**<11.2> 16기 수업에서 추가.**

**NO91)**

**1:00:00 IOCP\_TEST끝나고 조상현대표의 GCGC자료와 멀티스레드 설명부분 누락.**

**<11.4> 16기 수업에서 추가.(캐시관련)**

**<7.27>**

덤프 파일은 (xxx.dmp) 형식으로 파일이 남게된다.

윈도우가 오류가 났을 경우(블루 스크린 등) 덤프를 빼서 MS에게 메일을 보내는 경우를 본적이 있을 것이다.

**서버에서 덤프남기기**

**서버쪽에서는 풀덤프를 빼서 프로세스의 모든 메모리를 저장한다.**

(게임서버가 2GB일경우 덤프도 2GB)

**파일의 크기가 매우 클것이지만 저장만 할것이므로 문제되지않는다.**

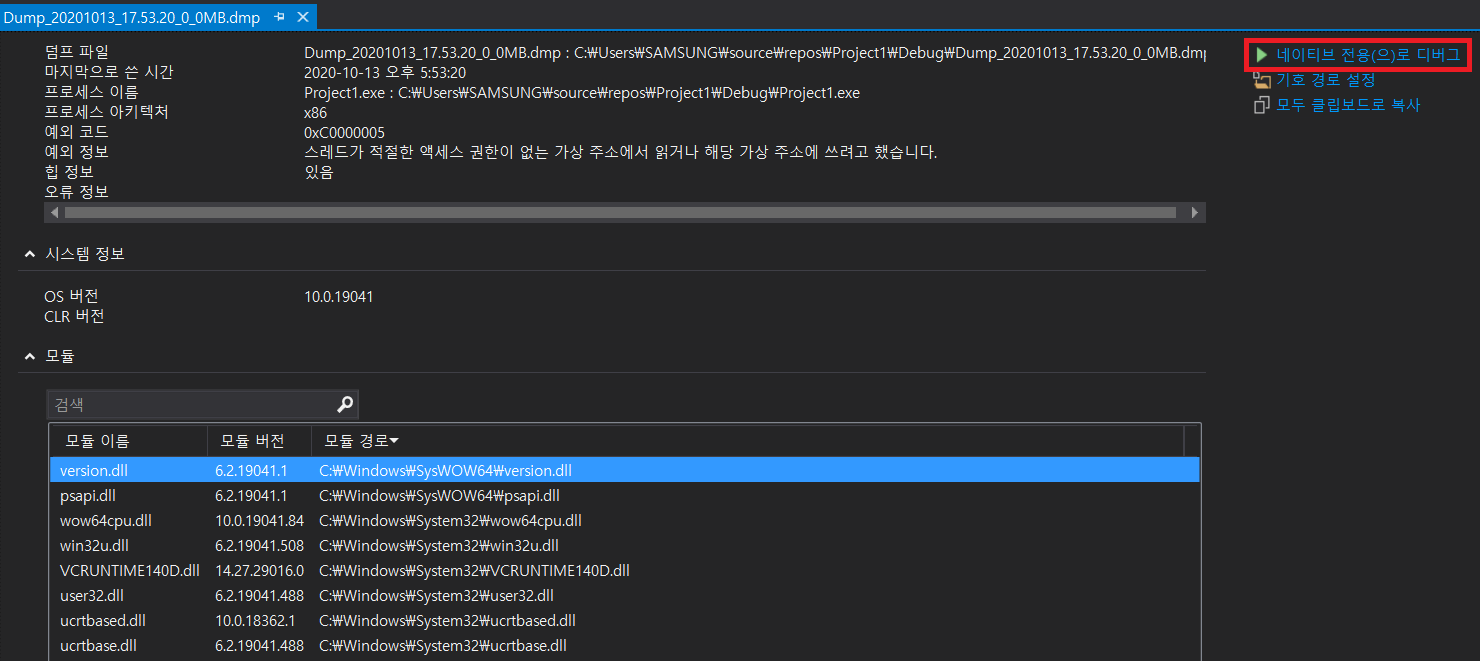
**클라쪽은 풀덤프의 파일크기가 부담되기 때문에, 스택메모리와 스택상태만 저장하기도 한다.**

(몇KB정도)

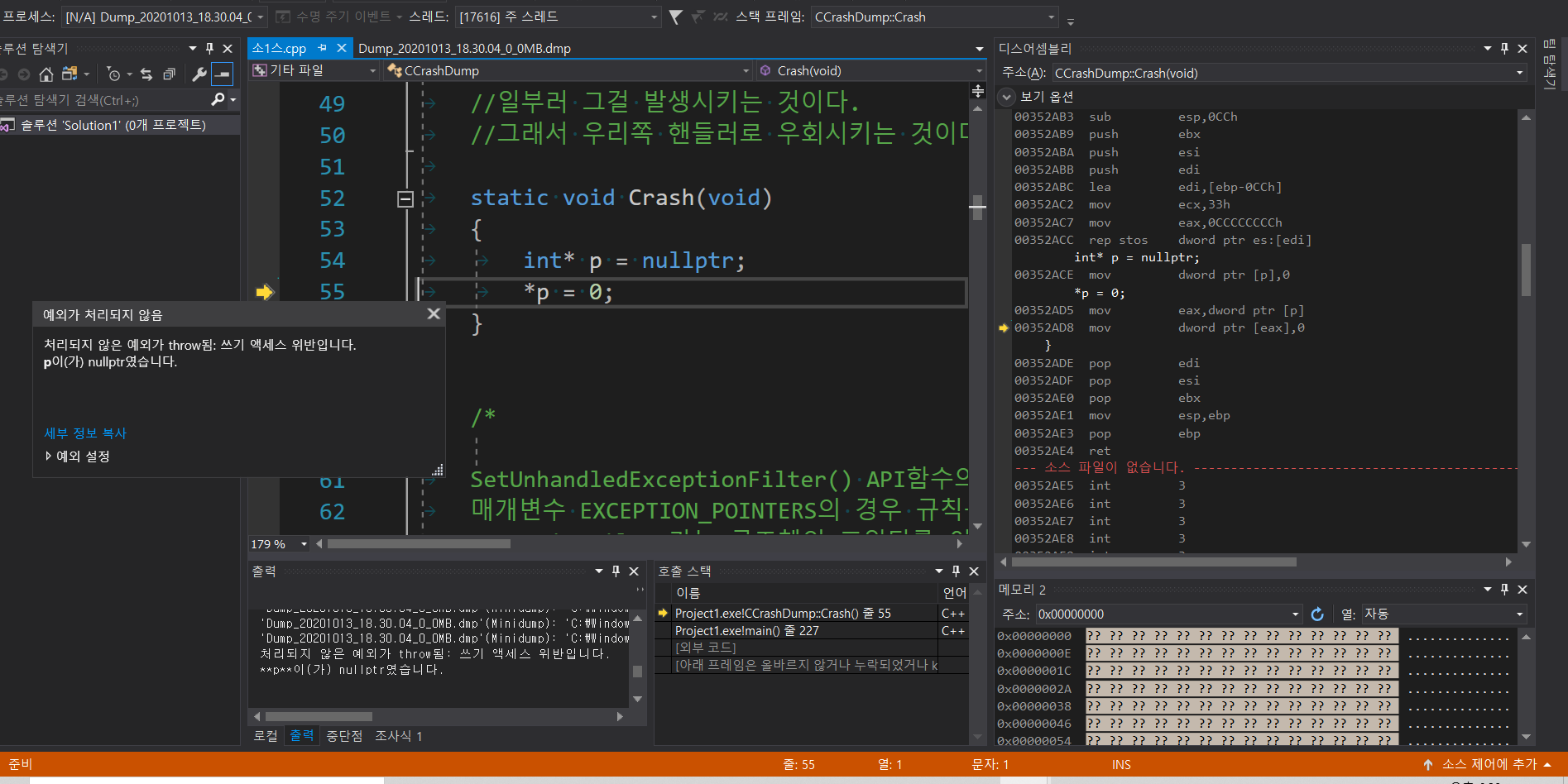
스택만 뺀경우에는 전역메모리 등을 확인할수 없기 때문에 상황파악이 더 힘들어질 것이다.

덤프파일은 실행시 오픈된다.

WINAPI에서 메모리덤프를 제공하고, 이를 통해 파일로 저장한다.



사용한 Dll을 확인할수있음.



**플레이 버튼을 눌러 디버깅한다면, 우리가 F5를 사용하여 디버깅하는것처럼 디버깅이 가능해질것.** (호출스택, 조사식, 메모리, 스레드 등 확인이 가능한 상태)

서버에서 오류가 날시 모두 이방식으로 확인을 하게 될 것이다.

빌드시에 exe파일과 pdb파일이 나오고, 실제 exe파일 실행하다가 크래시가 나면 덤프파일이 나온다. 이때 덤프파일은 빌드당시의 pdb파일이 있어야 맵핑되어 소스코드가 나온다.

**멀티스레드 환경에서의 디버깅**

멀티스레드에서 동기화문제는 더 이상 브레이크포인터로 정확히 확인할 수 없다.

**오면 안되는 상황을 캐치하여 덤프를 저장하고, 이를 기준으로 디버깅해야한다.**

**단, 락프리쪽으로 넘어가면 이미 덤프를 저장하려는 순간에도 다른스레드는 멈추지않기 때문에, 저장된 데이터가 새로 바뀌는등 상황이 바뀌게 된다.**

(\*pObject라는 객체에 대해 동시에 다수의 스레드가 접근할 때, 해당객체가 NULL인 경우 Crash를 냈다. 이때 덤프를 남기려는 순간 다른스레드에서 값을 수정해버릴 수도 있음.)

**따라서 멀티스레드 환경에서는 덤프만을 100%믿을 수 없는 상황이고, 이에대한 해결책은 우리가 직접 찾아야 한다.**

**Dump(덤프)를 남기기 위한 조건**

덤프를 남기기위해서는

**1. 이 덤프를 남길 때 실행한 실행파일(exe)**

**2. 해당 실행파일을 빌드할 때 있던 pdb파일이 필요하다**.

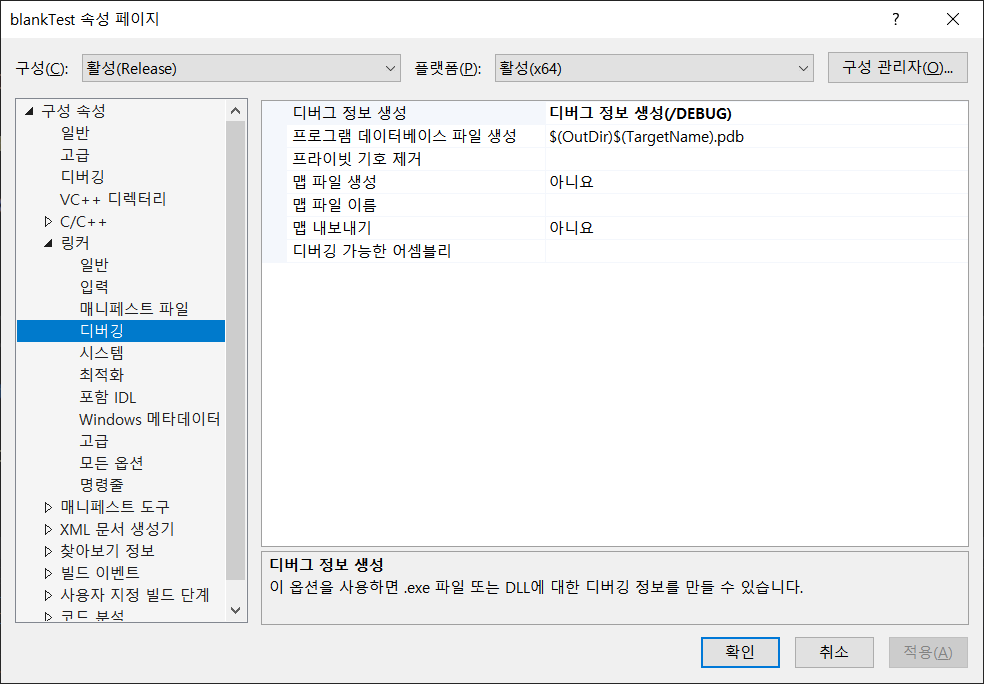
((

**Pdb 파일**

**pdb파일은 우리쪽 소스코드와 기계어를 맵핑시켜주는 역할로, 무조건 보존되어야 한다.**

pdb파일이 없이 실행파일(exe)만 가지고서 디버깅을 한다면, 브레이크 포인터로 디버깅하는것 역시 불가능하고 [프로세스에 연결] 로 메모리/어셈블리 확인만이 가능하다.

([링커 -> 디버깅] 에서 설정가능)



빌드할때마다 pdb파일이 생성되기 때문에 이를가지고디버깅 할 수 있는 것.

**실행파일(exe)과 pdb파일**

**pdb에는 빌드할때마다 타임값으로 소스코드 버전이 관리되고 있기 때문에,**

**해당 실행파일을 만들때 생성된 pdb파일만이 맵핑이 가능하다.**

**따라서 실행파일과 pdb파일을 세트로 보관해야 한다.**

**(새로 빌드한 pdb파일과 실행파일(exe)는 맵핑되지않는다.)**

저 같은 경우도 서버에다 실행파일과 같이 pdb를 올린다.

**pdb파일이 없거나 버전이 다른경우**

VisualStudio에서는 버전에 맞는 pdb프로그램 데이터베이스 파일을 찾을수 없다고 에러가 난다.

**차선책으로는 WinDebug같은 프로그램을 사용하여 강제로 pdb를 올려 셋팅을 해야한다.**

이는 메모리 위치를 맞추는 등 굉장히 복잡한 작업을 요하고, VisualStudio에 비해 보기도 어렵다.

(저도 프린트물 들고 따라하는 수준)

**따라서 실행파일 빌드 시 같이 남는 pdb파일을 반드시 보존시킬 것.**

**오류 예시**

아래와 같은 구문에서 오류가 났다고 가정하자.

pObject->Action(0);

(만약 pObject객체가 존재하지않는다면 this를 호출하는 내부에서 오류가 나게된다. 따라서 위문제는 객체문제X)

**위와 같은 경우 100% Action이 가상함수이며, 우리는 최우선적으로 내부 가상함수 테이블이 왜 손상되었는지 원인파악을 해야한다.**

**Dump코드**

덤프를 남기는 행위는 WINAPI에 있는 MiniDumpWriteDump()를 호출시켜 진행한다.

Dump코드는 사실상 다 똑같기 때문에 이해하여 사용하면 된다.

#pragma comment(lib, "DbgHelp") //MiniDumpWriteDump

#include <stdio.h>

#include <windows.h>

#include <Psapi.h> //PROCESS\_MEMORY\_COUNTERS

#include <DbgHelp.h> //\_MINIDUMP\_EXCEPTION\_INFORMATION

#include <crtdbg.h> //\_CrtSetReportMode

class CCrashDump

{

public:

CCrashDump()

{

\_DumpCount = 0;

\_invalid\_parameter\_handler oldHandler;

oldHandler = \_set\_invalid\_parameter\_handler(myInvalidParameterHandler);

//myInvalidParameterHandler() 정의부분 참조.

//밑 구문 역시 CRT오류 메시지표시를 중단하는 것.

//우리는 바로 덤프를 빼고 프로세스를 종료시킬 것이다.

\_CrtSetReportMode(\_CRT\_WARN, 0);

\_CrtSetReportMode(\_CRT\_ASSERT, 0);

\_CrtSetReportMode(\_CRT\_ERROR, 0);

\_CrtSetReportHook(\_custom\_Report\_hook);

//위 네줄은 경험한 적은 없으나 만약을 위함

//pure virtual function called 에러핸들러를 사용자 정의 함수로 우회시킨다.

//별도 참조. (밑에 언급)

\_set\_purecall\_handler(MyPurecallHandler);

SetHandlerDump();

}

static void Crash(void)

{

int\* p = nullptr;

\*p = 0;

}

/\*

우리가 정의한 MyExceptionFilter는 SetHandlerDump()함수 내부에서,

API함수인 SetUnhandledExceptionFilter()의 매개변수로 전달될 함수.

이 함수의 매개변수 EXCEPTION\_POINTERS은 규칙을 따른것.

ExceptionFilter라는 구조체의 포인터를 여기서 받으면 된다.

그럼 예외발생시 이 인자로 자동으로 들어올 것이다.

우리는 여기서 덤프를 여기서 뺀다.

\*/

static LONG WINAPI MyExceptionFilter(\_\_in PEXCEPTION\_POINTERS pExceptionPointer)

{

SYSTEMTIME NowTime;

long DumpCount = InterlockedIncrement(&\_DumpCount);

//----------------------------------------------------------

//현재 날짜와 시간을 알아온다.

//----------------------------------------------------------

WCHAR filename[MAX\_PATH];

GetLocalTime(&NowTime);

wsprintf(filename, L"Dump\_%d%02d%02d\_%02d.%02d.%02d\_%d.dmp",

NowTime.wYear, NowTime.wMonth, NowTime.wDay, NowTime.wHour, NowTime.wMinute, NowTime.wSecond, DumpCount);

wprintf(L"\n\n\n!!! Crash Error!!! %d.%d.%d/%d:%d:%d\n", NowTime.wYear, NowTime.wMonth, NowTime.wDay, NowTime.wHour, NowTime.wMinute, NowTime.wSecond);

wprintf(L"Now Save dump file...\n");

HANDLE hDumpFile = CreateFile

(

filename,

GENERIC\_WRITE,

FILE\_SHARE\_WRITE, //쓰기모드

NULL,

CREATE\_ALWAYS,

FILE\_ATTRIBUTE\_NORMAL, NULL

);

if (hDumpFile != INVALID\_HANDLE\_VALUE)

{

//덤프파일 정보 설정

\_MINIDUMP\_EXCEPTION\_INFORMATION MinidumpExceptionInformation;

MinidumpExceptionInformation.ThreadId = ::GetCurrentThreadId();

MinidumpExceptionInformation.ExceptionPointers = pExceptionPointer; //인자로 들어온 예외포인터

MinidumpExceptionInformation.ClientPointers = FALSE;

//msdn상에서는 외부디버거에서 해당 덤프를 뺄때 설정이라고 명시되어있음.

//TRUE/FALSE에 대한 차이점을 찾지못함.

/\*

MiniDumpWriteDump는 실질적으로 메인이 되는 함수.

이함수를 호출시키면 전달된 파일(핸들)을 대상으로 write가 시작된다.

\*/

MiniDumpWriteDump

(

GetCurrentProcess(),

GetCurrentProcessId(),

hDumpFile, //File 핸들.

MiniDumpWithFullMemory, //\_MINIDUMP\_TYPE. 풀덤프로 지정.

&MinidumpExceptionInformation, //우리가 정의한 미니덤프의 예외정보.

NULL,

NULL

);

CloseHandle(hDumpFile);

wprintf(L"CrashDumpSaveFinish!");

}

return EXCEPTION\_EXECUTE\_HANDLER;

//파일에 덤프를 모두 저장 후 리턴.

//해당값을 리턴해 예외처리가 끝났다고 알려, 예외창이 뜨는것을 막는다.

}

//클래스로 래핑하여 생성자쪽에서 호출된다.

//SetUnhandled..에 우리함수를 등록시킨다.

static void SetHandlerDump()

{

SetUnhandledExceptionFilter(MyExceptionFilter);

/\*

윈도우 API함수.

핸들링되지않은 모든 예외를 여기로 받겠다는 의미.

(Throw를 던졌으나 catch를 받지않은경우,

메모리참조 오류가 발생했거나 모든 예외를 받는경우 등.)

만약 catch를 못받을 경우 메인까지 튀어나와서 SHE로 예외를 발생시킨다.

매개변수로 우리가 정의한 함수를지정하면,

예외가 발생됐을때 이 함수가 호출된다.

\*/

}

/\*

CRL함수에 인자를 잘못넣는 경우 INVALID\_PARAMETER 발생.

(매개변수 가변인자 오류, 0이 들어갈수 없는곳에 0 전달 등..)

예외상황은 우리가 캐치하여 덤프를 빼야한다.

CRL에서 띄워주는 예외창을 막아야 한다.

(생성자참조.) \*/

static void myInvalidParameterHandler(const wchar\_t\* expression, const wchar\_t\* function, const wchar\_t\* file, unsigned int line, uintptr\_t pReserved)

{

Crash();

}

static int \_custom\_Report\_hook(int ireposttype, char\* message, int\* returnval)

{

Crash();

return true;

}

static void MyPurecallHandler(void)

{

Crash();

}

static long \_DumpCount;

};

long CCrashDump::\_DumpCount;

CCrashDump CrashDump;

**멀티스레드 염두**

위 코드는 멀티스레드라는 전제가 되어있지 않다.

**WorkerThread처럼 하나의 같은 코드를 여러스레드가 같이 사용한다면 한쪽에서 문제가 터졌을 때 연타로 터질 수 있다.**

(ex SessionMap에 예외가 발생한다면, 다른스레드에서도 연달아 예외 발생할 수 있음)

**이 경우 CreateFile이 연속으로 호출되고, Write를 오픈하다 실패하게 된다.**

따라서 카운트를 붙여 덤프남길때마다 서로 다른 덤프가 남게끔 파일을 생성한다.

따라서 멀티스레드에서 중복으로 돌던 코드가 문제가 터지는 경우 덤프가 여러 개 남을 것이고, 그중에 하나라도 살리면 된다.

(첫번째 파일을 잡는게 가장 좋을 것이나 안되는경우 뒤에꺼라도 잡아야 할 것)

**덤프파일이 남지 않는 경우**

**UnhandledExceptionFilter을 호출한다고 하여 덤프파일은 100%보장되지 않고, 남지않는 경우에도 뾰족한 방법이없다.**

덤프가 남지않는 원인에 대해서는 여러가지가 있다. 대표적으로는,

**1. 우리쪽으로 핸들링이 되지않은 경우**

**2. 핸들링이 되었는데 덤프를 남기는 과정에서 실패하는 경우**

(덤프를 남기는데 실패하는 경우 용량이 0이거나, 깨진상태거나, 아예 남지않는 등 다양한 결과가 나오게 된다.)

**우리가 임의로 NULL포인터를 찔러 메모리 참조 오류를 내는경우는 거의 100% 보장되므로, 이를 주력으로 간다.**

**1. 우리쪽으로 핸들링 되지않은 경우**

**이는 SetUnHandledExceptionFilter()이 다른쪽에서 호출되어 가로채간 경우이다.**

(생각보다 많이 발생한다)

윈도우 2013까지는 CRL 내부에서 **시큐리티 쿠키**(함수 리턴시 스택에 대한 오버플로우, 언더플로우를 체크해주는 기능. 컴파일 버전마다 다르다.)가 항상 들어갔다.

**시큐리티 쿠키 내부에서 예외가 발생하여 SetUnhandleExceptionFilter를 호출시키는 경우, 이쪽 런타임 라이브러리 내부로 핸들링이 전환된다.**

이 경우 덤프를 빼지못하게 될 것.

((

**WINAPI를 후킹하여 가로챈 핸들링을 다시 가져오는 방법**

이를 우회하는 방법으로 SetUnhandledExceptionFiler라는 API자체를 후킹(함수호출, 메시지, 이벤트 등을 중간에서 바꾸거나 가로채는 방법)이 존재한다.

이 방법은 내 프로세스 메모리에 함수테이블을 모두 조회해서 DLL커널내부에 존재하는SetUnhandleExceptionFilter에 맵핑된 함수 포인터 주소를 내함수로 바꿔치기하는 방식이다.

(이는 굉장히 복잡한 방식으로, 후킹이란 것은 API관점에서는 좋지 않은 행위이기 때문에, window8 이후로부터 구조가 바뀌면서 작동이 안됨…)

이에 대한 언급은 via에서도 소스코드를 제공하고 이를 참조하라는 정도로만 넘어간다.

))

**2. 핸들링 되었으나 덤프를 남기는 과정에서 실패한 경우.**

(Heap을 따로 만드는 이유와도 일맥상통한다)

단순히 동적할당받은 메모리에 대한 오버플로우라면 해제할 때 예외가 발생된다.

**하지만 힙을 심각하게 깨뜨리는 경우, 다음번 malloc호출 시 내부 Heapalloc할 때 예외가 발생된다.**

**이때 MyExceptionfilter가 호출되어 CreateFile이 호출될텐데, CreateFile은 힙메모리를 사용하므로 호출이 실패되어 덤프를 남기지 못하게된다.**

**외부에서 덤프를 남기는 방법**

내부에서 남기는 덤프가 위와 같은 이유로 남지않는 경우를 대비하여 외부에서도 덤프를 남긴다.

(내부, 외부에 모두남길 것)

이는 외부 덤프를 빼는 프로그램을 이용한다.

(윈도우XP까지는 윈도우 차원에서 닥터왓슨이란 것이 제공되었지만, 64비트용이 나오지않았기 때문에 사용할 수 없다.)

현재 VisualStudio를 설치한 환경에서는 Just In Time Debug를 사용하도록 셋팅되어, 어떤 예외가 발생하는 경우 VisualStudio로 디버그 하게끔 한다.

**지금 상황은 외부에서 덤프를 뺄 방도를 찾고있으므로,**

**MS의 WindowSysInternel에서 제공하는 프로세스 덤프라는 프로그램을 사용한다.**

이에 대해서는 생략한다.(이후 언급)

서버는 항시 문제가 터지게 되므로 실제 라이브 서비스에서도 덤프파일이 많이 남아있는 것을 확인할 수 있음. (해결하면 지우는식..)

**pure virtual function called**

//pure virtual function called 에러핸들러를 사용자 정의 함수로 우회시킨다.

\_set\_purecall\_handler(MyPurecallHandler);

자식클래스가 부모클래스의 순수가상함수를 정의하지않는 경우에는 컴파일 에러를 내준다.

**단, 포인터로 정의하여 접근하는 경우 컴파일러 입장에서는 알 수 없으므로 문법허용이 되며**

**이때 포인터로 접근한 객체내부의 this에 접근이 될경우 이때 pure virtual function called에러가 발생하게 된다.**

(나중에는 OnRecv, OnAccept, OnLeave등을 모두 가상함수로 잡아 상속받아 가져간다. 이때 실수하여 purecall에러가 나게된다.)

**C++ 라이브러리 예외 핸들러**

\_DumpCount = 0;

\_invalid\_parameter\_handler oldHandler;

oldHandler = \_set\_invalid\_parameter\_handler(myInvalidParameterHandler);

//myInvalidParameterHandler() 정의부분 참조.

//밑 구문 역시 CRT오류 메시지표시를 중단하는 것.

//우리는 바로 덤프를 빼고 프로세스를 종료시킬 것이다.

\_CrtSetReportMode(\_CRT\_WARN, 0);

\_CrtSetReportMode(\_CRT\_ASSERT, 0);

\_CrtSetReportMode(\_CRT\_ERROR, 0);

\_CrtSetReportHook(\_custom\_Report\_hook);

//위 네줄은 경험한 적은 없으나 만약을 위함

C++ 내부에서 문제가 발생하는 경우 에러창이 뜰것이다.

**이는 서버가 돌아가는 와중에 뜨면 안되므로 우리쪽 함수인 myInvalidParameterHandler로 우회시켜, 내부에서 NULL포인터를 찌르게끔 한다.**

myInvalidParameterHandler(), \_custom\_Report\_hook(), MyPurecallHandler(), 모두 내부에서 Crash()를 호출한다.

**static으로 정의한 이유**

우리가 클래스로 묶은 CCrashDump 클래스의 경우 모든함수가 스태틱으로 정의되어 있다.

이는 어떤 곳에서든 서버를 즉시 종료하고 덤프를 남겨야 하는 상황에 Crash()를 호출하기 위함이다.

Static을 선언하는 경우 소속이 생기므로, (소속있는 전역변수, 또는 함수) 다른 파일에서 같은이름이 있다고 하더라도 구분된다.

**우리가 만드는 함수들은 함수포인터로 내부에서 등록이 되므로, 일반클래스 멤버가 아닌 static으로 가야한다.**

클래스가아닌 전역함수 덩어리로 존재해도 문제없음.

**Crash();**

**이제부터는 와서는 안될곳에 Crash()를 박아넣고 브레이크 포인터와 같은 역할로 써먹을 것이다. 단순히 NULL포인터에 값을 찔러도 되지만, 유지보수를 위해 Crash()로 통일한다.**

실제 라이브 서비스 코드에서도 Crash()코드는 2~30군데가 넘게 들어갈 것이다.

**위쪽 가상메모리 테이블의 상단부는 접근안되는 구역으로 잡아놨다. (????)**

**MyExceptionFilter()**

덤프클래스 안쪽의 우리의 MyExceptionFilter() 함수 내부는 디버깅 되지 않을 것이다.

VisualStudio가 디버거가 됐기 때문에, UnhandledException필터를 VisualStudio가 하게된다. 따라서 이 안쪽은 브래이크를 걸수 없음.

**프로세스의 메모리사용량을 얻는방법**

현재 프로세스에 대한 메모리를 판단해서 해당 메모리 크기를 덤프파일 이름에다 남기도록 한다.

덤프파일의 용량이 곧 메모리크기 이기 때문에, 없어도 상관없음.

//----------------------------------------------------------

//현재 프로세스의 메모리 사용량을 얻어온다.

//----------------------------------------------------------

HANDLE hProcess = 0;

PROCESS\_MEMORY\_COUNTERS pmc;

hProcess = GetCurrentProcess();

if (hProcess == NULL)

return 0;

if (GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc)))

{

WorkingMemory = (int)(pmc.WorkingSetSize / 1024 / 1024);

//이부분. 생략해도 상관없음. 불필요한 코드가 될수있음.

}

CloseHandle(hProcess);

**<7.29>**

**=========================================================**

**(로그부분 추가분)**

**----------------------------------------------------------**

**DB저장**

데이터 DB저장은 행동 할때마다 비동기처리되는 별도 스레드에 큐잉하면서 들어간다.

로그는 별도의 로그 DB에 저장한다.

**게임로그**

게임로그는 클래스화/라이브러리화 시키지않는다.

게임로그는 컨텐츠와 밀접하게 들어가는 부분이기 때문에 컨텐츠를 만들때 들어간다.

**게임로그로 해당 유저가 게임내에서 어떤 행위를 했는지 CCTV를 설치한것처럼 모든것을 알수있어야 한다.**

(단순한 좌표이동의 경우는 기록하지않지만, 마을에서 마을 이동정도의 큰 이동이라면 이 역시 기록)

**게임서버에서 버그가 난 상황**

컴파일러가 알려주는 버그가 난 상황인 경우

**1.코드확인**

**2.텍스트로그 확인**

**3.게임로그 확인**

로그를 확인하여, 해당유저의 행위를 똑같이 따라 해본다.

**재현되지 않는 경우 라이브서비스의 DB를 그대로 백업받아 해당 유저로 로그인을 해서 실제로 완벽하게 똑같이 따라해볼 것.**

게임로그를 볼수있게끔 웹프로그램으로 만들어 놓는 작업은 웹개발자가 할것이고, 서버프로그래머의 역할은 깔끔하게 저장을 해주는것.

**당연히 로그를 넣을수록 게임서버는 느려질 것이나, 게임로그는 반드시 남겨야 하는 데이터이므로 감안하고 가야한다.**

지금우리의 서버는 단순히 코드만 들고가지만 실제로는 연동될수있는 다른서버, DB연동 등을 포함시킨다면 성능이 떨어질수밖에 없다.

서버를 아무리 잘 만든다 한들 DB가 받쳐주지못한다면 (스케일업을 했음에도) 동접을 떨어뜨려야 하는것.

**stateful서버의 데이터저장**

**stateful에서의 데이터 저장은 1차적으로 메모리 변화과 있고, 그다음 DB에 저장된다.**

따라서 롤백(데이터가 DB에 저장되지않아 그전 데이터로 돌아가는 현상)이란 개념이 존재한다.

**로직상에서는 거래를 했다고 해도 로그는 해당시점에서 남기는것이 아니며, 해당내역을 DB에 저장할때 남긴다.**

**이때 로그와 DB반영 중 둘중 하나만 남는다는것은 있어서는 안되는 일이다.**

**하지만 트랜잭션을 사용하지않는 이상 어느것 하나가 누락될수 있는 가능성이 분명히 존재한다.**

(트랜잭션은 commit/롤백중 하나는 반드시 보장되게끔 했다.)

**1. 게임내에서 레벨이 오른 것 (레벨 변경)**

**2. 레벨이 오른 실제 컨텐츠 플레이어를 업데이트 (로직상 변경)**

**3. 유저가 레벨이 올랐다는것을 삽입 (DB저장)**

위 목록을 같은 DB에 넣고 트렌젝션으로 묶어서 해결한다면 완벽하게 끝난다.

**1. 트렌잭션을 사용하지 않는 경우**

**단점**

**DB저장과 로그중 하나가 유실될 수 있는 상황**

위와같은 경우 유저가 직접 게임사에 전화를해서 해결을 봐야한다.

**이유**

**성능, 게임로그의 통합(회사의 모든 게임을 로그 시스템을 남기도록 목표를 하는경우) 등의 문제로 인하여 트렌젝션을 포기하고 로직과 DB저장이 분리가 되는경우,**

완벽하게 잡을 수없는 상황이므로 최선을 다해 저장하고 끝낸다.

**분명히 유실될수 있는 상황이지만, 이는 하드웨어적인 문제이며 m/s단위보다 작은 찰나의 순간이므로 거의 가능성이 없다.**

**단점 해결방안**

무엇을 먼저 남길지는 선택사항이며, 유실된 것에 대한 해결또한 생각해야 할것.

(로그먼저? 컨텐츠 반영먼저?)

**2. 트렌잭션을 사용하는 경우**

**단점**

이 경우 서버마다 로그가 나오게 된다는 단점이 있다.

따라서 관리가 매우 힘들어 진다.

**이유**

**유실걱정 없이, 위 목록을 같은 DB에 넣고 트랜잭션으로 묶어서 해결한다면 완벽하게 끝난다.**

**트랜잭션 구현**

트랜잭션도 DB라는 시스템도 C로 구현되어있기 때문에 우리가 직접 구현하는것이 가능하다.

그래서 소프트웨어 트렌잭션으로 직접 구현하는 케이스도 존재한다.

이는 차후의 방법으로, 처음부터 이방법을 사용하는것은 권장하지 않는다.

**게임로그 특징**

**- 게임로그는 무조건 DB 에 저장한다. (파일저장은 의미X)**

**- 게임 데이터가 저장,변경되는 순간에 함께 기록. (DB의 변화)**

**- DB 저장 트랜젝션과 함께 처리.**

**- 트랜젝션을 하지않아도 DB 쿼리와 함께 로그쿼리 전송**

**- 게임 데이터가 변경 로그가 남아야 한다.**

**게임로그 저장**

**게임로그는 DB에 저장되는 시점에 함께저장된다.**

**서버상에서 로직의 처리에 따라 게임로그가 저장되는 것이 아니다.**

**따라서 서버다운으로 유저들의 데이터가 롤백되었다면, 당연히 게임로그도 없을 것이다.**

우리는 stateful서버이기 때문이다.

(stateful/stateless는 막힘없이 설명이 가능해야 할 것)

- 게임로그는 일반적으로 다음과 같은 형태를 가질 수 있음.

Server / Type / Code / AccountNo / Param1 / Param2 / Param3 / Param4 / ParamStr ...

모든 로그에 대해서 위와 같은 포멧을 사용한다.

모든 컬럼은 숫자 타입을 사용하는게 좋다.

숫자만으로 표현이 안되는 경우는 마지막 컬럼을 스트링으로 하여 사용.

Code : 돈획득 / Param1 : 1000원 (X)

Code : 몬스터 잡아서 돈 획득 / Param1 : 잡은 몬스터 / Param2: 1000원 획득 / Param3: 총 231000원 보유

어떤행위를 통해 무엇을했고, 무엇을 얻었고, 결과는 어떻게 되었는지 구체적으로 기록해야한다.

로그만 보더라도 이 유저가 무엇을 했는지, 무엇을 얻었는지, 출처는 어디인지를 모두 직관적으로 확인이 가능해야한다.

**게임로그 테이블**

**게임로그 테이블은 분류별로 나누지 않고, 기간별로 나눈다.**

(많이 실수하는 부분)

게임로그는 최대한 컨텐츠 별로 상세하게 나누어 기록해야 한다.

로그를 남기는 작업은 완성단계에서 테스트가 들어간다.

**로그작업은 매우 중요한 작업이므로, 완성 후 모든로그가 정상적으로 출력되는지 반드시 확인해야 한다.**

(중요한 작업이므로 신입한테 시키지않음)

**게임로그 분산저장**

게임로그는 하루에 TB단위로 남을 때도 있을 정도로 용량이 매우 크다.

따라서 분산하여 저장하고 관리해야 한다.

**이때, 게임로그는 컨텐츠별로(ex Item/Character/Quest) 로그테이블을 나눠서는 안된다.**

게임로그를 남기는 이유는 유저의 행동을 추적하기 위함인데, 컨텐츠로 나누게되면 이를 추적하기 굉장히 어려워 진다.

그리고 컨텐츠 별로 나눈경우 차후 추가적으로 분산이 필요할때도 난감해진다.

**결국 관리나 검색이 힘들어지고 쿼리가 쌓였을 때 처리가 곤란해지기 때문에,**

**모든 게임로그는 하나의 테이블에 남기며 기간별로 분리한다. (일단위, 주단위, 월단위 등)**

(월단위 선호)

**log\_202002**

**log\_202003**

**log\_202004**

**위와같이 DB테이블을 기간별로 분류하고, 테이블 안에는 모든 게임로그를 저장한다.**

게임로그는 해당 게임과 DB 에 종속적인 느낌이 강하기 때문에 별도의 라이브러리 제작은 하지 않을것이며

실제 게임개발시 DB 저장 방식에 따라서 그때그때 제작하여 코드에 박을 것이다.

**그밖의 게임로그**

DB에 변화가 없음에도 불구하고 기획팀 / 사업팀에서 필요로 하는 경우 넣어줘야 한다.

어떤 버튼을 어떻게 몇번을 누르는지, 특정시점에 유저들의 행동패턴은 어떤지, 특정 지역에 몇명정도가 모이는지 등..

위 사항중 몇몇가지는 클라쪽에서 코드를 심고 외부로 빼는것도 가능하다.

(게임로그와는 별도)

**시스템로그 (파일에 저장하는 로그)**

**시스템로그는 개발자를 위한 로그로 개발과정에서 원하는 문자열 파일등으로 저장한다.**

**시스템로그는 전역함수 또는 싱글톤으로 구현한다.**

**개발 도중 모니터링 서버에는 없는 항목이고, 모니터링 서버에 추가하자니 복잡하므로 주기적으로 잠깐 보고싶은 경우 사용한다.**

(ex 직렬화버퍼/플레이/몬스터/아이템 등을 저장하는 메모리풀을 모니터링하는데 추가적으로 NPC객체에 대한 메모리사용량을 주기적으로 보고싶은 경우)

**- 로그분류 / 날짜,시간 / 문자열**

**- 파일저장시 - 분류별로 월별로 파일 나누어 지도록.**

**- DEBUG / WARNING / ERROR / SYSTEM(Notice:주의) 등 4단계로 로그 기록 설정.**

**- 코드 설정을 통해서 콘솔/파일.(지금은 둘다권장)**

**SYSLOG\_DIRECTORY(L"Log\_MatchmakingServer"); // 시스템로그를 저장 할 폴더 지정**

이때 서버가 여러 개인 경우 한폴더에 여러개의 게임서버 실행파일(게임서버/채팅서버 등..) 이 들어갈 것이다.

이 경우 시스템로그가 겹치고 관리가 힘들것이므로, 처음 메인코드에 넣어야한다.

(메인 함수 들어가면 로그폴더부터 설정하고 로그레벨을 지정한다.)

((

**모니터링 서버**

현재는 게임로그/시스템 로그로만 나눴으나, 상황에 따라 모니터링 서버를 만들수도 있다.

(모니터링 서버는 보통 회사차원에서 관리된다)

**모니터링 서버는 수집하는 모든데이터 (CPU/메모리 사용률, 동시접속자 등)를 로그로 누적하여 남길 것이다.**

**별도의 서버를 빼는것으로 새로운 항목을 넣고자 한다면 DB저장, 네트워크 연동 프로토콜 추가등 복잡한 과정이 필요하다.**

))

**시스템로그 성능이슈**

**시스템로그가 많이 남을수록 당연히 성능은 떨어진다.**

**시스템로그는 비정상적인 상황에서 남기 때문에, 성능에 대한 이슈는 크게 신경쓰지않는다.**

**시스템로그 코드**

**SYSLOG\_LEVEL(LOG::LEVEL\_ERROR); // 로그 레벨 지정**

**void Log(WCHAR \*szType, en\_LOG\_LEVEL LogLevel, WCHAR \*szStringFormat, ...);**

**Log("Battle", LOG::LEVEL\_DEBUG, "[DBAcc]:%u [Slot]:%d [GameAcc]:%d d [RandKey]:%d", a,b,c,d,e,....);**

(시스템 로그는 컨텐츠별로 분류해도 된다)

**시스템 로그레벨은 매크로가 아닌 런타임에서 판단하여, 서비스 도중에도 변경이 가능해야한다.**

# 모든 로그 저장은 설정레벨 (DEBUG, WARNING, ERROR) 과

# 호출시 로그 레벨을 비교하여 상위 레벨의 경우에만 저장을 한다.

DEBUG 모드 : DEBUG, WARNING, ERROR 모두 저장

WARNING 모드 : WARNING, ERROR 저장

ERROR 모드 : ERROR 저장

**시스템로그 파일저장**

파일이름 : 년월 + 타입.txt 로 저장.

**경험상 일단위는 파일이 너무 쪼개지고, 주단위는 분류가 애매하므로 월단위 저장이 가장 편리.**

ex) 201509\_Battle.txt

파일내용 :

[Battle] [2015-09-11 19:00:00 / DEBUG] 로그문자열.........

[Battle] [2015-09-11 19:00:01 / WARNG] 로그문자열.........

[Battle] [2015-09-11 19:00:01 / ERROR] 로그문자열.........

위와 같이 줄 단위로

종류 / 날짜,시간 / 로그레벨 / 로그문자열 을 저장한다.

로그문자열 앞까지는 줄을 맞춰야 대량로그를 확인하기 용이하다.

**시스템로그 파일저장 – 로그카운터**

시스템로그를 저장할 때, 인터락을 심으면 굉장히 도움이 된다.

[Battle] [2015-09-11 19:00:00 / DEBUG / **000000001**] 로그문자열.........

[Battle] [2015-09-11 19:00:01 / WARNG / **000000002**] 로그문자열.........

[Battle] [2015-09-11 19:00:01 / ERROR / **000000003**] 로그문자열.........

**멀티 스레드 환경에서 스레드가 치고들어오는데 컨텐츠별로 파일까지 나뉘어져있다.**

**(게임로그는 컨텐츠분류 X 시스템로그는 컨텐츠분류 O)**

A유저가 B유저를 처치하고 아이템을 얻었다고 가정하자.

이경우 로그는 배틀로그와 아이템로그 두개가 남을 것이고, 로그가 쌓이다보면 순서를 알아보기 힘들어 진다. (초단위 로그가지고는 정확히 판별하기 어렵다)

**따라서 로그카운터를 심어 순서를 더 명확하게 한다.**

**컨텐츠(TYPE)별 분류**

**게임로그 - DB저장 – 시간별 분류**

**시스템로그 – FILE저장 – 컨텐츠분류**

**DB는 하나의 테이블에 몰아서 저장하고, 검색이 용이하다.**

**FILE은 검색이 어려우므로 종류별로 분류하는게 편리하다.**

201509\_Battle.txt

[Battle] [2015-09-11 19:00:00 / DEBUG / 000000001] 로그문자열.........

[Battle] [2015-09-11 19:00:01 / WARNG / 000000004] 로그문자열.........

[Battle] [2015-09-11 19:00:01 / ERROR / 000000006] 로그문자열.........

201509\_Item.txt

[Item] [2015-09-11 19:00:00 / DEBUG / 000000002] 로그문자열.........

[Item] [2015-09-11 19:00:01 / DEBUG / 000000003] 로그문자열.........

[Item] [2015-09-11 19:00:01 / DEBUG / 000000005] 로그문자열.........

위와 같이 2개의 로그 파일이 남겨질 때 한 로직의 과정에서

여러개의 로그 파일로 저장이 되기도 하는데 이에 대해 정확한 순서 확인이

필요하다면 로그 카운터를 확인하면 된다.

**시스템로그 파일 Open/Close**

LOG(xxxxxxx);

함수 호출시 매번 File Open > Write > File Close 는 기본.

매번 파일을 열고닫는 비효율적인 방법을 택하는 이유

**1. 서버가 운영되는 상황에서 로그 파일을 열고 확인이 가능해야 한다.**

**2. 경우에 따라서 로그 파일을 서비스 중 삭제 해야하는 경우도 있다.**

**(반복 오류로 인하여 너무 용량이 커지는 경우 기가단위가 되면 열고 확인이 어려움)**

**멀티스레드 환경에서 로그(파일)저장**

**멀티스레드 환경이므로, 로그(파일)저장에 실패하는 경우가 있을 수있다.**

**1. 같은 타입의 로그를 두곳이상에서 남기려고 하는 경우**

이를 막기위해 동기화객체를 사용하는경우,

**병렬로 돌아도 상관없는 로그들이 동기화 객체로인해 돌지 못하는 상황을 막기위해서 타입종류별로 동기화 객체를 나눠야 할것이다.**

**단, 성능에 영향을 줄 정도로 로그를 쏟아내고있다면 이미 문제가 심각한 상황.**

이를 고려하지않고 가도 큰 문제는없다.

**2. 동시접근이 아닌데도 파일오픈 자체가 실패하는경우**

백신프로그램이 도는 경우 파일오픈에 실패하는 경우가 존재한다.

최근 백신들은 파일의 변화가 감지되면 이 파일에 들어가 확인하고 나오는 작업을거친다.

**굉장히 짧은시간이지만 우리서버와 같이 빈번하게 파일 읽기/쓰기가 반복된다면 파일이 열려있는데 파일오픈을 시도할 수도있다.(실패)**

**로그기록은 굉장히 중요하므로 실패시 일정횟수 재시도 이후 서버를 꺼야한다.**

(이는 개인적인 생각. 물론 신입때 결정할 사항이 아니다)

이렇게 재시도하는 코드를 넣은경우, 파일은 중복 오픈이 불가능하기 때문에 동기화객체를 넣지않아도 돌아가게 될것이다.

**시스템로그 파일저장 코드**

// 일반 포멧 형식의 텍스트 로그 남기는 함수

**void Log(WCHAR \*szType, en\_LOG\_LEVEL LogLevel, WCHAR \*szStringFormat, ...);**

가변인자 함수를 사용한다.

va\_list va;

va\_start(va, szStringFormat);

HRESULT hResult = StringCchVPrintf(szInMessage, 256, szStringFormat, va);

va\_end(va);

**로그 함수 내에서는 위와 같이 va\_list 와 StringCchVPrintf 를 사용한다.**

**시스템로그는 안전해야하므로 스트링관련 기능시 반드시 strsafe.h의 문자열 함수를 사용한다.**

**시스템로그는 우리의 최후의장치라고도 알수있으므로, 해당 로그자체에 문제가 생기면 안될 것. 무조건 성공해야한다.**

따라서 버퍼를 초과하여 데이터가 절사되는 경우, 이조차도 로그로 남겨야 할 것이다.

**로그오류에 대한 로그를 남긴다면 재귀적으로 들어갈수있는 위험이 생기게 되는데, 이때는 어떻게든 알아볼수있도록 장치를 해둘 것. 이는 게임로그도 마찬가지이다.**

(파일 이름을 남기다가 짤릴수도 있음)

16진수(HEXA)

// 바이너리 데이터를 Hex 16진수로 남기는 함수.

void LogHex(WCHAR \*szType, LOG\_LEVEL LogLevel, WCHAR \*szLog, BYTE \*pByte, int iByteLen);

바이너리 값 자체가 확인이 필요한 경우가 있을 때 사용한다.

(메시지 수신버퍼 내용 확인, 원인미상으로 객체 메모리 자체 로깅 등...)

**LogHex 사용처**

Recv했는데 마샬링이 되지않는 경우, 해당 클라는 끊도록 조치했다.

**이때 단순히 끊는 것 뿐 아니라 무엇을 보냈는지 확인하고, 이러한 데이터가 빈번하게 온다면 해당 IP또는 해당 유저를 차단시켜야 한다.**

**이 경우 간혹 클라쪽 버그인 경우가있는데, 이떄 16진수로 모두 찍어서(LogHex) 무엇을 보냈는지 메모리를 하나하나 확인해볼수 있다.**

이밖에도 수신버퍼/객체메모리 자체를 저장해야 하는 경우 유용하게 쓰인다.

**시스템로그 콘솔출력**

# CONSOLE (화면출력) ----------------------------

파일에 저장되는 내용 그대로를 화면에 printf 로 출력한다.

간혹 문자열이 너무 길어지는 경우 일부 항목을 제외할것.

**서버개발자의 코딩 외 작업**

**서버에대한 큰 작업을 했다면 프로토콜 설명 등 구체적인 내용을 모두 문서화하여 남겨야한다.**

실제 현업에서는 코딩외에도 문서작업이 굉장히 많을 것.

**실제 남긴 로그살펴보기**

첨부파일 없음. 수업시간에만 공개.

**1. 블락된 IP**

국가별로 서비스할 때 사용한다.

국가별IP를 제공하는 웹사이트를 이용한다. (정확도 98%)

방화벽 같은 경우도 국가차단이 내장되어있지만, 결국 해당 DB를 활용한다.

기기별로 갱신하면서 csv별로 모두 공개한다.

차단할때마다 IP와 계정같은 것이 나오는데, 가끔 잘못판단된 경우를 확인해야한다.

(문의가 오는경우 확인)

**2. 응답이 없는 경우**

150초 타임아웃 처리.

특정 시간동안 수신이 전혀 되지않으면(하트비트 포함) 끊는다.

이때도 로그를 남긴다.

보통 동접 3~4만명 정도 될경우 열댓개정도 남는걸 볼 수 있다.(생각보다 많이남음)

이때 원인미상으로 급격하게 이 수치가 늘어난다면.. 추적해서 파악해야 할 것.

**3. 체크섬 오류**

우리는 체크섬을 위변조를 판단하기 위해 넣는다.

**간혹 패치이후 클라쪽에서 패치가 되지않아 이런 오류가 잡히는 경우가 있다.**

이경우 문의가 왔을 때 클라가 재설치하라는 안내라도 해야하기 때문에 파악해야한다.

**4. 프레임딜레이**

이전더미에도 포함되어있었다.

**1초에 얼만큼의 루프를 도는지 파악하는것으로, 서버에도 들어간다.**

**메인 업데이트로직에서 루프돌때마다 일정시간이상이 걸리는경우 로그를 남긴다.**

(불필요하게 매번남기지 않음)

**만약 100프레임(10m/s한번 주사)인데 200m/s이 넘어가는경우 로그를 남긴다.**

(로그를 남기는 상황이면 = 유저는 더더욱 렉을 느낄것이다. 빠르게 원인파악을해서 없애야할 것.)

모든 오류가 잡혔다면 서서히 기준치를 낮춘다. (200m/s -> 100m/s -> 50m/s..)

**5.중복 로그인**

같은 계정으로 두번의 로그인이 들어왔을때, 로그조차 남기지않고 중복되는 계정을 끊어버린다면 이는 전혀 파악이 되지않을 것이다.

**기타 추가사항**

**로그중복**

로그는 중복으로 많이 남는다.

(전사람이 남긴걸 모르고 새로만들어 남기거나, 또는 전사람 로그가 구체적이지 않아서 수정하고싶은데 혹시라도 문제가있을 수 있으니 새로만들어 남기는 등..)

**오래된코드**

데미지가 10000이상을 핵으로 처리해버리는 로직이 있다고 가정하자.

이때 새로운 컨텐츠로 인해 10000이상이 뽑힐경우 일반유저들또한 핵유저로 판단한다.

프로그래머는 기획자 수식을 그대로 넣을뿐이고, 기획자도 최대데미지를 모르는경우가 허다.

이는 몇 년적 로직을 모두 확인할 수 없기 때문에 종종생기는 문제.

**운영자계정 감시**

운영자 계정은 명령어를 주게된다.

(아이템생성/유저강제 이동/이벤트 권한등..)

**이때 운영자 계정이 해킹당하는 경우 심각한 사항을 초래하기 때문에 운영자 계정에 대해 모두 로그를 남긴다.(채팅까지 포함된다.)**

**이는 DB차원으로 남기는것도 좋은 방법이다.**

해킹 말고도 운영자가 유저에게 해코지를 하거나 비리를 저지르는 경우도 막아야 한다.

서버 프로그래머가 로그까지 다 지워주고, 로그서버에 남은 로그인기록, 실시간 백업등을 처리한다면 완벽범죄가 될수 있겠지만 쉽지않을 것.

**현업에서의 작업**

원인미상의 에러상황이 발생하는 경우

**게임로그 서치, 시스템로그 서치 및 변경, IDC센터에 해당시점에 문제가 있었는지 확인 등 얻어낼수있는 모든 정보는 다 수집해서 판단해야 한다.**

어떻게든 코드를 넣어서라도 추적을 해야하는 상황.

서버작업은 이런작업의 일상이다.

1~2년차에는 컨텐츠 코딩을 하게될것이다.

이후 시니어급 이상이된다면 버그를 해결하는 등의 작업을 할 것이다.

**단순히 컨텐츠적인 버그가 난다면 서비스해도 상관없겟지만, 크리티컬한 버그로 Crash가 나는데 디버깅이 되지않는 상황이라면 끔찍하다.**

주어지는 것은 덤프밖에 없다.

버그잡을때는 로그가 몇백MB가까이 남을 수도 있다.

**다른서버(MySQL, 아파치 등)**

MySQL이나 아파치도 기본은 파일이고, 가동/로딩될 때 해당 프로세스 옆에 모두 로그를 남긴다.

서버를 20개 운영할경우 서버마다 로그파일이 생기기 때문에 관리가 힘들어진다.

이를 일괄관리하고자 한다면 주기적으로 스케쥴러를 통해 수집하는 솔루션을 갖추던가, 배치파일등을 이용해 코딩하여 준비하는 등의 작업이 되어야 한다.

**DB저장와 파일저장**

**DB에 쏘는 것은 유실될 가능성이 존재하기 때문에, 파일에 남길 수 있다면 파일에 남겨야한다.**

하지만 파일의 경우 관리가 힘들 수있다.

저 같은 경우 파일을 선호해서 아침마다 터미널에서 서버와 로그를 확인하는 형태를 선호하는데,

이 방식이 귀찮다면 별도 네트워크로 연결된 서버나 특정폴더에 자동으로 카피하게 써도된다.

**시스템로그 보존**

**시스템 로그도 결국 모두 백업하여 보존시켜야 할 데이터다.**

그리고 이 시스템로그도 결국은 다 백업하고 보존시켜야될 가치있는 데이터다.

**우리는 새로운 컨텐츠가 추가되면서 조금식 변하는 수치에 무뎌진다.**

**하지만 이전 로그를 보고 년단위로 비교하면 경우 확연하게 티가날것이다.**

(소스코드역시 모두 버전관리 된다)

**이 경우 어떤 패치이후 이런 문제가 발생했는지 확인해봐야 할 것.**

**콘솔 출력 로그**

printf출력하되, 쓸데없는것을 줄이고 줄을맞춰 보기용이하게 만든다.

(년월일시분초 등을 제거하고, 로그레벨 정도만 간략하게 한줄에 나올수있게끔)

**게임로그 제공**

게임로그 같은 경우는 관리자페이지로 제공하여, 날짜/계정을 기재하면 해당 로그가 웹에 보이게끔 웹개발자가 페이지를 만들 것이다.

이는 우리가 할일은 아니다.

**<7.31> 빠짐. (49제. 수서) 16기 수업 11.6으로 대체**

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

**<11.6>**

**종료에 대한 추가적인 언급**

**4Way handShaking도중 종료절차가 완전히 진행되지 않는 경우**

**보낸 FIN에 대한 ACK가 안오는경우, 또는 못받은 경우 등이 존재하겠다.**

이때는 는 Shutdown을 아무리해도 Recv 완료통지가 떨어지지않는다.

(단, 지금예제에서는 이런 케이스는 없을 것)

**데이터를 보내고있는 도중 shutdown을 보내는 경우,**

Send쪽 완료통지는 떨어진다.

Recv의 경우는 어떻게 될지 알수없음.

**끊어지지않고 TIME\_WAIT이 남는 경우**

TCP입장에서는 떨어지지않으면 정리단계가 아니다.

**종료되지 못하고 TIME\_WAIT상태로 남아있는 소켓도 일정시간(몇십초 가량) 지나면 OS가 정리한다.**

OS가 정리해주는것도 Recv 0에 대한 완료통지가 오게된다.

TIME\_WAIT이 남은상태로 그냥 둔다는것도 말이안되는것은 아님.

**TIME\_WAIT이 남았을때 문제**

리소스가 남아있는것에 더불어 한가지 문제가 되는 상황을 알아본다.

중복 로그인 같은 경우,

예전에는 나중에 들어오는 유저를 끊었지만 최근추세는 접속중인 유저를 끊는다.

이때 TIME\_WAIT이 남아서 끊기지않는다면 기존접속 유저가 끊기지않고 남아있을 것이다.

**가능하다면 이러한 상황은 만들지 않는것이 제일 좋기때문에, shutdown을 권장하지않는다.**

**Cancel I/O**

Recv를 걸어놓은 상태에서, TCP스택에 데이터가 들어와 내 송신버퍼에 복사하는 과정에서 CancelI/O를 호출한다면 Transferred가 0이 떨어지지않을 것.

이같은 경우는 우리가 로직으로서 예외처리를 해줘야 한다.

**Send하고 곧바로 CloseSocket하는경우**

상대방이 데이터를 모두 받았다면 (Send의 데이터와, RST든 FIN이든)

**내 수신버퍼에 데이터가 있더라도 FIN이 오는경우 FIN먼저 처리한다.** (데이터는 버려진다.)

FIN이 먼저 도착하는것은 아니지만, 받는쪽에서 이와 같이 처리하는것.

**네트워크 라이브러리의 개념**

우리는 현재 네트워크 로직들을 라이브러리 개념으로 만들고자 한다.

전역함수로 만들수 있겠지만 독립성,캡슐화 등을 위해 클래스화 한다.

해당 클래스 안에서는 WorkerThread, AcceptThread등이 들어가게되고 접속자관리/세션관리/송수신관리 모든것을 책임진다.

사용자(컨텐츠)쪽에서는 해당 네트워크 라이브러리를 가져다쓰면 네트워크쪽은 끝나는것.

**네트워크 라이브러리와 다른 클래스와의 차이점**

직렬화버퍼, 링버퍼, 메모리풀같은 것들도 우리의 클래스로 빼서 사용했다.

단지 스레드의 존재유무차이는 아니다.

**일반적으로 만든 클래스의 경우, 우리쪽에서 호출을 해서 사용한다.**

**네트워크 라이브러리는 반대로 내부에서 코드가 돌고 우리쪽으로 알려주는 형태이다.**

이와같은 형태로는 어떤것이 있을까?

**기본은 네트워크쪽에서 컨텐츠쪽으로 전달된다.**

**SendPackcet(), Disconnect()와 같은 함수로 네트워크쪽에 요청할수도 있다.**

**NetWork Contents**

**AcceptThread ㅡ>**

**WorkerThread ㅡ>**

AcceptThread, WorkerThread등이 컨텐츠쪽으로 결과를 통지해서 알려줘야 한다.

**따라서 가상함수, 함수포인터, 펑터 등의 방법이 존재한다.**

**가상함수의 사용**

**가상함수를 이용한다면, (가장편함) 해당 라이브러리함수를 컨텐츠쪽에서 상속받아 사용하는 모양새가 된다.**

핸들러 역할이 되는 가상함수가 존재하고, 상속을 받아 해당 가상함수 알맹이를 채워나가면 callback이 간단하게 구현이 가능해진다.

(핸들러 - 함수포인터)

따라서 컨텐츠 쪽에서 명시적으로 호출하는것이 아니라, 특정상황이 됐을때 자동적으로 호출당하는 입장이 된다.

**흔히 접할수있는 IOCP모듈화**

**IOCP클래스 / TCP클래스 / USER클래스**

**IOCP클래스 - 실제 네트워크 파트가 돌아가는 부분**

**TCP클래스 - IOCP클래스에 등록**

**USER클래스 - TCP클래스를 상속하여 정의, 알맹이를 채움**

**TCP클래스**

**TCP라는 이름의 클래스를 두고 내부에 소켓과 간단한 송수신 버퍼를 멤버로 두고있다.**

TCP플레이어를 상속받아 Player, Session 등의 이름으로 포장한다.

Accept만 업로드하는 클래스를 MyAccept등의 이름으로 상속받아 기능을 확장한다.

(새로운 사용자가 왔을때 무엇을 할지)

**그리고 MyAccept는 우리가만든 TCP클래스와 연계한다.**

**TCP를 상속받은 플레이어들을 등록시켜주면, 새로운 소켓이 들어왔을 때 플레이어를 생성시킨다.**

**IOCP클래스**

**IOCP클래스에 TCP클래스를 등록하고 TCP포인터들을 관리한다.**

TCP클래스를 상속받아 Session을 만들고, 이를 IOCP클래스에 등록한다.

최종적으로는 IOCP클래스에서 가상함수를 통해 우리쪽으로 전달시켜준다.

**USER클래스**

TCP클래스를 상속받아 USER클래스를 만들어 알맹이를 채운다.

USER클래스에 대한 생성과 관리는 우리쪽에서 한다.

IOCP클래스는 TCP에 대한 포인터들을 관리한다.

뭔가 네트워크쪽의 일이 완료된다면 USER쪽으로 송신을 넘겨준다.

**Accept클래스 동작예시**

Accept클래스 – 우리쪽에서 알맹이를 채워준다. (Accept스레드 내장)

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

IOCP클래스 - TCP에 대한 포인터관리 및 처리. 가상함수를통해 우리쪽으로 전달한다.

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

TCP클래스 - 상속받은 내용대로 실질적으로 생성하고, IOCP에 등록한다.

**이제 실제로 accept가 될때마다 TCP클래스에서 실질적으로 생성되고, IOCP에 등록된다.**

일반 IOCP모듈은 Accpet전담 클래스, TCP전담 클래스가 따로나와서 세션마다의 송수신, 송수신버퍼, 소켓, Send의 행위가 들어간다.

따라서 생성주체는 USER클래스(우리쪽)가 되어, 던져줄수밖에 없다.

IOCP클래스 내부에서는 우리가 상속받은 유저를 알수없으므로, 해당객체를 만들어 던져주는 것.

**Accept클래스 -> 유저클래스 (상속하여 생성)**

**Accept클래스 -> IOCP클래스 (유저를 IOCP로 등록)**

**AcceptMy에서 가상함수를 상속받아 알맹이를 채워놓으면 실제 Accpet()가 호출될 때 우리쪽으로 핸들링 해준다.**

**그럼 여기서 유저를 생성(new) 그리고 해당 포인터를 IOCP에 등록.**

**네트워크 라이브러리 내부로 모두 통합시키는 방법**

이형태는 모듈화입장에서는 좋을지 모르나 노출이 된 형태이다.

**우리가 택할 방법은 네트워크 라이브러리내부로 모두 포함시키고 외부에 노출시키지 않는다.**

세션역시 완전히 감춘다.

**네트워크쪽에서는 소켓이라는 변수를 통해 Send/Recv를 했듯, 컨텐츠쪽에서는 SessionID가 소켓의 역할을 가지고 보내기/받기/끊기를 수행할 것이다.**

**일반적인 IOCP모듈화의 장단점**

위와같이 일반적인 IOCP모듈화는 옛날방식이다.

보통 세분화된 것을 장점으로 꼽는다.

Accept는 Accept만을 진행하고, TCP는 하나의 소켓대상으로 TCP통신을 전담한다.

IOCP쪽에서는 WorkerThread에 대한 것을 전담.

내부에서 또다시 모듈화를 진행하는 것.

**프라우드넷과 일반IOCP모듈의 차이점**

프라우드넷은 USER클래스가 모두 Stub안에 들어가있고, 핸들러 역할을 하는 인터페이스만 바깥으로 빼놓고 등록하는 형태이다.

**프라우드넷 - RPC 적용**

프라우드넷은 엔진이라는 입장으로 RPC라는 개념을 적용해야 한다.

**프라우넷 내부에 객체를 하나두고, 내부에서 WorkerThread와 AcceptThread를 둔다.**

**우리가정의한 패킷에 대한 메시지를 디자인하면, 메시지마다 자동으로 함수들이 자동으로 만들어진다. 그리고 Switch-case문을 거쳐 함수코드까지 호출되는것이 자동으로 내부에서 구현된다.**

**따라서 메시지마다 핸들러(On…함수들)가 유동적으로 나와야 하는 상황이다.**

메시지가 유동적이므로 네트워크 본체라이브러리 내부에 핸들러가 들어갈 방법이 없음.

(우리는 가상함수를 가지고 간단하게 핸들러 역할이 될 함수들에 대한 이름을 고정으로 정해서 박아놓고 가기 때문에 간단하게 갈수있다. (OnClient, OnClineLeave, OnRecv OnAccept등))

**1. 내가정의한 메시지에 따라 stub클래스(인터페이스)의 가상함수가 만들어진다.**

stub이라는 이름의 클래스로, 가상함수만 존재하는 인터페이스 클래스가 있다.

**stub클래스 내부에는 우리가 정의한 메시지에 대한 Proc가 요소마다 가상함수로 (자동으로) 만들어져있다. (껍데기)**

(우리는 통합된 클래스로 진행할 것이고, 이는 핸들러 역할을 하는 함수들이 정해져있기 때문에 단순하게 넣어도 상관없음. 하지만 RPC입장에서는 완전히 동적으로 만들어져야 한다.)

class STUB (인터페이스 클래스)

메시지 1->

메시지 2->

메시지 3->

우리가정의한 메시지마다의 프로시져가 가상함수로 자동으로 만들어진다.(껍데기로)

메시지 처리기-----------

class Stub

virtual ㅡ

ㅡ

ㅡ

ㅡ

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

**2. 하나하나 요소에 대한 내용들을 상속받아, 알맹이를 채운다.**

가상함수로 이루어진 인터페이스 클래스 stub을 상속받아, 각각의 메시지마다 실제 Proc할 내용을 채워야 한다.

**3. 이제 변경시킨 stub을 프라우드넷 클래스에 어태치(포인터 등록)시킨다.**

프라우드넷 클래스 입장에서는 외부함수들을 알수없으므로, 내부에 ProcMessage()함수를 고정으로 두고 대량의 switch-case문을 가진다.

**4. 함수를 호출시켜 포인터를 등록하면, 프라우드넷 멤버에 보관한다.**

stub클래스 내부에서는 바깥에서 받은 stub에 대한 포인터, 클래스를 대상으로 프로세스 메시지를 호출시킨다.

그리고 내부적으로 분기문을 타고 MessageProc가 호출되는 구조로 간다.

**5. 메시지 처리가 완료되면 컨텐츠쪽으로 넘겨준다.**

**프라우드넷 클래스에 등록시켰다면 Recv완료통지가 왔을 때 WorkerThread에서는 메시지를 확인하고 처리가 완료되면.**

MessageProc에서 분기타서 가상함수를 통해 우리쪽 코드로 넘어오게 되는 것.

그럼 여기서 분기타고 실제로 가상함수를 타고 우리쪽 코드로 넘어오게된다.

**이렇듯 복잡하게 가는 이유**

우리가만들 클래스 같은 경우는 어떤 핸들러 역할이될 함수들이 정해져있기 때문에 단순히 넣어버려도 된다.

**하지만 프라우드넷의 경우는 메시지 자체가 동적으로 진행되어 프로그래머가 원하는대로 추가/삭제가 되어야 한다.(RPC)**

만약 모든 코드를 클래스하나에 몰아넣는다면 코드를 짤때마다(사람이 바뀔때마다) 클래스가 바뀌어야 한다.

**따라서 stub이라는 클래스 인터페이스 덩어리를 따로빼놓고, 실제 코드에 대한 작업은 우리가 하고싶은대로 알맹이를 채우는 것.**

**중간정리**

기본인터페이스로(가상함수), 기본동작을 하는 시스템적인 핸들러를 둔다.

OnClientJoin, OnClientLeave, OnConnectRequest, OnError.

**프라우드넷은 가상함수가 존재하는 인터페이스 stub클래스가 존재하고, 이를 우리쪽에서 상속받아 프라우드넷에 등록한다.**

(우리가만들 네트워크 라이브러리는 이거를 그냥 상속받아 버리는것으로 설계한다.)

**프라우드넷 방식의 경우는 프라우드넷 클래스를 상속받는 구조가 아니고, 프라우드넷 클래스 인스턴스 자체가 단독으로 존재한다**.(new로 생성하든, 메인의 지역으로 선언하든..)

그리고 제공한 인터페이스 클래스를 상속받아 어태치 시키는 구조.

순수가상함수로 stub을 상속받아 우리쪽 Cpp코드에서 알맹이를 채우고 프라우드넷에 어태치하면, 우리쪽 코드와 연동이 된다.

**우리가 택할 방법**

위와 같은 방식도 괜찮은 방법이지만, 우리는 **이런 메시지에 대한 처리나 분기문자체를 컨텐츠로 넘긴다. 우리입장에서는 이것까지 굳이 자동화시킬 필요가 없기때문.**

**우리는 네트워크 라이브러리안쪽에 가상함수를 넣어버리고 직접상속해서 사용하는 방법을 택한다.**

단, 우리가 택하는 방법은 아니지만 프라우드넷 방식도 숙지하고 있어야 할 것.

**함수 이름은 프라우드넷을 따라간다.**

RAN서버, RAN클라이언트와 같은 이름으로 간다.

이 경우 면접관이 프라우드넷 서버 설계로 진행했냐고 질문이 들어올 수 있음.

**LAN과 WAN의 용도분리**

프라우드넷에서는 LAN과 WAN의 용도가 나뉘어져있다.

LAN : 서버-서버

WAN : 일반클라-서버

LAN통신과 WAN통신을 한쪽에 몰아넣고 코드상에서 메시지를 나눠 구분할수도 있으나,**내부통신(LAN)인 경우 사설IP로(private) 진행하기 때문에, IP자체에서 막는게 훨씬 간단하고 바람직하다.**

(일반 사용자들은 0.0.0.0, 서버통신은 내부용 172.16.n.n 등으로 바인딩되는 식)

**WAN통신에서는 조작을 조금이라도 방지해보고자 체크섬/암호화 같은 것들이 들어가고,**

**반대로 서버간 통신(LAN)은 일반 사용자 통신(WAN) 보다 신뢰할 수 있는 환경이라는 전제로, 길이에 대한 헤더만 넣고 들어간다.**

(WAN, LAN에 따라 프로토콜 자체도 분리하여 사용할 것이다.

우리 프로젝트에서 들어간 2byte헤더가 LAN통신용 프로토콜로 들어가게 될 것)

**서버끼리 통신하는 LAN클라이언트 - LAN서버와**

**일반클라를 대상으로하는 NET서버 - NET클라이언트**

를 세트로 만드는것으로 진행한다.

왜이렇게 나눴는지에 대한 이유를 잘 염두해 둘 것.

**서버간 통신(LAN)**

**# 서버간 통신 모듈 제작.**

**목적: 네트워크 서버-서버간의 통신을 목적으로 한다.**

- 대규모 접속처리 필요성 X (일반적으로 몇십개, 정말 많은경우도 600개정도 상회)

(대규모 접속처리와 소규모 접속처리에 대해 구현이 달라지는 것은 없으나, 우리는 동적할당이 아닌 배열과 같이 미리 확보해놓고 가는 스타일로 가기 때문에 메모리사용 크기가 다를 것.)

**- 최대한 간단한 프로토콜을 사용하며, 보안은 신경쓰지 않는다.**

**- 간단한 사용법을 가지도록 만든다.**

**- 서버간의 통신 이지만 그래도 클라-서버 개념이 도입되어야 한다.**

서버와 서버간의 통신이지만, 어쨌든 Accept, Connect를 해야한다.

따라서 LAN서버간에도 클라역할 - 서버역할로 구분이된다.

프라우드넷 같은 경우 LAN클라-LAN서버를 Reliable UDP로 구현되어있다.

(로컬환경에서도 패킷은 계속해서 손실이 나기 때문에, 이를 염두하고 있어야 할 것.)

(Connect()함수의 경우 SendFrom, RecvFromt시마다 주소를 받기 번거로우므로 저장해두는것으로, 네트워크 상에서는 어떠한 기능도 없다.)

**UDP로 구현한다고 해도 역시 서버-클라 개념이 나와야하는데,**

**UDP는 IP와 포트를알고 Send하면 끝이지만 여기서는 결국 Reliable UDP가 구현되어야 하기 때문에 TCP의 기능이 구현되어야 한다.**

3Way handShaking에 준하는 연결과정 등이 나와야 할 것.

**내부통신을 UDP로 송수신한다고 해서 동등한 입장에서 서로 쏟아붓는 것이 아니다.**

**- 유저간 통신 모듈을 사용하여, 유저처럼 서버를 연결하여 유저와 똑같이 처리하는것도 가능하지만, 분리하는 이유**

기능상 구현하는 것은 아무문제가 없으나,

**서버간의 데이터 송수신과 일반유저의 송수신이 같은 분기문에서 처리된다면 시스템적인 메시지와 유저컨텐츠 처리가 같이 들어가는 모양이된다.**

**1. 성능 영향**

**추가적으로 서버간의 통신은 중요도가 높으므로, 클라와의 통신량이 많다고해서 서버간의 통신량이 떨어져서는 안될 것.**

**따라서 유저수와 상관없이 성능이 보장되도록 한다**.

**2. 보안상 문제**

애초에 프로토콜이 공개되지않기 때문에 알아내기는 힘들것이나 찝찝할 뿐더러 퇴직자가 접속하는 등의 예외의 상황이 있을수 있음.

**3. 프로토콜**

유저간 통신 모듈은 대체적으로 무거운편이며, 프로토콜자체도 사이즈가 크기 때문에 서버간 통신은 개별적으로 분리하여 간단하게 구현한다.

따라서 IOCP핸들/스레드를 모두 분리시켜 메시지 프로시져자체를 따로 처리한다. 관리상에도 이점이 있을 것.

일반적으로도 서버간의 통신은 별도의 랜카드를 사용하여 일반 유저접속용IP와는 다르게 간다.

**# 서버간 통신 모듈 설계**

**클라역할 - CLanClient**

**서버역할 - CLanServer**

**- 설계 가이드.**

**\* IOCP 모델을 사용함.**

**\* 클래스 내부에 워커 스레드를 가짐.**

**\* 이벤트 함수 (접속완료,받기완료,보내기완료) 는 순수가상 함수로 제작.**

**\* 실제 사용부는 위 두개의 클래스를 상속 사용하도록 제작.**

**\* 내부에서 자체적으로 세션 연결을 관리함.**

**(세션은 외부에 노출되지않는다.)**

**- bool Start(...) 오픈 IP / 포트 / 워커스레드 수 (생성수, 러닝수) / Nagle옵션 / 최대접속자 수**

**해당 인스턴스를 선언(new로 생성/메인함수에 지역으로 선언)이 되어있어야한다.**

인자로 IP, 바인딩포트, 워커스레드 숫자 등을 지정할 수 있게끔 한다.

Nagle옵션을 여기서 지정한다는 것이 애매할 수 있다. 알아서 선택하여 갈 것.

WorkerThread/AcceptThread를 언제 만들것인지는 선택사항이다.

생성자에서 호출될수도 있겠고 Start()함수에서 호출될 수도있겠다.

**어쨌든 Start()함수가 호출되면 서버가 가동되어 접속자 받기가 시작되어야 한다.**

**Start에서는 여러가지 에러(바인딩에러 등..)가 날수있기 때문에 반환값으로 성공여부를 알려준다.**

**- void Stop(...)**

Stop의 경우 컨텐츠쪽에서 서버를 끊어달라고 요청이 온 것으로, 네트워크 라이브러리 쪽에서 호출되어서는 안된다.

메모리정리, DB백업 등의 작업은 컨텐츠랑 연관된 서버종료 작업이다.

이 함수는 네트워크 라이브러리 입장이다.

**1. 일단은 Listen소켓을 닫아 Accept를 닫아야 한다.**

**2. 연결끊기에 대한 행동이 들어간다. (shutdown/closesocket)**

**세션정리는 IOCount가 0일 때 종료되도록 했으므로, 연결끊기로 모든IO가 완료되어 Session이 정리되도록 유도한다.**

따라서 현재 접속한 모든 세션을 대상으로 Disconnect를 요청하고 기다리는 형태가 될 것.

(정상적인 경우 길어도 몇초정도의 시간안에 모두 끊어진다.)

접속중인 Session이 0이되었다면 이제 WorkerThread를 종료한다. (종료 유무는 선택사항이다.)

WorkerThread를 생성자에서 만들었다면 소멸자에서 호출이 될 것.

((

**서버 중지 후 재가동**

잠시 중단되었다가 재가동 하는 기능을 구현할 경우, 스레드가 가장 문제가 될것이다.

**1. stop할 때 스레드를 모두 파괴시키고 다시 만들기**

**2. 스레드를 그대로 두고 다시돌게끔 만들기. (굉장히 복잡해질 것)**

어차피 stop은 서버를 끄고 다시 키려는것이기 때문에 재시작에 대한 구현은 하지않는다.

))

Stop의 경우는 Start와 달리 반환값이 bool이 되지않는다.

어떻게든 모두 끊고 나와야 하는 입장이다.

PQCS로 종료시킨다 하더라도 안꺼지는 경우가 있을 수있는데, 이 경우 일정시간 기다려보고 terminate로 파괴시킨다.

보통 현업에서도 서버가 꺼지는 경우는 아무도 신경쓰지않기 때문에 다들 엉망진창으로 끈다.

**- int GetClientCount(...)**

현재 접속한 인원 당연히 내부에서 카운팅. 어셉트될떄 ++ 나갈때 --.

**- bool Disconnect(SessionID) / SESSION\_ID / HOST\_ID**

**- bool SendPacket(SessionID, Packet \*) / SESSION\_ID / HOST\_ID**

이둘은 컨텐츠에서 서버로 요청이 들어오는 것이다.

프라우드넷은 hostID라는 이름을 사용하지만, hostID는 의미가 모호하므로 SessionID로 간다.

**virtual void OnClientJoin(Client 정보 / SessionID / 기타등등) = 0;**

**< Accept() 하여 접속처리 완료 후 호출한다.**

**SessionID는 필수. IP포트는 선택사항이다. Socket은 컨텐츠로부터 완전히 감춘다.**

**virtual void OnClientLeave(SessionID) = 0;**

**< Release 후 호출**

모든 종료절차가 끝나고 컨텐츠쪽으로 SessionID를 통해 알려준다.

**virtual bool OnConnectionRequest(IP,Port) = 0;**

**< accept 직후**

**return false; 시 클라이언트 거부.**

**return true; 시 접속 허용**

**Accept를 받자마다 세션도 셋팅하기전에 호출되며, IP와 포트를 인자로 받는다.**

**만약 OnConnectionRequest()가 false를 반환한다면 즉시 closesocket한다.**

그럼 여기서는 이 IP와 포트를 허용할것인가 거부할것인가 결정한다.

허용은 return true

불허는 return false

(테스트환경에서는 모두 허용.)

불허의 경우 특정 IP를 블락시킬 때 사용한다.

**코드차원에서 특정 IP블락**

**대규모로 간다면, 대량으로 쏟아지는 접속에 대한 IP차단을 서버쪽에서 하기에는 부담이 크기 때문에, 방화벽선에서 차단하는게 바람직하다.**

또한 로그인 재접속 같은 경우는 계정이 대상이므로 해당되지않음.

**NET서버역시 LAN서버와 똑 같은 구조로 갈것이기 때문에, OnConnectionRequest는 LAN송신 뿐 아니라 WAN까지 고민한다.**

**코드에서 IP를 블락시키는 경우는**

**1. 내부에서 자동화시켜 뭔가를 일시적으로 해야할때를 사용하거나,**

자동화 코드를 넣었을 때 방화벽에 넣기전 일시적으로 차단할 대상이 생겼을 경우

**2. IP밴을 하고싶을 때 사용한다.**

서버패치 이후 모든 컨텐츠를 테스트 할때, 허용된 일부Ip만 허용하고싶은 경우.

(어찌보면 차단을 목적으로 하는 방화벽과는 입장이 반대이다.)

서버를 키고 화이트IP만 접속이되다가, 특정명령이 떨어지면 그때부터 오픈이 되는식으로 갈 것.

**추가적으로, 서버를 오픈하기전에 운영툴, 관리를 위한 로직, 패치제어, 화이트IP모듈 등 모든 것이 준비되어 있어야한다.**

운영툴, 관리할수있는거, 패치제어, 화이트IP모듈 이것저것 다.

**virtual void OnRecv(SessionID, CPacket \*) = 0; < 패킷 수신 완료 후**

**virtual void OnSend(SessionID, int sendsize) = 0; < 패킷 송신 완료 후**

**Send완료/Recv완료 후 할일이 있다면 넣는다. 필수사항아님.**

프라우드넷에는 OnRecv/OnSend가 존재하지않는다.

프라우드넷은 인터페이스를 통해 바로 던져야 하는 구조이고, 우리는 OnRecv라는 핸들러 하나로 커버한다.

**OnRecv는 SessionID를 가지고 하나의 메시지당 한번씩 호출되는 구조이다.**

**virtual void OnWorkerThreadBegin() = 0;**

**< 워커스레드 GQCS 바로 하단에서 호출**

**virtual void OnWorkerThreadEnd() = 0;**

**< 워커스레드 1루프 종료 후**

OnWorkerThreadBegin/End()는 프라우드넷에 존재한다. (우리입장에서는 선택사항)

WorkerThread가 깨어날때 begin되고, 다시 block되기 위해 GQCS를 호출하러갈 때 End된다.

**외부에서는 네트워크 관련 로직이 완전히 감춰져있기 때문에 어떤 WorkerThread가 어떻게 도는지 알방법이 없으므로, 위 함수들이 알려주는 역할을 한다.**

(디버깅 목적, 스레드에 대한 상태 모니터링 목적 등..)

**virtual void OnError(int errorcode, wchar \*) = 0;**

컨텐츠쪽에서는 뭔가에 대한 요청을 단순히 return값으로 판단하여 진행하지만,

**네트워크 라이브러리 내부에서 코드가 돌고있는 상황에서 문제가 난다면 이를 바깥으로 알려줘야한다. 이때 OnError()를 사용한다.**

혼자서 처리할수도 있겠지만, 해당 문제상황을 바깥으로 통보해주어야 할 것.

바인딩 에러, 직렬화버퍼 할당실패, 링버퍼 문제 등의 에러를 알려준다.

**가상 함수로 둔 이유는 우리쪽 컨텐츠쪽에서 코드가 돌다가 에러가 발생하는 경우 알려주기 위함이다.**

(클래스 내부에서 Try catch로 받는 것은 본인이 아는것이기 때문에 굳이 가상함수로 갈 이유가 없음)

**Recv/Send시 IO\_PENDING이 아닌 SOCKET\_ERROR라면 단순히 끊어버리므로 해당사항아니며,**

**여기서는 바인딩 에러, 내부 직렬화버퍼 할당실패, 링버퍼Inqueue실패 등을 관리한다.**

WSANOBUFS와 같은 에러는 우리 소켓의 문제가 아닌 시스템적인 문제이므로 알려줘야 할 것.

물론 네트워크 라이브러리 안쪽에서 자체적으로 Text로그를(파일) 내부에서 찍을 것이다.

**CLanClient (LAN CLIENT)**

**- bool Connect 바인딩 IP, 서버IP / 워커스레드 수 / Nagle옵션**

**- bool Disconnect()**

**- bool SendPacket(Packet \*)**

**virtual void OnEnterJoinServer() = 0; < 서버와의 연결 성공 후**

Connect후에 성공까지 떨어졌을 때 호출이 된다.

**virtual void OnLeaveServer() = 0; < 서버와의 연결이 끊어졌을 때**

연결이 끊어지고 난 후에 호출된다.

**virtual void OnRecv(Packet \*) = 0; < 하나의 패킷 수신 완료 후**

**virtual void OnSend(int sendsize) = 0; < 패킷 송신 완료 후**

**//virtual void OnWorkerThreadBegin() = 0;**

**//virtual void OnWorkerThreadEnd() = 0;**

**virtual void OnError(int errorcode, wchar \*) = 0;**

나머지 함수들은 LAN\_SERVER와 같다.

**Session객체 배열로 관리**

이제부터 내부에서 Session객체를 배열로 관리한다.

**워커스레드에서 반응이왔을때는 p로 세션이 들어왔기때문에 검색할 필요없음. 없지만, 외부에서 SessionID를 통해 요청이 온 경우 검색해야 한다.**

**어차피 우리는 Session을 알고있기 때문에 어떤 유저에게 데이터를 보내고싶을떄 단순히 Send만하면 끝난다.**

**또한 자료구조를 배열이 아닌 Map이나 UnorderedMap(hash)를 사용하는 경우 find()로 한번에 찾았으나, 검색이 필요한 상황이므로 성능을 문제삼을 수 있다.**

**하지만 우리는 계층을 분리하여 Session과 SessionID로 분리하였고, 자료구조역시 배열로 갈것이기 때문에 검색이 필요해졌다. 따라서 성능에 문제가 될수있는 상황이다.**

**나중에는 검색하는 부분을 없애고 Index를 찌르도록 바꿀 것이므로, 일단은 검색하는것으로 진행한다.**

배열형태이므로, 이제 Session안에서는 사용/미사용에 대한 flag가 들어간다.

Accept가 오는 경우 Session을 찾아서 빈공간에 할당.

**모든 동기화객체 제거**

모든 동기화객체 LOCK은 제거한다. (인터락은 대상아님)

네트워크 라이브러리 어떤곳에서도 동기화객체가 들어가지 않는다.

사실 현재도 (자료구조쪽 LOCK을 제외한) 모든 동기화객체를 제거해도 별다른 문제는 나지않는다.

두개이상의 스레드가 한Session을 건드리는 상황은 Recv완료, Send완료정도 이기 때문.

재연결도 문제없어야 한다.

**서버검증**

**1. 제일 원초적인 방법으로, 서버-클라를 동시에 만들면서 오류를 잡아가는 방법.**

**2. 테스트 클라이언트를 최대한 심플한 환경으로 만들어 (Selcet모델 / block소켓 / 싱글스레드) 서버를 테스트한다.**

IOCP서버를 몇 달을 걸려서 만드는데 더미를 그렇게 만들 수는 없음.

LAN서버/LAN클라를 동시에 만드는경우 역시 문제나는 상황을 잡아내기 힘들기 때문에 굉장히 난해하다.

따라서 현재 가진더미로 LAN서버를 만들고, 이렇게 검증한 LAN서버로 LAN클라를 만든다.

사실 구조가 거의 똑같기 때문에 LAN서버가 완성되면 이름을 NET서버로 바꾸는 것으로 NET서버도 완성한다.

단 일부 프로토콜 헤더가 바뀌는 차이가 있을 것.

수업 중복되는 부분.

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

**더미 테스트시 동접자**

**더미로 50명을 재연결 테스트 시, 더미에서는 끊었다고해도 서버쪽에서는 아직 진행중일 수 있기 때문에(IOCount가 0이 떨어져야 ReleaseSession되므로) 서버에서는 50명을 초과할 수 있다.**

**멀티스레딩 환경에서 파일쓰기**

**CRL은 Thread-Safe한 동기화 라이브러리이기 때문에, 모든 파일쓰기 함수는 CriticalSection이 내장되어있다.**

따라서 파일 포인터가 겹치는 경우는 걱정하지않아도 된다.

**실시간 파일 저장은 굉장히 느리기 때문에, 메모리상에 올려놓고 일정량이 쌓이면 한번에 파일로 남기는 방법도 택할 수 있음.**

**멀티스레딩 환경에서 로그**

파일 입출력은 멀티스레딩에서 안전하지만, 일반적인 로그를 남기고싶다면 락을 써야할것이다.

**이때 로그를 위해 락을 걸고 가게된다면 동기화오류가 나지않을 것.**

**이는 동기화오류를 잡기위해 로그를 남기려고했지만, 이미 락을 걸었기 때문에 동기화문제가 나지않는 현상이다.**

(한번씩 자기할일을 하고 진행되기 때문)

**멀티스레딩 환경에서 디버깅**

파일로 로그를 나기는 경우, IO작업으로 인해 Thread가 block되기 때문에 동기화이슈 문제가 안생길 수 있다.

(심지어 Interlocked을 넣어 사용하는 경우 역시 발생하던 문제가 나지않을 수있음)

**멀티스레드 환경에서 동기화 이슈를 추적하기 위함이라면, 굳이 문자열로 포맷을 남기지않아도 상관없다.**

**특정구역마다 번호를 남겨 어떤순서로 진행되었는지 알아보는 등의 방법을 사용할 수 있을 것.**

**스레드 함수에서 멤버함수 호출**

스레드함수는 static이므로,

**1. 매개변수로 this객체를 받아 멤버변수에 접근한다.**

**2. static함수 내부에 멤버함수를 두고, 멤버함수 내부에서 로직을 처리한다.**

**Start()내부에서 헤더지정?**

컨텐츠 헤더를 지정한다? (X)

**우리가 네트워크 라이브러리를 활용한다면, 네트워크 라이브러리만의 헤더가 있고 실제 게임헤더는 컨텐츠파트의 메시지헤더로 따로 들어갈 것이다.** (ex Type).

**컨텐츠쪽에서 Send를 호출한다면 네트워크 라이브러리 내부에서 네트워크 헤더를 붙여서 네트워크 송수신을 진행한다**.

따라서 여기서 헤더를 지정하는 것은 아님.

**LAN서버 / NET서버**

우리는 현재 LAN서버를 만들지만 LAN서버/NET서버 같은 개념으로 진행한다.

어차피 랜서버는 인원이 적기 때문에 바뀔일은 없다. 하지만 NET서버 같은 경우는 클라이언트 쪽에서 접근하는 서버이기 때문에, 테스트를 거쳐야 한다.

**AcceptEx()**

**AcceptEx는 소켓 재사용과, Accpet()의 비동기 처리를 위한 두가지 용도로 사용된다.**

**Accept()는 소켓생성이되고 closesocket시 소켓이 반환된다.**

**- 소켓을 만들고, Accept으로 소켓이 들어오면 리턴해준다.**

**AcceptEx는 소켓 리소스를 반환하지않고 재사용한다.**

**- AcceptEx는 미리 만들어진 소켓대상으로 OverLappedI/O처럼 비동기로 걸어두는 것으로, 미리 내부에서 소켓을 만들어 두고 재사용한다.**

**Accept() - closesocket()**

**AcceptEx() - transmitfile()**

**AcceptEx는 closesocket이 아닌 transmitfile()이라는 별도의 함수를 사용한다.**

Transmitfile()은 소켓의 리소스는 그대로 두지만 연결정보만 삭제시켜 끊는것으로, AcceptEx에서는 소켓의 리소스를 재활용한다.

**우리가 AcceptEx()를 사용하지않는이유**

**AcceptEx를 사용한다면 성능의 향상이 존재한다.**

**동기화 락을 걸면서 깔끔하게 가는 구조라면 상관없겠지만, 우리가 진행할 구조에서는 굉장히 복잡해지기 때문에 (소켓닫은 입장에서도 굉장히 복잡해짐) 시도하지않는다**.

실제 현업에서 사용하는 사람도 있다.

**대기열**

우리는 WAN서버, LAN서버 똑같이 간다. (헤더만 다름)

요즘게임은 기본적으로 대기열이라는 것이 존재한다.

현재 네트워크 라이브러리는 OSI 7계층 입장에서 구조적으로 본다면 5,6계층이다.

대기열의 기능은 컨텐츠에 가깝다.

**대기열의 기능이 들어가고자 한다면, '연결은 됐지만 대기열상태이다'라고 Callback개념을 통해 클라쪽으로 알려줘야 한다. ( OnEnterJoinServer() )**

**따라서 네트워크 라이브러리 단계차원에서 대기열을 구현하고자 한다면 대기상태/대기번호를 알려줄 핸들러(알려주는 함수)가 나와야 한다.**

이 핸들러를 통해 클라쪽에서 대기열에 대한 정보를 알려줄 것.

(대기열이 없는 경우라면 최대접속자가 모두 찬경우 아예 서버쪽에서 받지않는다.)

**결국 이 구조는 게임서버 내부에 '대기상태'라는 컨텐츠가 별도로 들어가는 것이므로, 대기열로 접속한 클라까지 동접자에 포함이 되어야 한다.**

당연히 컨텐츠 쪽과는 관여할 부분이 아니므로 분리되어있음.

**start()**

**스레드를 만들어 준비하고, listen()함수를 호출하면서 빠져나온다.**

start에서 인자로 들어온 최대접속자 수를 받아 Session과 Index배열을 확보한다.

(최대접속자는 당연히 나중에 외부설정파일로 다 뺀다)

**while()**

**{**

**Accept().**

**Session = accept(-)**

**ID,Port확인**

**OnConnectionRequest(IP,포트);**

**if(let == false)**

**{**

**closesocket(socket);**

**continue;**

**//여기로그는 굳이남기지 않아도 상관없음.(선택사항)**

**}**

**if(최대접속사 확인)**

**{**

**"(동일표시)**

**}**

**세션확보.-> 배열돌면서 셋팅. - (1)**

**OnClientJoin()**

**RecvPost();**

**}**

**배열 Index관리**

(1)의 상황에서, 동접을 2만명으로 잡는다고하면 배열이 2만개 준비된다.

이때 사용유무를 따지려면 루프를 돌아야하고, 운이 나쁜경우 뒤쪽으로 몰려 성능이 저하될수있다.

**따라서 사용하지않는 Index를 가지는 Stack을 별도로 두고 관리한다.**

**Stack을 사용하는 이유**

**일반적으로 Queue(RIFO)보다 Stack(FIFO)이 훨씬 더 가볍다. (캐시히트)**

**또한 스택방식을 사용한다면 방금썼던 것을 재사용하기 때문에, 큐방식보다 문제가 많이 터질것이다.**

**IndexStack의 호출부**

**AcceptThread쪽에서, IndexStack.push()가 호출 될 것이다.**

**이때 POP이 되는 Release부는 어떤 스레드에서도 호출할 수 있는 상황이다.**

(Workerthread뿐아니라 컨텐츠쪽에서도 Release가 호출될 수 있음(Release))

jfksj

**따라서 해당 스택에 동기화 객체가 필요한 시점. (LOCK사용)**

(이후 락프리 스택으로 대체한다.)

**이제 Start()가 호출될 때 Stack에는 배열의 모든 인덱스가 들어가 있어야하고, Release될때는 본인의 Index를 뽑아서 push해야한다.**

자기 Index를 찾으려면 결국 멤버에 보관해야 할 것.

**OnClientJoin - RecvPost의 순서**

만약 RecvPost를 먼저 걸게된다면, 컨텐츠쪽에서는 접속의 유무도 알지못하는데 데이터가 수신되는 것이다.

(마치 Accept로 받지도않았는데 소켓이 들어온경우와 마찬가지)

**따라서 OnclientJoin으로 접속자를 알려주고, 이에대한 처리가 완료되면 RecvPost로 송수신을 시작한다.**

**RecvPost에서 Release되는 문제**

**클라가 접속시도하여 연결되었는데, 내가 Accept로 받기전에 끊어질 수 있다.**

이 경우 네트워크 라이브러리에서는 Accept로 받은 소켓을 Recv로 걸었는데 끊기게된다.

이때 RecvPost내부에서 Release가 호출되는 모양새가 되는데, 이 상황을 고려하고 로직을 짜야한다.

**(Send하다가 10054에러로 Release된다. 이 같은 예외처리까지 가지않도록 그전에 예외처리를 하자는것인가?)**

**접속자Count, AcceptCount**

Accept에서 ++, Release --.

이 둘 스레드는 당연히 동시에 돌수있으므로 이둘의 증감은 Interlocked을 통한다.

**Release()가 중복 되는 경우**

컨텐츠쪽에서 이제 마구 Send()를 하는경우, 중복으로 Release를 타고 들어올수있다.

이는 차차해결해야할 문제.

(추가분 더 적어야함)

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

**<8.3>**

**동기화객체 삭제**

동기화객체를 모두제거하고, 배열을 사용하는 방식으로 진행해본다.

**Start/Stop**

**우리의 서버에서 프로세스를 종료하지않고 정지/재가동의 기능을 구현해본다.**

WorkerThread같은 경우 아예 모두 지웠다가 다시 만드는것도 가능하고, 멈췄다가 다시 도는것도 가능하다.

(정지 후 재가동하는 방식이 코드가 지저분해지고 까탈스럽기 때문에, 정지상태에서 모두 파괴시파괴 재가동할 때 새로만드는 방식으로 가도 무방하다.)

Start(이 함수를 통해 서버가 알아서 가동된다)에서 만들어도 무방하고, 미리 만들어놓고 blocked상태로 두어도 상관없다.

**실제 라이브 서비스에서 서버를 정지했다가 재가동하는 상황은 전혀 없기 때문에, 현업에서는 고려하지않아도 된다.**

**따라서 stop할 경우 스레드를 모두 파괴하고, start(재가동)할 때 다시금 스레드를 만든다.**

**서버 종료/정지**

근데 거의대부분의 서버는 정지/종료가 매우 지저분하다.

어차피 프로세스를 종료시킬 것이기 때문에 롤백만 일어나지않는다면 큰 상관이 없을 것

(ex 프로세스 자체가 꺼지기 때문에 동적메모리가 해제되지않아도 자동으로 해제(?)된다.)

**SessionArray배열**

배열요소가 사용중인지/아닌지에 대한 기준값을 정한다.

이때, 비어있는(사용하지않는) 인덱스를 검색해서 쓰겠다는 의미가 아니다.(이는 차후 해결한다)

**- AcceptThread**

Accept를 기준으로 Block.

(AcceptThread를 어디서 만들것인지는 우리가 알아서 정한다.)

AcceptThread와 WorkerThread는 클래스 내부에서 만들어져야 한다.

(미리 만들거나, Start에서 만들거나..등)

AcceptThread에서 제일 먼저 할일은?

ConnectionRequest도 괜찮다.

**동접이 꽉찬 경우**

동접이 꽉찬것에 대한 판단은 별도의 SessionIndex를 가진 stack을 두고, 이 스택의 사이즈가 0일 경우 꽉찼다고 판단한다.

**AcceptThread에서는 따로 할일이 없으므로 동접이 꽉찼다면 새로 들어오는 클라는 끊어야한다.**

해당 클라에게 어떤 피드백을 해준다는 행위 자체도 컨텐츠 프로토콜이므로 보내기 애매하다.

**대기열을 두는경우**

동접이 꽉차는 경우 대기열을 따로 두는것도 한가지의 방법이다.

대부분의 게임에서 대기열을 사용하고있다.

**대기열또한 프로토콜을 주고받는 컨텐츠이고, 대기중이라는 하나의 게임 상태를 나타낸다.**

**그래픽적으로 표현되지는 않지만 현재 해당 서버에 연결된 상태이고, 클라는 대기방에 들어가 순번을 기다리고 서버는 대기열순번을 프로토콜을 쏴주고 있는 방식.**

따라서 단순하게 연결된 소켓을 어딘가에 보관해놓고 자리가 나면 넣어준다는 방식으로는 해결되지않는다. 애초에 Acccept가 해결할 수 있는 방법이 없다.

(우리가 만드는 네트워크 라이브러리는 소켓API와 같은 형태)

**대기열을 구현하고자 한다면 해당 네트워크 라이브러리를 활용하는 서버에서 이 사용자들 까지 확보하는 충분한 여유분의 크기를 잡아야 한다.**

**더미테스트 재연결**

대기열이 없는 가정하에 내서버 동접자가 5000명이라 가정하자.

이때 5000개의 더미클라를 연결한다고 한다면 서버에서는 이를 정상적으로 받지못한다.

**더미는 연결을 끊자마자 그 즉시 재연결을 들어가기 때문에, 서버쪽에서는 아직 끊지못한 클라가 남아있게된다.**

**따라서 실제로는 기존에 정의한 MAX동접자보다 더 많이 올라갈 것이므로, 1000명정도의 여유분을 넣어줘야 한다.**

단, 이는 더미테스트의 경우에 국한된것으로 일반적인 서비스상황이라면 이렇게 까지 할 필요는 없다.

**AcceptThread**

새로운 사용자가 Accept로 들어온 경우,

(세션에 대한 확인을 먼저할지 OnConnectionRequest를 먼저 할지는 우리가 직접 판단한다.)

OnConnectionRequest에서 나오는 값을 가지고 종료에 대한 판단하여

**종료되었다면 다시 위로 올라갈것이고(while – continue) 아닌경우 세션이 확보될 것이다.**

만약 세션확보에 실패하는 경우 Accept에서 할 수 있는 일이 없으므로 closesocket으로 끊어버리고 나간다.

이후 완료통지를 받기위해 PostRecv로 IO를 걸어놓아야 할 것.

**Acceptthread**

**Accept()**

**OnConnectionRequest()**

**종료판단**

**세션확보(셋팅)** – 실패시 연결끊기

**PostRecv()**

**OnClientJoin()**

**OnClientJoin은 컨텐츠파트에 새로운클라가 들어왔다는 것을 알려주는 역할을 한다.**

**그래서 클라이언트 IP와 포트정보를 담아주고 확보된 SessionID를 담고 넣어주러 간다.**

(소켓과 같은 자료는 컨텐츠 쪽으로 넘기지 않는다.)

**WorkerThread**

WorkerTherad에서는 이제 확실하게 Onrecv, OnSend한다.

WorkerThread쪽으로 들어간다고해도 큰 차이점은 없다.

OnSend는 예외로 두더라도..

OnRecv하고, WorkerThread의 begin/End로 빼기로 했으니 생략.

동기화객체는 모두 삭제한다.

**GQCS완료통지가 오는경우 먼저 에러처리를 판단하여, 종료됐다면 어떻게 할것인가?**

(CloseSocket을 어떻게 할것인지 잘 생각 해 볼것.)

((

**Transferred가 0이 떨어지는 경우는 이미 연결은 끊어진 상태이다.**

(또는 흔히 나오는 문제로, 로직을 잘못짜면 정상인데도 불구하고 transferred가 0이 떨어짐)

**어차피 한쪽에서 에러가 났거나 transferred가 0이 떨어지는경우 다른 IO도 더 이상 작동되지않는다.**

(이전에 언급한 대로 Recv가 걸려있는 상태에서 Send작업 후 closesocket을 해버린다면, transferred가 0이 온 다음에 정상 send완료통지가 올 수 있다. 뭐가됐든 완료통지는 올 것.)

추가적으로, 우리는 closesocket시 필수적으로 LINGER를 걸고 RST쏘는걸로 진행한다.

))

**GQCS()완료통지**

**에러처리**

**If(transferred) {…}**

**If(Recv완료){}**

**WorkerThread - Recv**

완성된 메시지를 확인해서 Msg Packet당 OnRecv로 들어간다.

(한번 Recv완료통지가 왔을 경우 Recv링버퍼에 있는 모든 데이터를 반복문을 통해 받는다.)

우리는 라이브러리 – 컨텐츠 부분으로 파트를 나눴기 때문에 라이브러리쪽 헤더(2byte Len)은 빼고 OnRecv로 들어간다.

**WorkerThread – OnSend**

특이사항 없음.

**변경사항**

모든 동기화객체 제거

리스트-> 배열로 변경

라이브러리 코드를 클래스로 랩핑

이후 세션에 대한 동기화를 완전하게 잡은 후 별도의 인터락으로 뺄 것.

**세션에 대한 동기화 - SendPacket**

진행하는 프로젝트는 에코서버이므로, 세션에 대한 동기화가 완벽하게 잡히지 않은 상태이다.

SendPacket(SessionID, \*Packet)

**SendPacket()은 컨텐츠 쪽에서 네트워크 라이브러리쪽으로 요청하는 함수이기 때문에, SessionID를 전달하면 FindSession으로 Session을 찾아내야 한다.**

**Copy오버헤드 없애기**

**TCP스택 / 소켓 송신버퍼 / SendQ / CMsg**

🡨----방향으로 복사가 이뤄진다.

**CMsg(직렬화버퍼)의 주소를 링버퍼에 저장하고, 소켓 송신버퍼를 0으로 만들어 CMsg포인터를 즉시 TCP스택으로 복사할것이다.**

결과적으로 Copy의 행위를 두번 줄이는 것.

**이때 송신버퍼를 0으로 만들어 비동기를 유도했으므로, WSANOBUF에 들어있는 직렬화버퍼의 버퍼들은 페이지 페이지 락이 걸리게 될것이다.**

**보통은 페이지락에 대한 문제로 WSAENOBUFS가 나오는 경우는 거의없을것이나, 반드시 염두해두고 있어야 한다.**

**Copy를 줄이고 얻는 장단점**

Copy를 두번이나 생략시켰으므로, 분명히 성능향상이 존재한다.

그리고 거의 가능성이 없긴하지만, 페이지 락이 걸려서 WSAENOBUF의 위험성이 내제되어있다.

**CPU사용률에 대해**

이와별개로, CPU입장에서는

**1. CPU사용률을 단순히 떨어뜨리는것**

**2. CPU를 타이트하게 사용하여 사용률을 높이고 CPU사용시간을 확보할 것**

이둘은 별개의 얘기다.

**특정 함수를 호출시켰는데 CPU사용률이 높고 호출이 빠른거랑, CPU사용률은 낮지만 내가 바로 처리할수 없는 경우 느낌은 다르다.**

무조건적으로 어떤 것이 좋다고 말하기에는 애매한상황.

**SendQ 변경하는 이유**

우리는 위와같이 Copy를 줄이기위해 직렬화버퍼 자체의 포인터 주소값을 넣게끔 구조를 변경할 것이다.

**이때 만약 Send를 1회로 제한하지않는 구조라면, 굳이 SendQ를 변경할 이유가없다.**

**보낼게 있을때마다 WSABUF로 셋팅하여 쏴버리면 끝이기 때문.**

하지만 직렬화버퍼 자체를 WSABUF에 셋팅하여 보내는경우,

완료통지가 오고 난 후에 직렬화버퍼를 해제해야 할것이다.

**따라서 이경우에도 Send를 1회로 제한하는 것이 성능하락이 없을 것이다.**

**SendQ 변경**

이제 SendQ에 직렬화버퍼 자체의 포인터(주소값)를 넣게끔 구조를 변경한다.

(일단은 현재 만들어놓은 링버퍼로 진행하고, 이후에는 Rock-Free Queue로 교체한다.이 경우는 포인터를 만드는 Q자체가 될것이다. 템플릿이기 때문에 Q자체가 Q<CMsg\*>형태가 될것.)

**포인터 자료형 자체를 말하는 것이 아니라, 동적할당한 직렬화버퍼의 주소를 말한다.**

SessionID 검색

Session[N] 얻음

세션[N] SendQ.Enque(&Pakcet, 8);

(참고로 서버는 무조건 64비트로 간다. 32비트 서버는 더 이상 나오지않는다)

**SendPost() - WSABUF**

이제 SendQ안에는 여기저기 널려있는 직렬화버퍼 자체의 포인터가 하나씩 들어있는 상태.

이제 Direct Inqueue/Dequeue는 필요없으므로 사용하지않는다.

**WSABUF는 직렬화버퍼의 주소를 넣는것이아니라, 직렬화버퍼의 버퍼포인터를 넣는다.**

**WSABUF 개수는 (UseSize / 8), (SendQ.Size / 8), 100, 500, 1000 등 다양하게 잡을수있음.**

**어차피 지역변수로서 사용하기 때문에 상관없다.**

버퍼 하나하나 마다 WSABUF를 하나씩 쓸것이므로(직렬화버퍼는 연결된 메모리가 아니기때문)

**만약 SendQ에 직렬화버퍼가 백개 꽂혀있다면 백개에 대한 셋팅을해서 한방에 쏠것이다.**

**WSASend가 호출될 때 WSABuf가 늘어나는 만큼 호출리턴은 느려질 것이나, WSABUF의 개수와 비례하여 시간이 증가하는 것은 아니기 때문에 많이 넣을수록 이득이다.**

단, WSABUF가 몇백개씩 쌓이는 케이스는 없음. 억지로 스트레스 테스트를 하지않는이상.

(현재 에코 예제 같은 경우 1,200개씩 보내니까 최대1,200개가 쌓일 가능성이 생긴다.)

이제 반복문 돌면서 SendQ에 직렬화버퍼 셋팅

(직렬화버퍼 버퍼포인터와, 해당 버퍼의 길이셋팅)

**For()**

**\*Packet = SendQ…**

**WSABUF[n].buf = packet->ptr**

**WSABUF[n].Len = packet->GetUseSize();**

이렇게 차곡차곡 쌓은 WSABUF를 WSASend로 한방에 쏴준다.

**직렬화버퍼(Msg) 유지**

현재 우리는 Send시에,

**직렬화버퍼 셋팅 -> 링버퍼 Inqueue -> 링버퍼 Dequeue (WSABUF셋팅) -> WSASend(내부 송신버퍼에 COPY)**

위와 같은 과정을 거치고있다.

이때 Memcpy를 줄이기위해

**직렬화버퍼 셋팅 -> 직렬화버퍼 포인터를 링버퍼에 Inqueue -> 직렬화버퍼 개수만큼 WSABUF셋팅 -> WSASend**

(이후 송신버퍼까지 0으로 만들어, 프로토콜 스택까지 다이렉트로 복사하는 것을 목표로한다.)

**이 경우는 직렬화버퍼가 송신버퍼 역할을 담당하게 되므로, 직렬화버퍼는 유지되어야한다.**

**따라서 우리가 만든 메모리풀(FreeList)을 사용하여 유지시킬것이다.**

**메모리풀을 적용해보기전에, 직렬화버퍼를 동적할당하고 완료통지가 왔을떄 해당 직렬화버퍼를 지우는 방식으로 잘 작동되는지 반드시 먼저 확인 할 것.**

((

단, 지금 예제에서는 성능향상을 기대하기 힘들다.

**우리 에코프로젝트의 프로토콜은 10byte이므로, 인큐/디큐시 10byte가 Copy된다.**

**이때 우리가 개선시킨 형태는 직렬화버퍼의 포인터를 넣는 형식이므로 (8byte) 큰 차이가 없음**

실제 게임입장에서 본다면 크게 이득 볼수있는 상황이다.

))

**중간정리**

현재 완성한 구조는

**WSASend시 SendQ에 있던 데이터를 byte단위로 보내고,**

**완료통지시 받은만큼(transferred) RecvQ의 Front를 이동시켰다.**

이제 개선시킬 구조는,

**WSASend시 직렬화버퍼 자체를 던졌다.**

**Send에 대한 완료통지가 왔을 때, 동적할당된 직렬화 버퍼를 지워야한다.**

**이때 완료통지 쪽에서는 직렬화버퍼를 몇 개를 지워야 할지(해제) 알 수 없으므로, 직렬화 버퍼를 몇 개 보냈는지 크기를 알수 있어야 한다.**

애초에 에코서버이므로 transferred size가 의미없었지만, 개선된 구조에서는 더더욱 의미없어졌음.

(Transferred는 통신량을 구할 때 사용)

**따라서 Session마다 Send요청에 들어간 버퍼 개수가 저장되고, 완료통지 왔을 때 이를 참조하여 해당 개수만큼 뽑아낼 것이다.**

((

재언급

만약 직렬화버퍼를 100개 보낸다면, 반드시 요청한만큼 완료통지가 올것이다.

단, 한가지 예외로 도중에 끊긴경우는 우리가 요청한 크기보다 작은 값이 올 수도 있음.

))

Send에 대한 완료통지가 왔다면, SnedQ내부에 동적할당된 직렬화버퍼 n개를 꺼내 해제해줘야할 것이다.

**이를 해제하기위해서는 동적할당 한 포인터를 기억해 놓아야 한다.**

**1. 패킷포인터의 배열을 만들어 개수만큼 뽑기**

**2. peek()를 통해 front를 변경해가며 뽑기**

**1번방식의 경우 배열로 잡고 미리 반복문을 돌기전에 한번에 뽑아놓는 방법.**

지금 상황에서는 사용할수 있겠지만, 락프리로 넘어가서 SendQ를 리스트 형태로 바꾸게된다면 적용할 수 없게된다.

**2번 방식의 경우 기존 링버퍼의 peek함수를 변경.**

우리가 지금껏 사용하던 링버퍼가 기존의 링버퍼의 개념과 부합하므로, 수정하는것은 권장하지않는다.

(지금 시도하는방법은 링버퍼를 링버퍼가 아닌것처럼 사용하려고 하기 때문에 일어나는문제)

또한 락프리구조로 들어가 SendQ가 리스트형태로 바뀌게된다면, Peek를 할경우 차례대로 next를 타고들어가서 peek해야할 것이다.

(next-> next-> next-> … 굉장히 비효율적인 방식.)

**또한 Peek하여 별도의 공간에 복사해놨다가, 완료통지 왔을 때 Deque하는 방식은 비효율적이다.**

**3. 따라서 그냥 Deque하여 보관하고, 완료통지가 왔을 때 메모리해제한다.**

Peek역시 Deque와 비슷하게 무겁기 때문에 성능을 떨어뜨린다.

**정리**

**직렬화버퍼를 동적할당하여 SendQ에 복사**

**WSABUF에 직렬화버퍼 개수만큼 세팅**

**세팅한 직렬화버퍼 포인터들은 따로 보관**

**WSASend()**

**Send완료통지가 왔을 때 보관해놓은 직렬화버퍼 포인터를 해제.**

**추가언급**

우리가 현재 동기화객체를 넣는 이유는 SendPacket이라는게 전혀다른 곳에서 호출되는것에 대비하는 것이다.

아직까지 이런 상황은 나오지않았으므로, 아직 해결하지 못한 것.

**현재 IOCount를 잡고 0이되는순간 Release하기 때문에 IO가 일어나는 상황에서는 끊기지않는다. (따라서** **SendQ에도 Lock을 걸지않는다.)**

**배열에 비어있는 세션자리를 찾아내기 위한 스택자료구조를 사용한다. 이때는 동기화객체를 사용해야 할 것이나, 이를제외한 모든곳에서의 동기화객체는 제거한다.**

**동적할당 BroadCasting (예시)**

특정 컨텐츠에서 패킷을 만들었다고 가정한다.

Cpacket\* Packet = 할당.

Packet-> … 데이터를 셋팅.

SendBroadCast(…,Pacekt);

해당 패킷을 접속한 모든 클라한테 보내야 한다면?

**멤버에 동적할당한 참조카운트가 존재하고, 해당 참조카운트가 0이 될 때 지워지는 스마트포인터 구조가 되어야 한다.**

**따라서 어떤 세션SendQ에 들어갈때마다 참조카운트가 증가하고,**

**완료통지가 올때마다 참조카운트를 차감하여**

**최종적으로 0일 때 해당 메모리가 해제될 것이다.**

(SendQ에 넣었지만 Send할 상황이 아니라서 하지 못하는 경우도 존재할 것. 어쨌든 SendQ에 포인터가 들어간것이므로, 이 부분까지 추가해야 할것.)

SendPacket(SessionID, \*Pacekt)

{

}

**스마트포인터 Count증가/감소 방식**

SmartPtr<int>ptr = new int;

**스마트 포인터는 만들어지는순간 1이된다.**

SendPacket(SmartPtr<int>ptr);

**이렇게 다른 함수에 던진다고 한다면, 인자타입 역시 스마트포인터가 되어야 한다.**

해당 함수로 복사되면서 Count가 증가할 것.

이후 모두 사용하고 함수가 종료되면 소멸자가 호출되면서 Count가 차감된다.

**스마트 포인터 방식으로 Session요소 동기화? (복습)**

**Session의 경우 우리손에서 완전히 떠나는 경우가 생기는데, 스마트포인터 방식은 RefCount가 0이되면 삭제된다.**

**따라서 어딘가에서는 반드시 물고있어야 스마트포인터 방식을 적용할 수 있다.**

**직렬화버퍼 자체를 스마트포인터로 사용?**

스마트포인터 클래스는 껍데기일뿐이고, 실지 내부에 있는 포인터를 카운팅한다.

**하지만 CMSg(직렬화버퍼)의 경우 본인이 참조카운트의 대상이 되므로 CMsg가 사라진다면 그대로 삭제되어 버릴것이다.**

이를 자동화 시키고 싶다면 CMsg를 랩핑하는 클래스가 나와야한다.

**따라서 우리는 스마트포인터 방식을 채택하여 수동으로 올리는 형태를 취할 것이다.**

**스마트 포인터 방식을 적용하여 BroadCast();**

**BroadCast할 직렬화버퍼를 스마트 포인터방식으로 적용하여,**

**직렬화버퍼를 던질때마다 Count를 증가시키겠다는 얘기가 된다.**

이경우 SendQ에 넣는 행위자체가 하나의 복사이므로 참조카운트가 증가되어야 하고,

따라서 SendQ는 매개변수로 스마트포인터 객체타입을 받아야 할것이다.

진짜 스마트 포인터를 쓴다고 치면은..우리가 뭔가 컨텐츠쪽에서 필요할 때.

SmrtPtr<Cpacket>Packet = new Cpacket; (+1)

Packet = …

SendPacket(SendPtr<Cpacket>ptr). (+1)

**환형큐(링버퍼) 에 이 스마트객체 포인터가 여러 개 들어있는 것.**

(단순히 Cpacket의 포인터를 전달하는 것은 내부적으로 엉망으로 돌아갈 것이다.)

**스마트포인터 객체를 링버퍼에서 Dequeue할때는 뽑아서 복사시키고, 소멸자가 호출되어야한다.**

**중간정리) 직렬화버퍼 자체를 WSASend에 전달하는 이유**

**우리의 목적은, 링버퍼를 거치면서 계속하여 Copy되는 것을 줄이고자 함이다.**

**(내용자체를 모두 링버퍼에 복사하는 것이 아니라, 직렬화버퍼의 포인터를 넣는 형태)**

**스마트포인터 방식의 오버헤드**

그런데 위와 같은 스마트포인터 방식을 취하고자 한다면..

스마트포인터가 생성될때마다 객체추가가 되므로 성능이 매우 떨어진다.

(최초로 동적할당 후 객체가 생성될때마다 포인터를 복사해줄 것)

**따라서 우리는 스마트포인터 방식이아닌, 참조 카운트 방식을 수동으로 하는 것을 채택한다.**

(왜 스마트 포인터를 사용하지않는지 설명할 수 있어야 하므로, 두방식 모두 구현은 할 것)

**수동 참조 카운트 방식**

**우리는 스마트포인터 방식을 본따 수동으로 참조 카운트를 둘 것이다.**

그렇다면 참조 카운트 증가 시점은?

((

만약 반복문으로 여러대상에게 Send하는데,

그사이 Send에 대한 완료통지가 도착한다면 지워질수 있지 않은가?

현재 쓰는쪽에서 붙잡고 있기 때문에 쓰는동안에는 안전할 것이다. (소멸자가 호출되지않음)

(참조카운터는 소멸자에 의해 감소되다가 0이될 때 해제하기 때문)

))

다른 큐에 넣는 순간 우리쪽을 떠나는 것이기 때문에, 넣는 즉시 카운터를 증가시켜야 한다.

**스마트포인터 방식을 채용한 수동차감**

addRef() - 참조카운트 증가.

SubRef() or free(); - 참조카운트 감소

스마트포인터 방식은 차감하는 동시에, 0이되는경우 알아서 소멸된다.

**따라서 free에서 0이되면 삭제해야 하는데, 본인이 본인을 지운다는 이 구조는 사실 좋은 모양새는 아니다.**

**생성/삭제 위치가 다르다는 것은 좋은 설계는 아님.**

{

CMsg\* msg = new CMsg; +(1) : RefCount

SendPacket(pSession, msg);

{

SendQ.Enque(&p, 8); +(2)  
 }

} -(1)

위와 같은 증감 현황을 우리가 수동으로 addRef/free를 통해 RefCount수치를 조정한다.

**RefCount 수동차감 시점**

**실지 스마트포인터의 경우는 단순히 복사될 때 증가되고, 참조하는 곳이 없을 때 감소된다.**

**이때 증감은 자동으로 이루어진다.**

**이와달리 우리의 수동차감 시점은 다른스레드에서 참조할수 있을 때 증가시키고,**

**참조하지않을 때 감소되어야 한다.**

**스마트포인터 방식 자동화 (RefCount증감의 자동화)**

만약 자동화를 해야한다면 어떻게 해야할지 살펴본다.

우리는 실제 프로젝트에는 수동화할것이나 왜 자동화하지않는지 정확하게 파악하고 넘어가야한다.

addRef();

SubRef() or free();

**결국 스마트 포인터 방식을 적용한다면, 0이 떨어질 때 해제될것이다.**

**따라서 SubRef()값을 확인하여 RefCount == 0 인경우 delete한다.**

**별도의 껍데기 클래스**

**만약 자동화방식을 적용한다면, 최종적으로는 할당/차감에대한 코드를 함수로 한번 랩핑한다.**

(나중에 메모리풀로 바꿀 것이므로, 지금은 지저분하더라도 그대로 진행)

CSmartMsg()클래스로 래핑한다고 가정하자.

CMsg\* p = new CMsg;

P2 = p;

P3 = p;

주소복사가 일어나므로, RefCount가 증가해야한다.

**하지만 래핑한 클래스와 Msg는 다른 클래스이므로, 대입연산자/복사생성자를 활용할 수 없는 상황이다.**

**따라서 CsmartMsg의 인자로 CMsg타입의 객체를 던져야 하는 상황이다.**

그리고 CSmartMsg내부에서는 알아서 RefCount가 증감되고, 우리입장에서는 이 껍데기 클래스를 생성/삭제해야 하는 상황이다.

그렇다면 CSmartMsg()의 생성자에서 CMsg를 받아 넣어야한다.

**CSmartMsg p = new CSmartMsg(CMsg);**

CSmartMsg는 내부적으로 CMsg\*를 멤버로 들고있어야 하겠다.

(위에문단 수정필요)

**복사되는 경우는?**

이때 SendPacket을 호출하고자 한다면,

**SendPacket(SessionID, CSmartMsg p)**

위와 같은 형태가 되어야 할것이다.

**CMsg를 CSmartMsg클래스로 랩핑하여 사용할 것이기 때문에, 이 상황에서는CMsg는 어디서도 사용되지않는다.**

**또한 인자로 전달될때는 복사되는것이기 때문에, 포인터나 래퍼런스타입이 아닌 객체 자체를 던져야 내부에서 자동적으로 RefCount가 증가 될 것이다.**

**따라서 SendPacket으로 들어오면서 RefCount는 증가되고, 전달한쪽에서는 스코프가 닫히면서 감소될 것이다.**

**내부 복사생성자 코드**

CSmartMsg(CSmartMsg &SmartMsg)

{

\_ptr = SmartMsg->ptr;

\_ptr->addRef();   
}

(복사 생성자의 문법이 래퍼런스를 받도록 한다.)

**SendQ.Enqueue(CSmartMsg);**

그럼 자동화 시킨 이 형태에서, 직렬화버퍼를 SendQ에 Enqueue하는 경우는 어떻게되나?

**SendQ에는 CSmartMsg의 주소가 Enque되므로, SendQ자체가 CMsgPtr자체의 Q가 되어야한다.**

만약 SendQ가 배열형태의 환형 큐라고 가정한다면,

CSmartPtr \_Data[1000];

위와같이 CSmartPtr 자체를 들고있는 형태이다.

**SendQ.Enque(p)**

**{  
 \_Data[rear] = p; (자동적으로 대입연산자 호출됨)**

**}**

**결국 SendQ내부에서 CSmartPtr를 받을떄마다 CSmartPtr의 복사생성자가 호출되도록 하는 것.**

**SendQ.Deque(&CSmartMsg);**

Deque의 경우는 큐에서 데이터를 빼고, 인자로 들어온 CSmartMsg에 복사해줘야 하는상황이다.

**Deque(CSmartPtr &pCnt)**

**{**

**pCnt = \_Data[front]; (대입연산자 호출)**

**\_Data[front++].~CpacketPtr(); (소멸자 호출하여 -1)**

**}**

\_Data는 처음부터 배열로 확보되어있으므로,

**Deque스코프가 끝난다고 해도 RefCount가 감소하지않는다.**

**이 경우 Deuqe인자로 전달한 것까지 RefCount가 2가되므로, 명시적으로 소멸자를 호출하여 1로 맞춰준다.**

**실제 STL내부**

실제 STL내부도 정확히 위와 같은 방식으로 돌아간다.

Vector<class> vector1;

**이 경우 class타입의 배열이 내부에서 생성되고,**

**Push하는 경우 대입연산자 호출이 일어나며,**

**pop하는 경우 내부에서 명시적으로 소멸자를 호출시킨다.**

우리는 STL을 사용할 때 항시 포인터를 사용했으므로 이러한 상황이 없었음.

**SendQ -> WSABUF**

실제로 SendQ에 넣은 직렬화버퍼 포인터들을 빼서 WSABUF에 직렬화버퍼의 버퍼를 셋팅할 때는 완료통지 왔을 때 해제되어야하므로 Deque는 사용되어서는 안될 것.

**따라서 peek()를 이용해야한다.**

Peek()와 Deque()는 인자로들어온 객체에 복사해주는것은 같으나,

내부데이터를 실제로 지우는 것에대한 유무가 다르다.

**스마트포인터 자동화방식의 단점**

**수동으로 하는경우 역시 CMsg의 포인터타입을 인자로 전달하여 Copy가 일어난다.**

**따라서 CSmartPtr에 대한 Copy를 오버헤드라고 보지는 않는다.**

**스마트포인터 자동화방식의 단점은, 생성소멸자의 호출이 빈번하다는 것이다.**

**수동으로 증감하는 경우 내가필요할때만 RefCount를 증감하겠지만,**

**자동으로 증감하는 경우 내부적으로 원하지않을때도 RefCount가 증감된다.**

**최적화 유무와 성능**

단독포인터가 아닌 스마트포인터를 권장하도록 가고있다.

**생성자/소멸자에 대한 오버헤드는 최적화를 키는경우 어느정도 해결된다.**

(C++, 모던C++모두 성능을 얘기할 때 최적화컴파일을 한다는 전제로 얘기한다.

**최적화 컴파일을 하지않는경우 어마어마한 성능하락이 있을 것.)**

**최적화를 끄고 Release로 진행하고 마지막으로 최적화를 킬 때 문제가 생기는 경우도 존재한다.**

멀티스레드 환경에서는 순서가 뒤바뀌기 때문에(락을 걸고 동기화한다면 문제X),

덤프가 남지않는다면 굉장히 난감하다.

**최적화 사용시 디버깅**

문제가 나고 난다음 최적화 컴파일을 끄고 테스트한다고해도 같은 문제가 난다는 보장되지 않으므로 디버깅이 굉장히 어려워지게 된다.

덤프가 남는 경우 CallStack은 확인할 수 있으나, 이것조차 inline처리가 되어버렸다면 정확한 파악이 불가능하다.

**이때는 단순히 어디쯤인가를 파악하여 volatile을 심어 파악해야한다.**

**현업에서의 최적화**

**현업에서는 최적화컴파일을 하는 쪽으로 기울어져있지만, 위와 같은 문제로 현업에서는 최적화 컴파일을 하지않고 가는 경우도 존재한다.**

디버그로 서비스하는 회사도 존재한다. 생각보다 엉망진창인 경우들이 많음.

디버그로 가는경우 STL의 성능은 1/100정도까지 떨어질 것이므로 사용이 불가능하다.

(엄청나게 많은 예외처리.)

STL을 사용한다 함은 최소한 Release모드로 가고, 최적화까지 추가되어야한다.

**Stateless와 C++**

모바일 쪽에서는 stateless를 모두 웹으로 진행했다.

Node.js, php… 등.

**C++개발자 입장에서 이는 성능포기가 매우 심하기때문에, C++로 stateless가 개발되기 시작했다. (stateless라는 개념이 필요한거지 웹서버가 필요한 것이 아니다.)**

stateless에서 웹서버를 선택한 이유는 개발이 편하고, 편하다보니 웹개발자가 싸기때문.

대신 어마어마한 성능저하를 안고간다. (천배이상)

이제는 아예 공식적으로 stateless에서 웹을 버리자고 발표된 상황이다.

**우리의 C++서버는 stateful개념이 적용되어, 데이터공유, 스레드간 동기화 이슈등의 문제 때문에 난해한 것이다.**

**stateless면 스레드마다 개별적으로(싱글스레드) 가기때문에 어려운게없음.**

일단은 웹언어만 버리는 추세이지만, 더 나아간다면 http까지 버릴 것이다.

(아직은 클라쪽에서 바이너리로 가는것에 거부감을 가지고있음)

이는 cpp로 http를 통신하는 서버를 만들겠다는 뜻으로, 결국 성능을 위해 예전으로 돌아가는 모양새다.

http는 어떤 플랫폼에서도 가능한 호환성을 가지고있으나 프로토콜이 너무 크기떄문.

**IOCP 1차 수정 정리.**

소스코드를 랜서버 클래스에 포함시킨다.

직렬화버퍼 포인터를 받는 SendQ로 작업하여,

직렬화버퍼의 포인터를 SendQ에 꼽아넣고

여러 개의 WSABUF에 담아 WSASend에 전달한다.

**<8.7>**

**헤더 관련**

우리는 컨텐츠 - 네트워크 라이브러리의 계층을 나눴으므로,

**직렬화버퍼에는 payload(8byte)만 들어있어야 하며, 헤더는 네트워크 라이브러리 안쪽에서 알아서 해결해야 한다.**

일단은 컨텐츠쪽에서 2byte헤더도 넣는것으로 진행한다. (이후 수정할 것.)

**아니면 처음부터 직렬화버퍼 내부 멤버버퍼를 경계로 나누거나 헤더길이(5)만큼 버퍼크기를 추가해도 상관없다.**

(우리는 최종적으로 5byte가 최종적인 헤더로 완성될 것이다.)

(현재 2byte길이 헤더는 LAN통신용 헤더이므로, 체크섬/암호화 같은 안전장치가 없다.)

이 경우, 직렬화버퍼에 <<(Enqueue)할 때 처음부터 헤더뒤의 구간을 건드린다.

클래스 내부에서 Setheader()와 같은 함수를 만들어, 헤더를 미리 꽂아두고 데이터를 받을 것.**이때 헤더의 길이가 다르므로(서버통신 2byte, 컨텐츠 5byte) 처음부터 Setheader()를 두개로 나누거나, 들어오는 인자 크기로 구분해도 상관없음.**

**SetHeader함수는 private으로 막되, LANServer에서 접근할 수 있게 friend를 선언한다.**

(friend선언을 하지않고 완전히 독립된 클래스로 간다면 성능을 낼수가 없음)

**네트워크 파트안에서 헤더의 동적할당 시점**

**우리는 지금 Copy를 줄이기위해 직렬화버퍼자체를 SendQ에 곧바로 꽂았지만, 보낸 직렬화버퍼를 보관하기위해 동적할당을 하고있으므로 또다시 new/delete에 대한 오버헤드가 이슈로 떠올랐다.**

**따라서 new/delete대신에 메모리풀을 사용하기위해, 직렬화버퍼 자체의 생성을 private으로 막고 내부 메모리풀로 사용하게끔 만들 것이다.**

이 경우 어디서든 부르고싶을 때 직렬화버퍼를 할당할 수 있으므로, 엄격하게 계층을 나눠 메모리의 생성위치를 살펴보자면

**컨텐츠 쪽에서는 보낼 payload만을 따로 하나의 패킷을 만들어 복사해서 넘기고, 내부에서 알아서 동적할당해서 처리하는 것이 맞을 수도있다.**

**하지만 이 경우 Copy라는 행위가 한번더 일어나게되고, 이는 큰 성능 저하로 이어진다.**

(직렬화버퍼는 굉장히 많이 호출될 것이기 때문)

**따라서 어디서 부르냐는 관점으로 보지않고 직렬화버퍼에 관리자로 보는것으로 넘어가자.**

**SendQ에 넣은 직렬화버퍼를 뽑는 여러가지방식**

**1. SendQ.peek()을 이용해 CMsg크기만큼 하나씩 뽑는 것.**

**2. SendQ에있는 모든 사이즈를 뽑고, 하나씩 세팅하는 것.**

**결국 우리는 템플릿타입의 list형태의 Q가 되어야 하므로, memcpy를 통해 한방에 Copy해오는 것이 불가능하다.**

이경우 peek은 내부적으로 어떻게 돌아갈 것인가?

**Peek(&p, pos);**

따라서 3번째 있는 데이터를 확인하고 싶다면,

**Peek(&p, 3);**

이경우 내부에서 3번째 데이터만큼 pos를 이동시키는것으로,

만약 100번째 데이터를 뽑고싶다면 0부터 100번째까지의 주소를 타고가야할 것이다.

따라서 peek이 아닌 다른 접근방식이 필요하다.

(현재는 링버퍼를 링버퍼의 용도에 맞지않게 억지로 쓰기 때문에 이런 문제가 발생)

차차 살펴볼 것…

**SessionID의 FindSession에 대한 오버헤드**

SessionID로 특정 Session을 찾아야할 때, SessionArray를 모두 순회해야하는 문제가 있다.

SessionID로 한번에 SessionArray의 Index를 찾고싶다면?

SessionID는 우리 서버가 켜져있을동안 유니크해야하므로, Index와 통일시킬수 없다.

**SessionID 자료형 영역분리**

Index / UniqueID(SessionID)

**2byte / 6byte 으로 분리하는 것이 바람직해 보인다.**

**(Index : 동접 65536(2byte)까지 표현가능 / SessionID : 300조 가까운 ID기록가능)**

Index를 하위에 넣는다면 10진수로 큰값이 나올것이고,

하위에 낳으면 그나마 보기좋은 값이 나올 것이다.

**따라서 SessionID의 생성 / 인덱스뽑기로 작동하는 매크로함수를 만들어 사용한다.**

Union을 사용..?

Union은 내부적으로 형변환하여 뱉어주나? 아니면 내부적으로 비트연산이 일어나나?

우리가 하기 귀찮은 것을 컴파일러가 대신해주는 것뿐이지 새로나온 것은 아니다.

6byte를 배열로 잡나?.. 캐릭터배열? 뭐 하고픈대로.

**SessionID 검사**

**현재 Session을 Array로 두고, Lock없이 관리하는 상태이다.**

**락이 없으므로 검색을 완료 했더라도 해당 요소가 우리가 찾는값이라는 보장은 되지않는다.**

**(현재 상황에서는 나오지않는다. 단, 외부(컨텐츠쪽) 다른스레드에서 접근을 한다면 나올수있는 상황.)**

우리는 WorkerThread에서 로직을 처리하고 가므로 상관없으나, 컨텐츠쪽에서 다른 스레드가 들어간다면 옛날에 썼던 SessionID를 사용할 수 도 있는 상황이다.

(마치 우리도 끊어진 소켓을 상대로 뭔가의 작업을 하는 실수를 하듯)

따라서 찾은애가 맞는지 검사까지 하고 내가 찾는 요소와 다른경우에는 return false로 나간다.

**컨텐츠쪽에서 실수하는 부분**

잘못된 SessionID로 뭔가를 요청하는 경우, (SendPacket, Disconnect) 서버쪽에서는 단순히 return false로 끝난다.

이는 컨텐츠 쪽에서 해결해야 할 문제이다.

**Error - LoginPacket**

현재

OnclientJoin() -> RecvPost() 순으로 들어간다.

**이 경우 OnClientJoin 부분에서 SendPacket을 호출하는경우,**

**IOCount를 혼자 1로올렸다가 다시 0으로 떨어뜨려 ReleaseSession까지 타는 경우가 존재한다.**

**현재 더미에서 Error - LoginPacket이 해당부분을 체크하고 있다.**

**일단 클라이언트가 연결이 됐다면 서버가 먼저 송신을 해야 한다.**

우리는 접속확인 역할을 하는 Packet을 보낼 것이다.

세션의 할당,해제 동기화 문제 테스트

-- 접속확인 패킷 ------------------------------

헤더(2) 길이

데이터(8) 0x7fffffffffffffff

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

의 값을 클라이언트 최초접속시 클라에게 1회 보냄.

CLanServer 의 OnClientJoin 부분에서 이 패킷을 바로 Send 한다.

정상적으로 오지않은경우 , LoginPacket Not Recv가 카운팅된다.

두번와도 문제삼을것.

# 테스트 부분.

- 제대로 된 접속확인 패킷이 오는가?

- LoginPacketDuplicate - 접속확인 패킷이 1회만 오는가? (2회 이상 수신시 오류)

- LoginPacketRecv : 3초 이내에 수신이 되는가 ? (3초이내에 오지 않으면 오류)

위 두 항목은 무조건 0이나와야 한다.

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

**위 상황은 재연결이 없는상황에서는 아무의미가 없으므로, 재연결을 체크하고 테스트할것.**

이제부터는 유효한Index를 모아놓는 스택을 제외하고는 어떤곳에서도 락이 들어가지않는다.

OnClientJoin()부분은 컨텐츠에서 채우는 부분이다.

그런데 서버에서 확인차 SendPacket을 한다?..

네트워크 라이브러리단의 일이 컨텐츠쪽으로 넘어온느낌…

그리고 OnClinetJoin내부에서 SendPacket하고, RecvPost를 걸러간다.

OnClientJoin() -{내부에서 SendPacket}

RecvPost()

근데이때 RecvPost걸러가기전에 완료통지 로직이 먼저 처리되버리면?

**지금까지 적용한 방식**

-------초기방식-------------------------

Cpacket Packet

Packet <<df df dfd;

SendPacket(&Packet);

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

-----1차 수정 (직렬화버퍼 자체를 전달)----------

Cpacket\* pPacket = new Cpacket; //이부분이 메모리풀이 된다. (ㄱ)

SendPacket(pPacket); (ㄴ)

CmemoryPool<Cpacket> g\_packetPool(0, false); **(??)**

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

//위 동적할당은 완료통지가 올 때 알아서 거기서 지우는 형태를 취한 것.

**메모리풀 안전장치**

메모리풀에서 동적할당한 메모리를 제공할 때, 잘못된 블록을 free로 전달하는 경우를 대비하여 우리가 제공하는 메모리뒤에 특정블럭을 붙였다.

만약 사용자가 free를 던지는 경우는 Critical한 에러이므로, 이를 허용해서는 안된다.

**메모리풀 직렬화버퍼의 문제**

**메모리풀은 무조건 속도위주로 가야한다.**

**따라서 직렬화버퍼의 경우에는 placement new 를 호출시켜서는 안될 것.**

우리는 메모리풀을 만들 때,

성능을 높이기위해 사용자가 생성자/소멸자의 호출여부를 결정할 수 있게끔 만들었다.

**1. 최초 생성시 1번만 호출하고, 할당할때는 호출하지않는다.**

**2. 생성/할당할때마다 생성자를 호출시킨다.**

**직렬화버퍼의 경우 Clear()의 호출로 간단하게 초기화하여 할당할 수있으므로, placement new를 호출하지않고 단순히 Clear만을 호출하고 간다.**

**하지만 플레이어, 몬스터, 아이템 던전 등에서는 초기화하는 코드가 많이 들어가기 때문에 placcement new를 호출하여 명시적으로 생성자를 호출시켜준다.**

앞서 우리가했던 직렬화 버퍼 구간이 이제 이렇게 바뀔것이다.

**( Cpacket\* pPacket = new Cpacket; )**

**-> ( Cpacket \*pPacket = g\_PacketPool.Alloc() )**

**pPacket->Clear();**

**1. 메모리풀로 직렬화버퍼를 할당받는다면, 이 직렬화버퍼는 누군가 쓰던 것일 것이므로 직렬화버퍼를 사용할때마다 Clear()를 호출해 줘야 한다.(번거로움)**

**2. 그리고 사용자가 메모리풀을 사용하지않고 new해버리는 상황을 막아야 한다.**

**메모리풀은 직렬화버퍼 안쪽으로 들어가며, 생성자는 private으로 막는다.**

**이제부터 직렬화버퍼는 무조건 메모리풀을 통해서만 할당이 가능하다.**

**메모리풀 직렬화버퍼의 구현**

**이제 전역에 메모리풀을 선언하고, 직렬화버퍼는 자기만의 메모리풀을 가진다.**

**직렬화버퍼안에 직렬화버퍼를 다루는 메모리풀을 넣는다. (static)**

(메모리풀은 건드리지않는다. 직렬화버퍼 클래스안쪽에 메모리풀을 static으로 넣자는 뜻)

**추가적으로, 현재는 락프리큐를 넣지않았기 때문에 alloc/Free시 Lock을 걸고 간다.**

**class Cpacket**

**{**

**static CmemoryPool<Cpacket> \_PacketPool;**

**static Cpacket \*Alloc()**

**{**

**p = \_PaketPool.Alloc();**

**p->Clear();**

**return p;  
}**

**static void Free(Cpacket \*p)**

**{**

**\_PacketPool.Free(p);  
}**

**void Free() <- ??**

**{**

**\_PaketPool.Free(p);  
}**

**}**

**참고사항**

**- 생성자를 막았기 때문에 CMsg선언이 바깥에서 선언이 불가능하므로, this가 필요없는 static을 통해 Alloc(), Free()를 호출하도록 했다.**

**- 추가로 클래스 내부의 멤버 스태틱의 초기화는 헤더파일이 아닌 CPP파일에 해야한다.**

**메모리풀 직렬화버퍼의 해제**

pPacekt = Cpacket::Alloc();

SendPacket(&Pacekt);

(1) Cpacket::Free(pPacket); (내부에서 (2)를 호출)

(2) pPacket->Free();

**우리는 이제 직렬화버퍼의 참조카운트 개념을 도입시킨다.**

**(실지 구현된 스마트 포인터의경우는 별도의 객체형태로 생성자/소멸자가 호출되는 오버헤드가 크기 때문에, 참조카운트를 수동으로 증가/차감 시키는 형태로 간다.)**

**직렬화버퍼의 참조카운트 방식 적용**

**이전 예제에서는, 직렬화버퍼를 메모리풀로 alloc하여 Send한다음, 완료통지가 오면 직렬화버퍼 메모리를 해제했다.**

**이제 참조카운트 방식을 적용한다면, 내가 할당받은 일은 내가 처리해야한다. (Count의 증감)**

pPacket = Cpaket::Alloc(); = 1

SendPacket(&Packet)

{

…Enqueue(&Packet); +1 = 2  
}

완료통지가 왔다면.. -1 = 1

별도의 워커스레드에서 완료통지가 온다면 Count를 차감한다.

**하지만 생성한쪽에서 +1이 되고 Enqueue로 들어가면서 Count가 2가되었기 때문에, 완료통지가 와서 -1하더라도 지워지지않는다. (1인상태)**

**따라서 내꺼에 대한 참조카운트 차감이 들어가야 한다.**

(스마터포인터는 누군가는 해당 포인터를 유효하게끔 유지하고 있어야 하므로, 처음부터 0으로 들고 있자는 것은 말이안됨)

**메모리풀 직렬화버퍼의 해제 구현**

pPacekt = Cpacket::Alloc();

SendPacket(&Pacekt);

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

(1) Cpacket::Free(pPacket);

(2) pPacket->Free();

직렬화버퍼에 참조카운트 방식을 도입함으로서, 이둘의 차이가 명확해졌다.

**참조 카운트는 직렬화버퍼 자체에 RefCount로 들어가서, 내가 0일 때 해제되어야 한다.**

**최종적으로 자기자신이 해제를 해야한다.**

1번의 경우는 전역함수이므로, 내부에서는 다시금 p->Free()가 된다.

static void Free(Cpacket \*p)

{

p->Free();  
}

void Free()

{

}

**::Alloc();과 짝을 맞춘다면(직관성) ::Free(pPacket)이 낫겠으나, 내부적으로 다시금 p->Free()가 호출되므로 (2)번으로 가는 것이 성능상 낫다.**

(static free와 일반free가 오버로딩 되는지는 저도 모르겠음. 확인 해 볼 것)

**RefCount 증감/차감 함수**

**void AddRef();**

래퍼런스 카운트 증가함수.

**void SubRef();**

차감은 곧 Free()로, 내가 다썼다는 개념이다.

**따라서 SubRef가 곧 Free()인 개념을로 사용해도 되고, Free()안에서 차감시켜도 된다.**

사용하는 쪽에서 SubRef()한뒤 0이되는 경우를 비교하여 Free()를 명시적으로 해도 큰 상관은 없으나, 이경우 바깥에서 비교해야 되므로 번거롭고 코드가 더러워진다.

**그렇다고 둘을 합쳐 Free()에서 알아서 차감하고 0일 때 알아서 해제되게 한다면, 함수이름이 직관적으로 떨어지지않음. AddSub() <-> Free()**

**::Alloc으로 할당할때는 참조카운트가 1로 셋팅하여 뱉어야 할 것. (지워지면 안되므로)**

**당연히 RefCount는 인터락으로 가야한다.**

**RefCount 증감/차감 시점**

SendPacket(&Pacekt)

{

Packet->addRef();

…Enqueue(&Packet); + 1 = 2;

}

스코프가 닫히면서 -1 = 1;

완료통 온 경우 -1 = 0;

pPacket->Free();

Cpacket::Free(pPacket); 것이다

큐에 Inqueue한다음 추가적으로 어떤 행동해서 전달이 되는 경우라면 쉽게 구현이될.

**하지만 우리구조는 SendQ에 데이터를 넣는순간 언제 Send될지 알 수 없으므로, 먼저 증가시켜놓고 큐잉할 것이다.** (던지고나서 증가 X)

**그리고 완료통지가 올 때 래퍼런스를 차감(InterlockedDecrement)되고,**

모두 성공하여 RefCount가 0으로 떨어진다면 실질적인 메모리풀이 반납된다.

For(…)

{

SendPacket(&Pacekt)

{

Packet->addRef();

세션.SendQ.Enqueue(&Packet); + 1 = 2;

}

완료통지… -1 = 1

}

pPacket->Free(); 또는 Cpacket::Free(pPacket);

반복문안에서 Enqueue될때마다 참조카운트가 1씩 증가된다.

**중간정리**

직렬화버퍼를 사용할때마다 Clear()해야하는 문제와,

직렬화버퍼는 이제 무조건 메모리풀로 할당되게끔 하기위해(new,지역선언 불가)

**직렬화버퍼의 생성자를 private으로 선언하여 생성을 막고 static Alloc()/Free()만을 사용하여 할당/해제를 하도록 한다.**

**(내부에서는 friend선언으로 접근)**

**직렬화버퍼가 static::메모리풀을 가진다.**

**메모리풀 사용으로 인한 성능향상 확인**

메모리풀을 사용하는경우,

Alloc() 내부에서 안쓰는 메모리블럭을 할당받아 반환한다.

**만약 메모리블럭이 없다면 malloc으로 블록을 생성하고, placement new를 이용해 명시적으로 최초 생성자를 한번 호출시켜준다. (직렬화버퍼는 placement new = false이므로)**

현재 코드를 일반적인 new/delete로 호출시키고 싶다면?

**-------------------------------------------------------------**

**p = CAPcket::Alloc()**

**p->Clear();**

**-------------------------------------------------------------**

-> new p;

**-------------------------------------------------------------**

**CPacket::Free(p)**

**0 == p->SubRef();**

**-------------------------------------------------------------**

-> delete p;

위와같이 바꾼다.

또는, (CMsg::Alloc()의 호출이 마음에 들지않는다면)

**직렬화버퍼의 new를 연산자 오버로딩(opeartor new) 멤버로 가서 그 안에서 메모리풀로 할당받게끔한다.(이경우 바깥에서는 무조건 new로 들어온다.)**

**Delete 역시 오버로딩해서 free()또는 SubRef로 넘어간다.**

**(이렇게되면 외부에서는 new/delete로 생성/할당하므로, 메모리풀이 빠진다면 일반적인 new/delete로 바로 치환하여 사용해볼수 있겠다.)**

단점은 모양새가 같기 때문에 헷갈릴수있음.

저는 그냥 alloc/free로 갔습니다.

**성능향상 - 생성자 부분**

직렬화버퍼의 생성자는 최초 메모리풀 안에서 확보될때만 호출된다.

**따라서 이를 단순히 new/delete로 바꿔서 하게되면 테스트시 엄청난 성능차이가 날것이다.**

**(new/delete는 호출될때마다 생성/소멸자가 호출되므로)**

이건약간 사기에 가까운행위..

왜냐하면 애초에 우리 메모리풀은 생성자/소멸자를 호출하지않기 때문에 메모리 내부 버퍼에 동적할당 할게 없다.

근데 new/delete로하면 안에서 버퍼를 new/delete가 이중으로 들어가는 꼴이된다.(?)

**(메모리풀 new, 직렬화버퍼 new?)**

당연히 new/delete가 매우 느릴수밖에 없는 상황이고, 이런 경우라면 해당 직렬화버퍼에 대한 new/delete자체에 대한 테스트자체를 할 필요가없음.

우리가 만든게 성능이 안좋게 나올 이유가없기 때문..(내가 잘만든줄 착각함)

**생성자 문제**

static선언으로 인해 소속은 정해졌지만, 문법상 전역으로 들어가게된다.

**실제로 직렬화버퍼를 사용한다고 한다면, 실제로 CPP쪽에 전역으로 선언되어야 한다.**

CMemoryPool<Cpacket>Cpacket::\_PacketPool(1000000,false);

(이와 같이 처음부터 100만개를 만들어 두고 갈수도있고, FreeList방식으로 갈 수도있다.)

**현재 우리는 네트워크라이브러리 계층을 짜고있는 것이므로, 어떻게 갈것인지는 서버에따라 결정나야 한다.**

얼마만큼 쓸지 용량이 예측되지않으므로 거의 대부분 FreeList로 가긴 할것이다.

((

직렬화버퍼는 평소에는 열댓개 유지된다.

**서버에 부담이 심해지는 경우 직렬화버퍼가 백만개 이상을 사용할 때도 있다.**

**(완료통지 왔을떄 직렬화버퍼를 해제하기 때문.)**

백만개를 사용하고 다시금 열댓개를 유지한다고 하더라도, 다시금 백만개로 치고올라갈 수있다.

(이와 같은 이유와, 만들기 너무 난해한 이유 2가지 때문에 가비지컬렉터는 넣지않았으나 하고싶다면 해보는것도 상관없음)

))

**해결책**

**1. 바깥에서는 함수를 통해 인자를 전달하고, 내부에서는 해당 인자의 크기만큼 특정포인터를 통해 동적할당한다.**

**2. 싱글톤처럼 ::Alloc함수안에 내장시킨다.**

내부에서 메모리풀이 NULL인경우 알아서 디폴트로 생성하고 가게끔 한다.

(1)번 방식의 문제점.

**포인터를들고 명시적으로 초기화시키는 함수를 호출시켜야 하므로, 사용차즉에서 누락할 가능성이 존재한다.**

**싱글톤을 사용하는 이유**

**1. 특정 객체(변수)를 유니크한 존재로 만들어 사용하고 싶을 때. (하나만사용)**

**2. 생성의 시점을 명시하기 위함.**

**Static의 경우 전역변수와 같이 취급되므로 프로그램이 시작될때부터 메모리를 사용한다.**

**포인터를들고 최초 호출시 동적할당하는 방식의경우, 사용할때 메모리가 할당된다.**

**멀티스레딩 환경에서의 싱글톤**

**싱글톤 구현방식**

1. GetInstance(전역함수) 내에 static 지역변수를 두는 방법.

2. 클래스 멤버 변수에 static변수로 자기 자신을 두는 방법

3. GetInstance에서 동적 할당을 하는 방법

4. 템플릿 싱글톤 클래스를 만들어서 상속받아 쓰는 방법

**이떄 지역변수와, 멤버의 포인터타입은 멀티스레드에서 안전성을 보장받지못한다.**

(static이 필요)

Static CPacket \*Alloc()

{

**if(\_PacketPool == NULL)**

**{**

**\_PacketPool = new CMemoryPool<Cpacket>;  
}**

p = \_PacketPool.Alloc();

p->Clear();

return p;  
}

싱글톤 객체를 호출 할때마다, 해당객체가 존재하는지 매번 if문으로 판단할 것이다.

**이때 지역변수, static지역변수,/ 또는 멤버의포인터인 경우 멀티스레딩 환경에서 안전하지않다.(??)**

**(동시에 치고 들어오는경우, 두개가 만들어 질 수 있다.)**

**인터락사용 - 멀티스레드 환경에서 포인터형태 싱글톤**

**포인터 형태로 객체를 두고 최초 호출하는 시점에 동적할당하는 싱글톤의 형태에서, 멀티스레드환경에서 안전하게 돌리기위해 인터락을 적용해 본다.**

인터락을 사용한다고 한다면 flag를 하나두고, 최초 생성인지 아닌지 판단하게 될 것이다.

**만약 최초접근시 인터락으로 변수부터 바꿔놓고 들어간다면, 내부에서 생성중인데 또다시 요청이 올 경우 nullptr이 반환되어버릴 수있다.**

**따라서 동적 인스턴스 싱글톤은 인터락으로 멀티스레딩환경에서 안전하게 구현하는 것이 불가능하다.**

**call\_once();**

C++11에서는 call\_once()라는 것이 추가되었다.

**멀티스레드 환경에서 안전하게 싱글톤을 사용하기위해 call\_once를 사용할 수 있다.**

**call\_once의 경우 Mutex헤더를 include해야 사용이 가능하며, 내부에서 SRWLock을 걸지만 재귀락을 커버하기위해 CriticalSection형태로 랩핑한 모양이다.**

**결국은 call\_once를 쓰든 동기화객체를 쓰든 똑 같은 상황.**

**따라서 싱글톤객체를 호출할때마다 인터락함수를 체크하는 코드가 들어가고, 이는 성능저하로 이어진다.**

**FreeList의 Alloc호출량**

**Alloc()의 경우, 함수자체는 가볍지만 어마어마한 호출량을 가질것이다.**

**만약 FreeList가 싱글톤으로 만들어져서, Alloc()이 호출될때마다 Interlock을 체크하는 코드가 들어간다면 엄청난 성능하락을 가져올것이다.**

**FreeList에서 Alloc증가량에 대해**

본인은 용량이 부족한 경우 하나의 Block식 Alloc했다. 많이잡아놔도 상관없음.

Vector와 같은 자료구조가 capacity를 크게 증가시키는 이유는, 하나씩 삭제가될때마다 기존것들을 다 옮겨야하는 문제가 존재하기 떄문이다.

따라서 copy의 횟수를 조절하기위해 1.5배 또는 2배씩 증가시킨다.

**싱글톤 - 결론**

CallOnce를 선호하지 않는 개발자의 경우 위와 같은 이유로 동기화객체의 동적생성을 좋게보지 않는다.

(단, 현업에서 CallOnce()를 사용하는 사람도 많다. 취향차이로 받아들일 것..)

**따라서 실제 싱글톤으로 만들지만 내부의 객체생성을 static으로 명시하거나, 바깥에서 명시적으로 호출한다음 돌아가도록 있게끔 구현한다.**

(싱글톤 클래스지만 main같은 곳에서 한번 호출하라는 얘기. 이경우 사용자가 실수할 가능성이 생기는 것.)

**서버입장에서는 어차피 사용하는경우 비용을 치르고(메모리를 잡아두고) 가는것이 좋지만, 클라의 경우는 필요할 때 쓰고 필요없으면 지우는 것을 기본으로 간다.**

**동기화 문제**

이제부터 모든 직렬화버퍼의 생성과 해제를 메모리풀을 통해 가게끔 한다.

현재 메모리풀은 list-stack방식을 택했고, 멀티스레드환경에서는 당연히 문제가 난다.

**일단은 메모리풀 내부에 동기화객체를 내장시켜서 Lock을 걸고 진행한다. (Alloc/Free)**

이후 Lock-Free알고리즘을 도입하여 동기화문제를 해결한다.

현제 프로젝트에서는 에코서버이므로 잘 테스트가 되지않을 것이다.

(확인방법은 이전과 동일하다.)

**해당 내용까지 모두 완료가된다면 ZeroCopy까지되게해서 테스트를하고 나름대로 결론을 도출해볼 것.**

(결과가 도출되신분들은 공유)

**Garbage Collecter도입**

윈도우 내부에서도 Garbage Collerter 스레드가 계속 돌고있다.

**가비지컬렉터는 주기적으로 돌면서 물리메모리에 상주해 있는 것들 중 Page-Out이 될만한 것들을 판단하여 Page-Out시킨다.**

또한 가비지 컬렉터는 스레드우선순위 역전 문제 같은 것을 해결하기 위해,

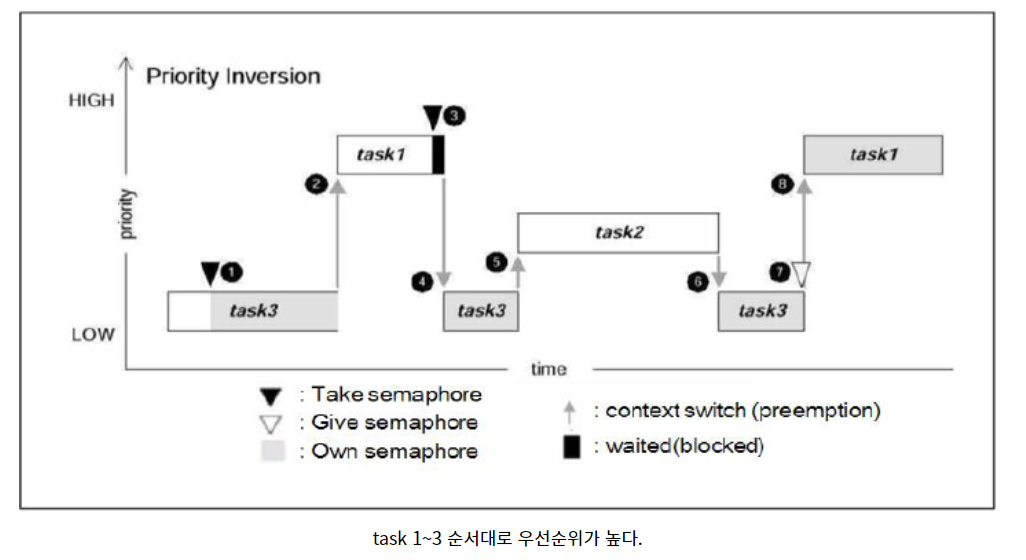
**현재 블락당한 스레드, 놀고있는 스레드, 대기중인데 Ready되지 못하는 스레드 등을 별도의 스레드가 계속해서 확인한다.**

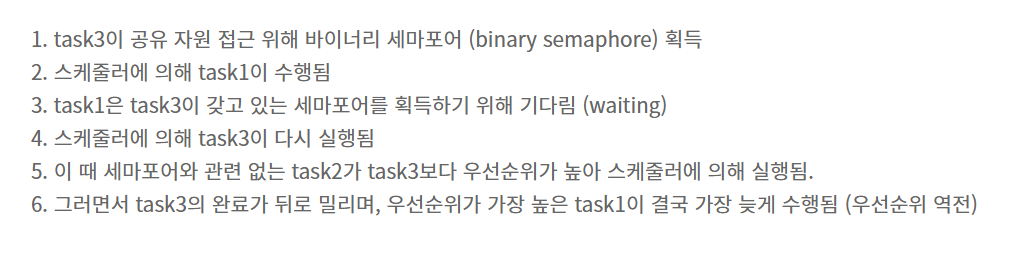
**이 같은 가비지 컬렉터의 기능을 성능저하없이 우리의 메모리풀에 구현한다면 매우 좋을 것.**

하지만 쓰지않는다고 해제하는 것은 메모리풀의 목적에 어긋나고,

구현이 매우 난해하기 때문에 대부분은 시도하지 않는다.

**스레드 우선순위 역전**





**이후 진행할 방향**

현재 프로젝트가 완료가 되면 이제 Lock-Free로 넘어간다.

Lock-Free가 완성된다면 해당 메모리풀의 동기화를 해결하러 온다.

그리고 락프리 내부에 동적할당 기능을 다시 메모리풀에 넣을 것.

(락프리로 만든 메모리풀을 락프리로 다시 넣는 것. 이렇게해서 락프리가 완성된다.)