 가장 직접적인 방법은 락을 사용하거나 데이터 레이스를 없애도록 프로그램을 변경하여 줄일 순 있지만 완벽하게 없애는 건 힘들다.

락을 안쓰고 잘 피해가는 프로그래밍을 해도 많은 함정들이 있음

협업할 때 멀티쓰레드를 사용하는 것은 병렬화로 성능을 향상시키는 것도 있고 같이 실행되는 쓰레드들끼리 동기화를 해야 되기 때문에 사용하는 거다.

동기화는 쓰레드끼리 데이터를 주고받는 행위로 공유 메모리는 통해 주고받는 방법밖에 없음, 시스템 콜을 통해서 주고받을 수 있지만 오버헤드가 뮤텍스보다 더 큼

컴파일러 최적화로 인한 사기( ready == false 무한 루프에 빠지던)를 안당하려면 **volatile** 사용하면 된다, 얘는 최적화를 진행해도 직접 메모리에 읽고 쓰고 레지스터에 할당하지 않으며 읽고 쓰는 순서를 지키게 된다. 그래서 멀티쓰레드를 제대로 알려면 어셈블리를 알아야한다, 안그러면 비쥬얼 스튜디오의 최적화나 마음대로 순서 바꾸는 행위를 모름.

소스 코드 만으로 해결이 되면 상관없지만 아무리 봐도 이상하게 돌아간다 싶으면 어셈블리를 확인하면 된다

최적화할 때 메모리 접근을 줄이도록 최적화할 수 있고, 메모리 접근을 줄이게 되면 동기화가 안될수도 있다, 그래서 멀티쓰레드 프로그래밍을 할 때 프로그램 자체의 알고리즘뿐만 아니라 우리가 읽고 쓰는 게 실제로 메모리에 들어가는지 신경써야 한다. 데이터 레이스 때문이고 락을 걸지 않아서 그렇다.

volatile int\* a 선언하면 a라는 포인터 변수는 volatile 포인터가 아니라 volatile한 int를 가리키는 포인터다 => \*a=1(a가 가리키는 곳에 1을 넣는다)는 정상 메모리에 접근동작 a=b(a의 주소에 b의 주소를 집어넣는다)는 레지스터로 작동할 수 있음

int\* volatile a 선언하면 int를 가리키는 volatile 포인터 a라는 뜻 => a\*=1하면 순서를 안지키고 레지스터 접근할 수 있음, a=b는 a라는 volatile 포인터를 선언했으므로 최적화하지 않고a 메모리에 무조건 b를 넣어줌

그래서 결론은 포인터가 있을 때 volatile 쓸꺼면 포인터가 volatile일지 포인터가 가리키는 값이 volatile일지 생각 잘 해서 써줘야함

**<메모리 일관성 문제>**

소스 코드도 문제가 없고, volatile써서 메모리 문제도 없고 어셈블리도 문제가 없는데 이상하게 돌아간다 => CPU가 문제 일으킨거, out of order execution이랑 write buffering

out of order execution: 비순차적 명령어 처리 또는 비순차적 실행은 고성능 마이크로프로세서가 특정한 종류의 지연으로 인해 낭비될 수 있는 명령 사이클을 이용하는 패러다임이다. 명령 실행 효율을 높이기 위해 순서에 따라 처리하지 않는 기법이며 수많은 프로세서가 채용하고 있다.

write buffering : CPU가 메모리에 쓰려고 하는데 메모리가 바쁘다, 캐쉬 미스가 나서 당장 사용 불가능하다, 이면 캐쉬에 내가 쓰고자 하는 주소가 올라올 때까지 기다리지 않고 버퍼에 놔두고 CPU대로 따로 동작을 한다, 그러고 캐쉬에 올라오면 write 버퍼에 있는 게 캐쉬로 들어옴, 즉 write가 딜레이가 일어나는거임. 멀티쓰레드 프로그래밍은 cpu가 어떻게 동작하냐가 큰 영향을 끼친다

싱글코어에선 절대 들킬일이 없게 가려버린다, re order버퍼를 이용해서 실행은 out of order로 했지만 명령의 완료는 순서대로 한다, write buffering도 CPU가 읽을 때 메모리에서 안읽고 write buffer안에 내가 읽으려는 값이 있으면 그걸 읽음

그러나 멀티코어에선 한 코어가 실행은 엉망으로 해도 완료는 제대로 한다 쳐도 옆에 코어는 실행만 보고 완료를 안봄, 메모리 읽을때도 자신의 write 버퍼안에 검색을 하지만 다른 코어에서 읽으려고 할 때 다른 코어의 write 버퍼를 확인 못함

그래서 피터슨 알고리즘이 제대로 작동 안함, 이게 바로 메모리 일관성 문제

\_asm mfence를 이용하면 이 앞뒤에 있는 명령은 서로 아웃오익과 라이트버퍼 문제가 발생하지 않음, mfence를 쓰면 그 위에 있는 애들이 아웃오익 실행이 다 종료되면 아래로 넘어가고 write 버퍼가 다 비워졌을 아래 명령을 수행함

C++ 표준에는 atomic\_thread\_fence (std::memory\_order\_seq\_cst)를 사용하면 위에 꺼를 시행해줌, 그래서 뮤텍스를 사용하면 메모리 일관성 문제, 메모리 접근 문제 다 해결된다

뮤텍스를 사용 안하려고 하니까 이러한 문제가 발생하는 것

**< Cache Line Size Boundary >**

Cpu 파이프라인이 순서대로 진행중일 때 중간에 순서가 끝도 없이 미뤄지게 되어 속도와 진행시간이 길어지는 경우가 발생 가능하다 = 파이프라인 해저드

올바른 결과를 위해서 뒤로 미뤄다보니 너무 길어지는 경우의 수를 없애기 위해 생겨난 것이 out of order

64바이트 캐쉬 라인 맨 앞으로 옮겨놓은 주소를 -2바이트 옮기면 2개의 캐쉬라인을 걸치게 된다, 그래서 읽고 쓸때 2개의 캐쉬라인을 왔다갔다하면 읽고 쓴다, 그래서 한 코어가 윗 캐쉬라인을 읽고 아래 캐쉬라인을 읽기 전 그 사이에 다른 코어가 읽어버리면 최종값과 초기값이 아닌 중간값이 읽어져버림

즉, 우리가 접근하려는 오브젝트가 캐쉬 라인에 걸쳐지면 최종값과 초기값이 아닌 중간값이 나온다.

그래서 포인터가 아니라 우리가 일반적으로 선언해서 쓰는 변수는 비쥬얼 씨플플이 알아서 4바이트 단위로 잘라서 저장을 한다. 그러나 포인터로 조작을 하게되면 캐쉬라인 걸치는 현상이 발생할 수 있음

#pragma pack 할때도 조심, 얘 쓰면 4바이트 단위로 할당이 안되므로

그리고 메모리 아끼기 위해서 배열을 선언하고 포인터로 그 배열이 값을 업데이트하면 중간값 발생할 수 있으므로 조심해야함

**<Atomic>**

<atomic> 라이브러리 선언으로 사용가능, 쓰레드마다 다 다른 순서로 보는게 아니라 모든 쓰레드가 봤을 때 메모리 업데이트가 똑같은 순서대로 발생하는 메모리

아토믹은 기본 자료구조만 가능함, 벡터-스텍-큐 같은 심화된 자료구조는 못쓴다

그러나 서버에선 벡터-스텍-큐 같은 심화된 자료구조를 이용하여 데이터를 동기화함

그럼 그 자료구조 내부에 기본 변수를 아토믹으로 선언하면 되지 않냐, 그렇지 않다.

자료구조 자체가 아토믹 해야지 아토믹 변수들이 모여있다고 저절로 아토믹 자료구조가 되지 않음, 그럼 아토믹 한 자료구조 구현을 위해선 Lock으로 아토믹 자료구조를 만들면 된다, 그러나 그래되면 성능 저하가 일어남.

결국은 lock이 문제임, lock 자체가 상대방 스레드가 하는 일을 끝날때까지 기다려야하게 해서 동시실행으로 인한 성능 개선을 얻기가 힘들어진다. 그래서 우리는 상대방 스레드 행동에 의존적이지 않는 구현방식이 필요함.

블록킹:

넌블록킹

**<Stress Test>**

동시 접속 테스트를 하면 적은 인원 테스트시 보이지 않던 버그들(특히 멀티쓰레드) 발견 가능하고 서버 프로그램의 병목 현상을 발견할 수 있음

CPU 병목 : 프로파일러 사용해서 어느 함수에서 cpu 타임 잡아먹었나 확인

네트워크 병목 : 네트워크 버퍼에 데이터가 쌓이는 양을 보고 확인

데베 병목 : 데베 리퀘스트 큐가 얼마나 쌓여있나 확인

메모리 할당 병목 : 말록이나 new 호출 잦을 때 발생, 서버 컴터 메모리가 작은경우

<자동 테스프 프로그램 작성시 고려해야 할 사항들>

개발 중인 게임이라면 캐릭터 생성도 자동으로 하게 해놓으면 편함

지형 정도는 인식하게 해놔야 실제 플레이랑 차이가 발생 안함

시장에만 다 모이게 하지말고 전체 월드를 관리자 레벨의 ID를 부여하여 텔포하게 만듦

서버 프로그램이 업그레이드될 때마다 같이 업그레이드 해줘야함

왜냐면 나중에 오픈 서비스 이후에도 이 테스트 프로그램을 계속해서 사용함

다중 접속이므로IOCP로 구현해야함, 클라인데 내부구조는 사실상 서버임, 그러나 서버랑 다른점은 connect로 접속함, 즉 IOCP를 이용한 connect임

접속한 아이디의 상태가 있어야함 (File State Machine), 그리고 그 상태에 맞춰서 패킷이 정해지고 액션이 정해짐, 이동-전투 정도만 받고 나머지 패킷은 무시함

간단한 그래픽으로 화면상에 전체 테스트 캐릭터들의 분포와 상태를 볼 수 있게 해야함

렉은 클라에서 보낸 신호가 서버에 갔다가 클라에 되돌려주는 시간을 측정하면 된다

Move packet에 전송 시간을 추가해서 측정한다, 그리고 time stamp를 추가해서 이건 언제 보낸 move 패킷이다 라고 기록을 해야 비교가 가능함, 이런식으로 측정한다.

그리고 서버랑 테스트 프로그램은 꼭 다른 컴퓨터로 돌릴 것

**<시야처리>**

브로드캐스팅 : 같은 내용은 여러 객체에게 동시에 보내는 것

근처의 오브젝트만 보여줌으로써 패킷의 개수를 줄여 성능을 개선하는 것

그러나 접속해 있는 모든 플레이어들을 루프를 돌며 거리를 재는 행위도 오버헤드 발생

보통 차원이 다르거나, 바다로 인해 대륙이 나누어져 있는 경우 Zone으로 나눔

그 zone안에서 또 sector, cluster로 나눈다, 서버의 검색 효율을 위한거라 클라랑 상관X

섹터의 크기가 너무 크면 시야 범위 밖의 개체가 많이 검색되서 여러 스레드가 한 섹터를 공유하므로 병렬성이 떨어지고 섹터의 크기가 너무 작으면 섹터 이동할때마다 락을 걸고 이동해야되는게 섹터 변경이 잦아져서 오버헤드가 늘어남

주로 기획 단계에서 시야범위가 정해지면 그거의 2배 크기 정도로 섹터를 나눈다

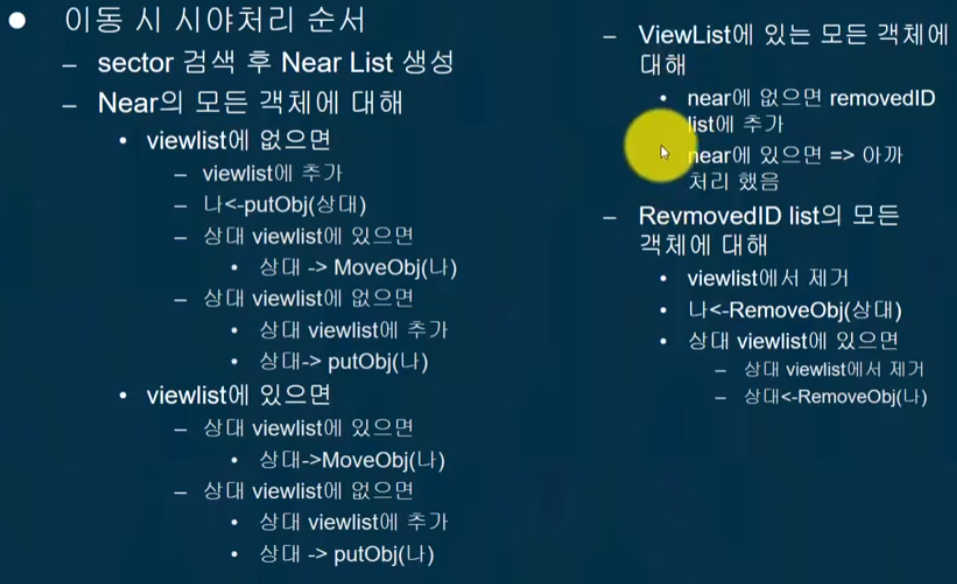
이제 모든 플레이어가 ViewList라는 객체로 다른 클라들을 관리해서 시야 범위에 들어오는 패킷을 먼저 받아주고 그 다음 move 패킷을 이용해 적은 과부화로 돌린다

ViewList 안쓰면 혼자일땐 상관없는데 2명이 동시에 움직일 때 한명 기준에서는 움직이려고 한 그 순간에 상대가 본인 시야범위 안이지만 실제로 둘이 동시에 움직여서 서로 시야범위 밖으로 나갔는데 나간걸로 인식이 안되는 경우가 발생 가능하다. 즉, 시야범위update가 아토믹하게 일어나지지 않고 경계에 애매하게 시야범위 안에 들어왔을 경우 enter 패킷을 못해줄 수 있음, 근데 ViewList를 쓰면 move 패킷을 보냈는데 viewlist안에 없다는게 확인되면 뒤늦게라도 enter 패킷을 보내줄 수 있는 확인장치가 됨

기존의 시야 리스트와 이동한 후의 시야 리스트를 만들어서 둘을 비교해 사라진 오브젝트는 leave처리 해주고 새로 생긴 오브젝트는 enter, 기존에 있는 오브젝트이면 move처리 해주면 된다.

이동시 상대방의 view\_list를 조작하지 않는 대신 상대방에게 접근/이동/이탈을 메시지로 보내는 방식을 사용하면 뷰리스트에 mutex를 쓸 필요가 없어진다.

Non blocking concurrent queue가 필요하다, 그리고 현재 new viewlist 만들 때 모든 클라이언트를 검색하게 되어있는데 스스로 섹터 분할을 구현하면 뷰 리스트를 최적화 할 수 있을거임.



removedID 따로 파는 이유가 viewlist도 멀티쓰레드에서 돌아가고 내 viewlist에 상대가, 상대 viewlist에 내가 접근할 수 있으니까 락을 걸어야하는데 내꺼에 상대를 지우기위해 내 viewlist에 락을 걸었는데 상대도 자신의 viewlist에 나를 지우려고 락을 걸어버리면 이중락 걸려서 데드락이 되어버린다.

viewlist내부를 돌려면 통으로 락을 걸어야하는데 객체 개수가 많아지면 락 걸려있는 시간 때문에 성능저하 일어나므로 viewlist를 락을 건 상태에서 복사를 하고 복사본을 통해서 비교한다

그리고 viewlist는 대부분 write보단 read하는 경우가 대부분이다, 그럴 경우엔 뮤텍스 락 보단 RWLOCK사용, 락을 걸 때 R모드-W모드 락을 걸 수 있는데 R모드로 락을 걸면 R모드로 락이 걸린 쓰레드끼린 동시에 실행 가능하고 W끼리는 상호 배제한다.

**<npc 및 타이머>**

Npc가 움직일때마다 패킷을 보내면 네트워크에 과부화가 온다, 그래서 특정 시간차를 두고 그때마다 패킷을 보내야함, 그게 바로 step 및 타이머.

NPC AI는 타이머 기반의 유한 상태 기계

모든 NPC를 스크립트로 돌리면 삽질이고 기본 AI는 하드 코딩, 보스몬스터 같은 애들을 스크립트로 구현하는게 효율적임, 그리고 겜서버는 무조건 Worker Thread 외에 다른 쓰레드가 있어선 안된다, 우리가 코어 수에 맞게 Worker Thread를 설정해놨는데 다른 쓰레드가 추가되면 안됨, 타이머 쓰레드든 가상 머신용 쓰레드든 전부 Worker Thread 내부에서 해결하도록 해야함

실습에선 객체 하나당 하나의 가상 머신을 둬서 worker thread가 가상 머신을 돌릴거임, 그럼 가상 머신 수가 엄청 많을거고 (클라가 1000개면 1000개) 대부분의 가상머신이 대기상태라 메모리 낭비가 있을거임, 그럼 그때그때 가상 머신을 할당해서 호출하고 초기화하는게 오히려 오버헤드 더 클수도 있음, 그래서 전용 가상 머신을 객체에 넣어서 돌리는게 심플하고 오버헤드가 더 적음.