**샘플 코드 프로젝트**

TCP 통신 기반의 IOCP 채팅 서버

목차

1. ServerSolution.sln 프로젝트 구성2

2. 실행방법2

2-1. ServerSolution 빌드와 결과물2

2-2. 서버 프로그램 시작3

2-3. TestClient 프로그램 시작3

2-4. TestClient의 기능3

3. 프로젝트 별 클래스 구성4

3-1. LowLib 프로젝트4

3-2. Server 프로젝트5

3-3. TestClient 프로젝트6

3-4. CommonFile6

4. 주요 기능 FlowChart7

4-1. 태스크 등록 및 동작 FlowChart7

4-2. 세션 연결 FlowChart8

4-3. 기본 로직 처리 FlowChart10

4-4. 멀티 스레드 IOCP와 MainProcess 처리 FlowChart12

4-5. Msg Read, Write14

5. 개선사항 15

**1. ServerSolution.sln 프로젝트 구성**

**- 작업 환경**

Visual Studio 2015 (v140)

**- LowLib :**

event\_select, iocp, msg 등 핵심 네트워크 라이브러리 로직 및 자료구조, iniReader, Log, Dump 등 시스템에 필요한 유틸 라이브러리 로직으로 구성

**- Server :**

Iocp로직을 이용한 서버 로직이 구현, session 클래스, session manager 클래스 등을 통해 접속된 접속 개체인

세션의 연결 관리를 담당, 프로토콜 처리를 간편하게 할 수 있는 Msg 클래스 및 Session의 Msg 핸들을 통해, 세션 별로 해당 프로토콜을 처리

**- TestClient :**

EventSelect로직을 이용한 테스트용 클라이언트 구현

MFC 기반 테스트 툴로, 서버와 클라이언트 간의 REQ요청 및 ACK처리의 테스트를 하기 위한 툴

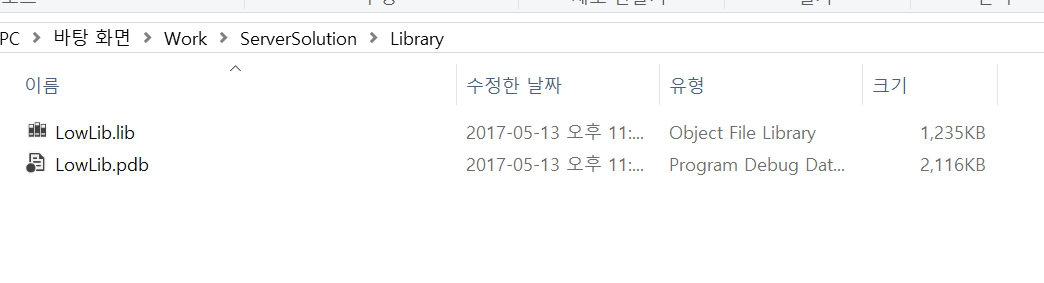
**2. 실행 방법**

2-1. ServerSolution.sln 빌드

빌드의 결과물들은 각각 별도의 빌드 결과 폴더에 존재

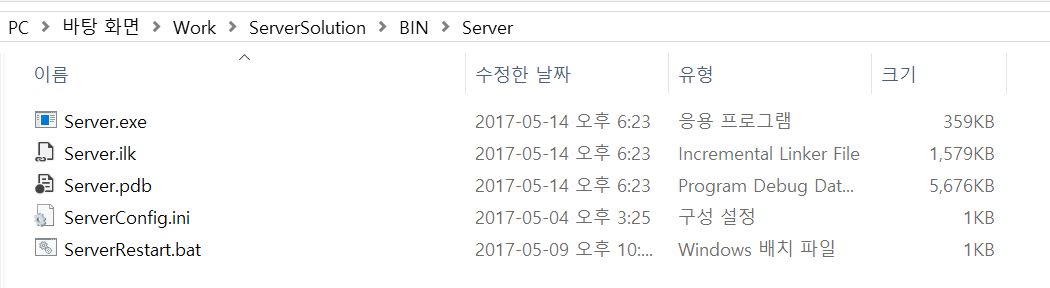
**- LowLib :**

./Library 폴더



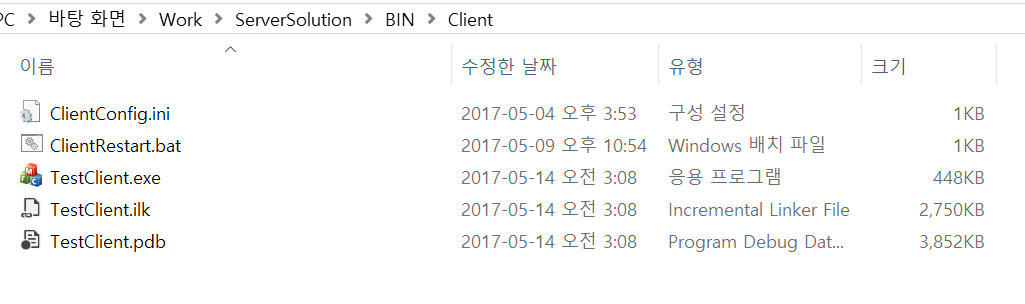
**- Server :**

./BIN/Server 폴더



**- TestClient :**

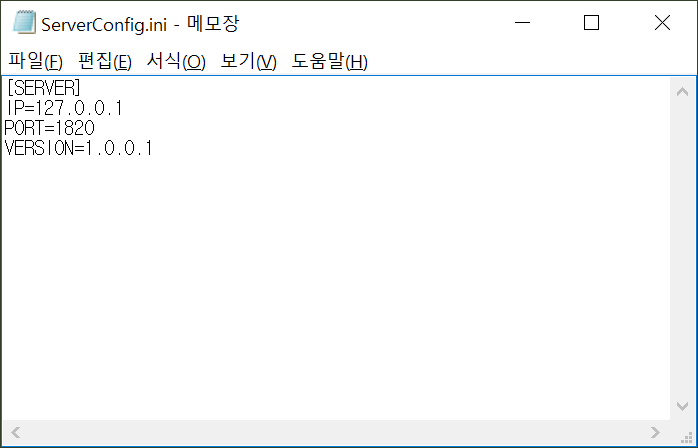
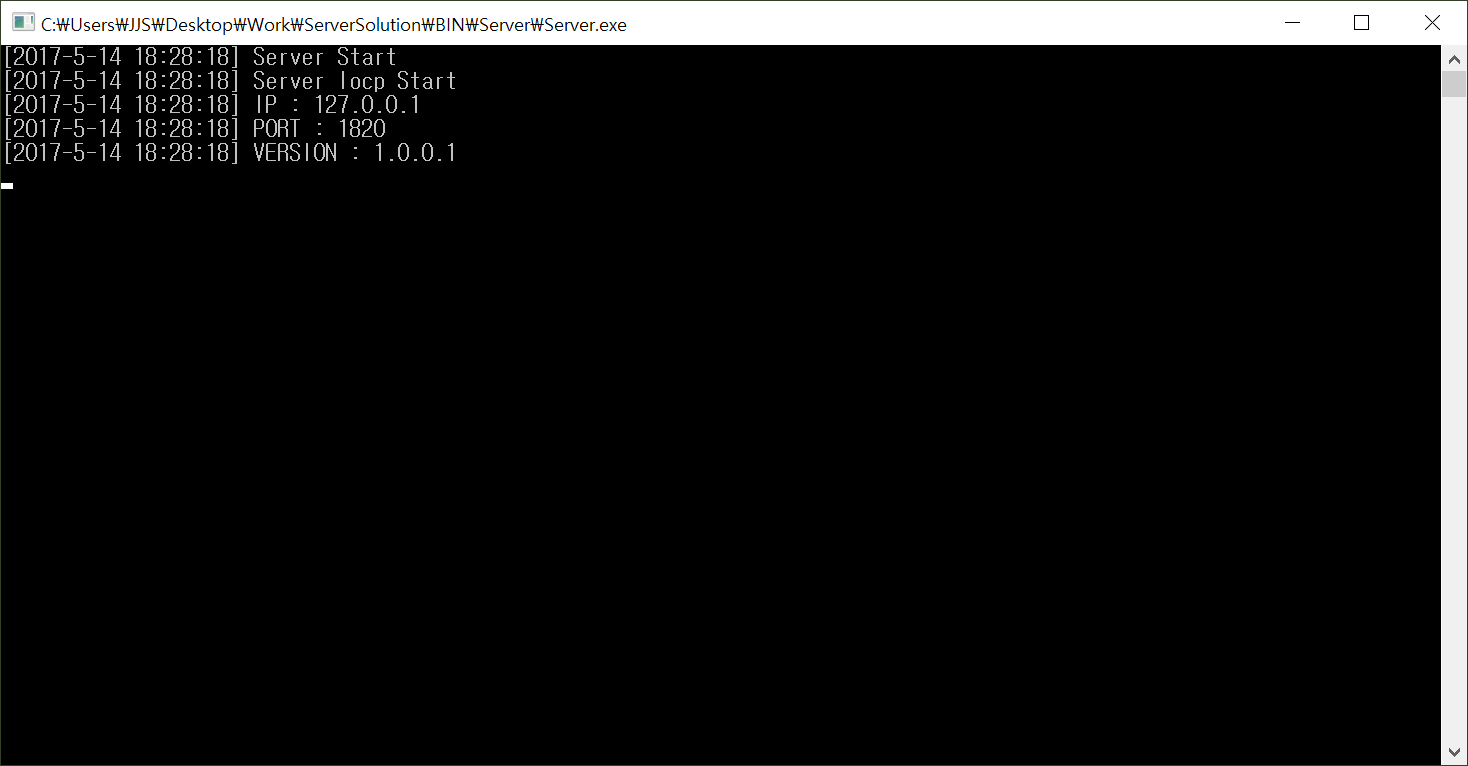
./BIN/Client 폴더



2-2. 서버 프로그램 시작

1) ./BIN/Server 폴더에 ServerConfig.ini 파일이 존재 하는지 확인

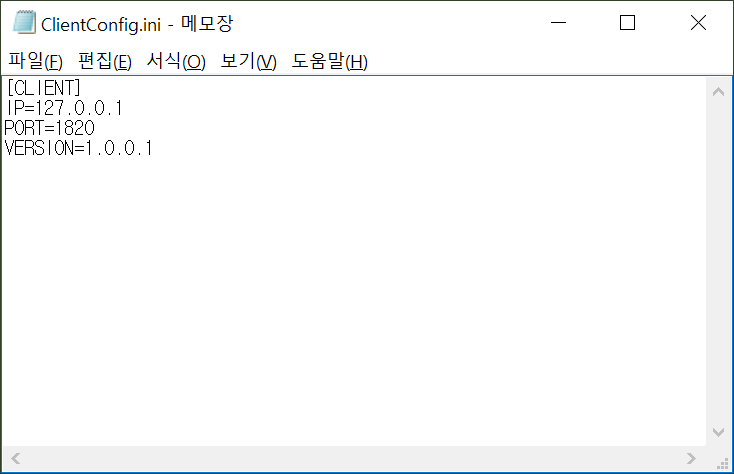
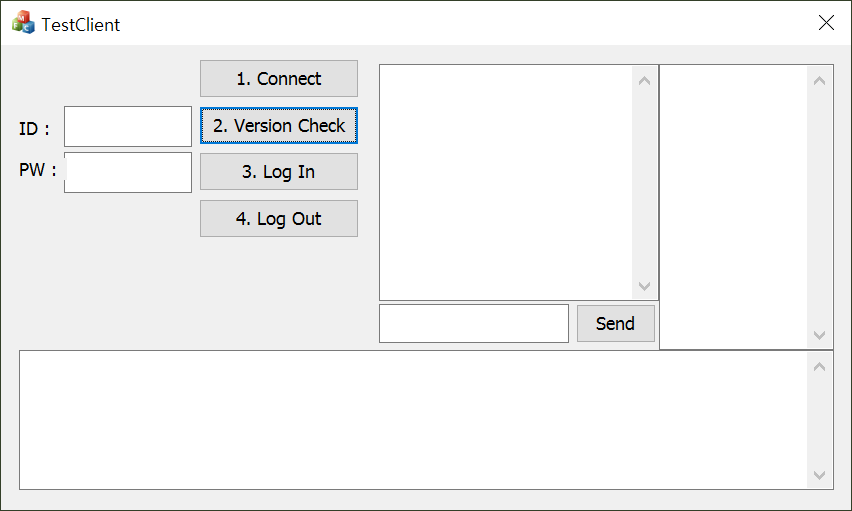
2) ServerRestart.bat 파일을 실행하여 서버를 시작



2-3. 테스트 클라이언트 프로그램 시작

1) ./BIN/Client 폴더에 ClientConfig.ini 파일이 존재하는지 확인

2) ClientRestart.bat 파일을 실행하여 클라이언트를 시작



2-4. TestClient의 기능

1)Connect : 클라이언트 통신 개체의 초기화 및 서버와의 Connect를 시도

2)Version Check : 클라이언트 버전의 유효성을 검증

3)LogIn : ID, PW를 받아 티켓을 생성, 티켓을 통해 인증이 완료되면 Session의 연결 등록 및 유저 포인터 생성, user\_id 발급 받는 기능 - 참고 : 한번 발급 받은 user\_id는 서버가 떠 있는 한(DB가 연동 되어 있지 않아 있기 때문) 유지 되며 동일한 ID로 접속 하는 경우, 이전의 user\_id를 그대로 발급 받는다.

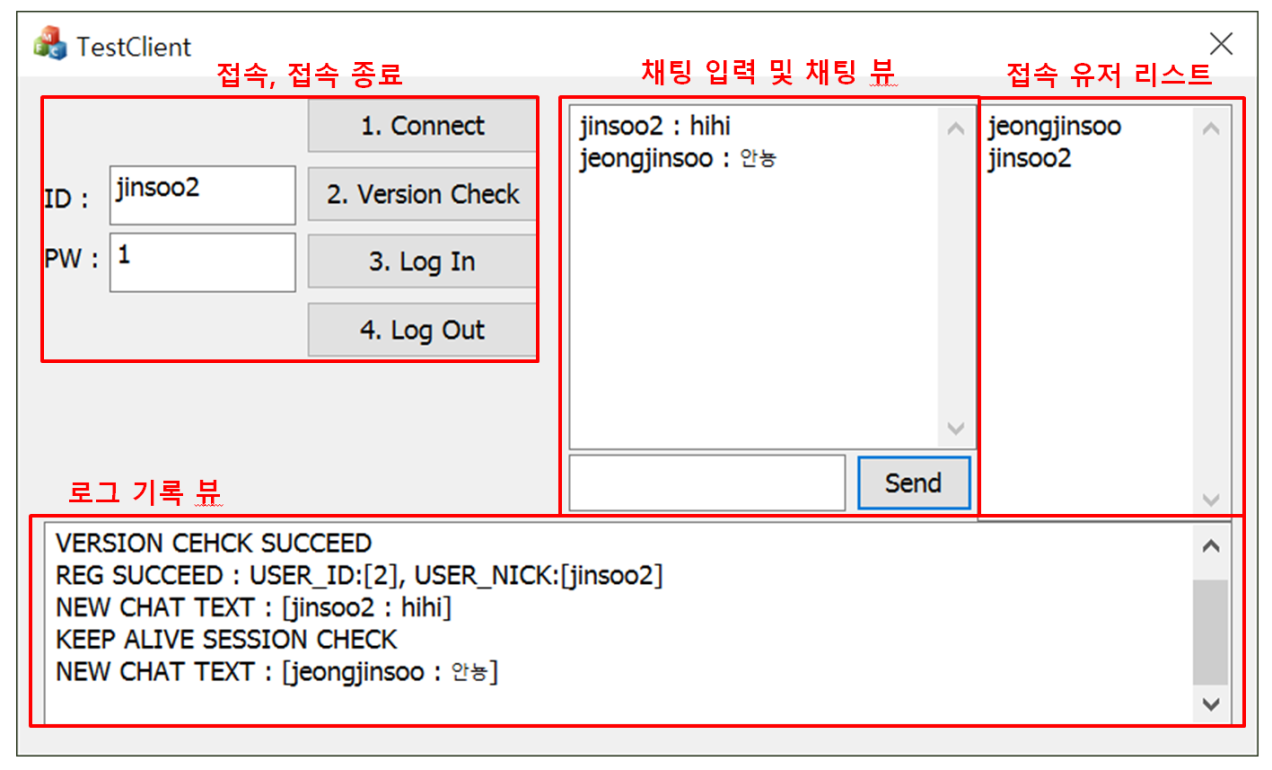
4)LogOut : 유저의 접속을 해제하는 기능

- 참고 : 세션의 연결은 끊지 않으므로, 재 접속 시에 다시 3. LogIn을 통해 접속을 시도할 수 있다

5) 접속 유저 리스트 : 현재 접속된 유저들의 리스트를 출력 (아래 이미지의 접속 인원은 2)

6) 채팅 입력 및 채팅 뷰 : 입력 및 채팅 기록을 볼 수 있는 뷰

7) 로그 기록 뷰 : 클라이언트에서 이루어진 행동의 로그들을 출력하는 뷰



**3. 프로젝트 별 클래스 구성**

3-1. LowLib 프로젝트

1) Network

|  |  |
| --- | --- |
| **클래스 이름** | **역할** |
| EventSelect | - 소켓의 IO관리를 담당하는 클래스  - 싱글 스레드 기반의 소켓 이벤트를 검사하는 구조  - 클라이언트에서 소켓 처리를 담당 |
| Iocp | - 소켓의 IO관리를 담당하는 클래스  - 다중 스레드 기반, IO의 발생을 담당 스레드가 처리  - 서버에서 소켓 처리를 담당 |
| Msg | - 전송될 버퍼에 필요한 데이터를 담고 조작하는 클래스  - 프로토콜의 ID, 전송될 버퍼로 구성 |
| SockSystem | - 소켓 통신 함수들을 랩핑한 클래스  - 소켓의 등록, 버퍼의 송신, 수신의 기능을 가집니다.  - TCP 기반의 소켓 송수신을 처리 |
| PacketSystem | - Msg의 송신을 원할 시 Msg 내부에 담겨진 데이터들로 실제 송신에 사용될 패킷으로 변환  - 수신된 패킷을 Msg 형태로 변환하여 반환 |
| Session | - 접속 개체의 통신을 관리하는 클래스  - Session의 접속 여부, Msg 송수신 요청을 담당 |
| SessionManager | - 접속 개체인 Session을 생성하고 관리하는 클래스  - 세션에게 Msg를 전송하는 SendMsg 및 Broadcast 등 전 세션을 대상으로 하는 작업 담당 |

2) Util

|  |  |
| --- | --- |
| **폴더 이름** | **역할** |
| DataStruct | - 템플릿 기반의 자료 구조를 구현  - Deque, LinkedList, CircleLinkedList, Queue, Stack 로 구성 |
| IniReader | - Ini 파일의 데이터를 추출, 세팅 할 수 있도록 랩핑한 클래스 |
| Log | - 서버에 출력될 로그 기록의 기능  - 서버에 남는 로그를 파일로도 만들어 저장 |
| Memory | - 메모리 관리에 관련된 클래스들이 존재  - AutodetectMemoryLeak : 메모리 누수를 체크하는 클래스, 오픈소스로 외부에서 참조  - MemoryPool : 메모리 new, delete 관리해주는 클래스, 자주 new, delete가 이루어지는 Msg, User의 메모리의 할당 해제를 관리 해 주기 위한 클래스, 오픈소스로 외부에서 참조 및 수정 |
| MiniDump | - UnHandled Exception으로 프로그램이 종료되었을 때, 덤프를 남기는 클래스  - MiniDUmpNormal 사용, 오픈소스로 외부에서 참조 및 수정 |
| Pattern | - 패턴 관련 클래스를 저장  - Singleton : 시스템 내에 단 하나의 클래스 객체를 가질 수 있도록 관리하는 클래스 |
| StringFunc | - std::string, std::wstring 관련 문자열 처리를 담당할 함수들의 모음 |
| ProcessObject | - Task에 등록되어 실제 작업이 수행될 개체 클래스  - Task에 등록될 클래스는 ProcessObject를 상속받아 Task를 생성하게 된다. |
| Task | - 다중 스레드 환경에서 처리될 작업을 Task 단위로 랩핑하여 관리  - ThreadManager를 통해 Task가 생성될 때 요청한 수만큼 스레드를 생성 |
| TaskManager | - Task를 생성 및 관리를 담당하는 클래스  - Content 로직에서는 TaskManager를 통해서 Task를 생성하고, 멀티 스레드 작업을 할당 |
| ThreadSync | - 정의된 동기화 객체를 통해, 스레드의 임계 영역 설정을 제어하는 클래스,  - 동기화 객체인 CriticalSection의 Lock, UnLock 을 처리 |
| Thread | - 스레드 관련 처리를 담당하는 클래스  - Task에 상태에 맞춰 등록된 ProcessObject의 Init, Dowork, Endwork를 호출 |
| ThreadManager | - Thread를 생성 및 관리를 담당하는 클래스  - Task로부터 생성요청을 받은 만큼 스레드를 생성 및 할당 |

3-2. Server 프로젝트

1) Content

|  |  |
| --- | --- |
| **클래스 이름** | **역할** |
| User | - 서버에 접속한 유저의 게임 내 정보를 관리하는 클래스 |
| UserManager | - 유저들을 관리하는 클래스, 같은 ID의 유저의 중복 접속을 방지, 접속한 유저를 찾는 등  - 전체 유저를 대상으로 하는 작업을 담당 |
| UserMsgHandler | - 접속된 User객체로 부터 Msg가 수신되면, 수신된 Msg의 프로토콜 ID, 버퍼의 데이터를 통해 알맞은 함수를 호출하고 수행할 수 있는 handler |

2) Main

|  |  |
| --- | --- |
| **클래스 이름** | **역할** |
| ChatServer | - 서버에서 사용될 IOCP의 기능을 상속 받은 클래스  - IOCP 연결 요청, 연결 종료 요청, 로직 처리를 MainProcess에게 요청하는 작업을 담당 |
| MainProcess | - 다중 스레드 환경으로 수신되는 메시지를 일괄처리하기 위한 메인 로직을 담당하는 클래스. |

3-3. TestClient 프로젝트

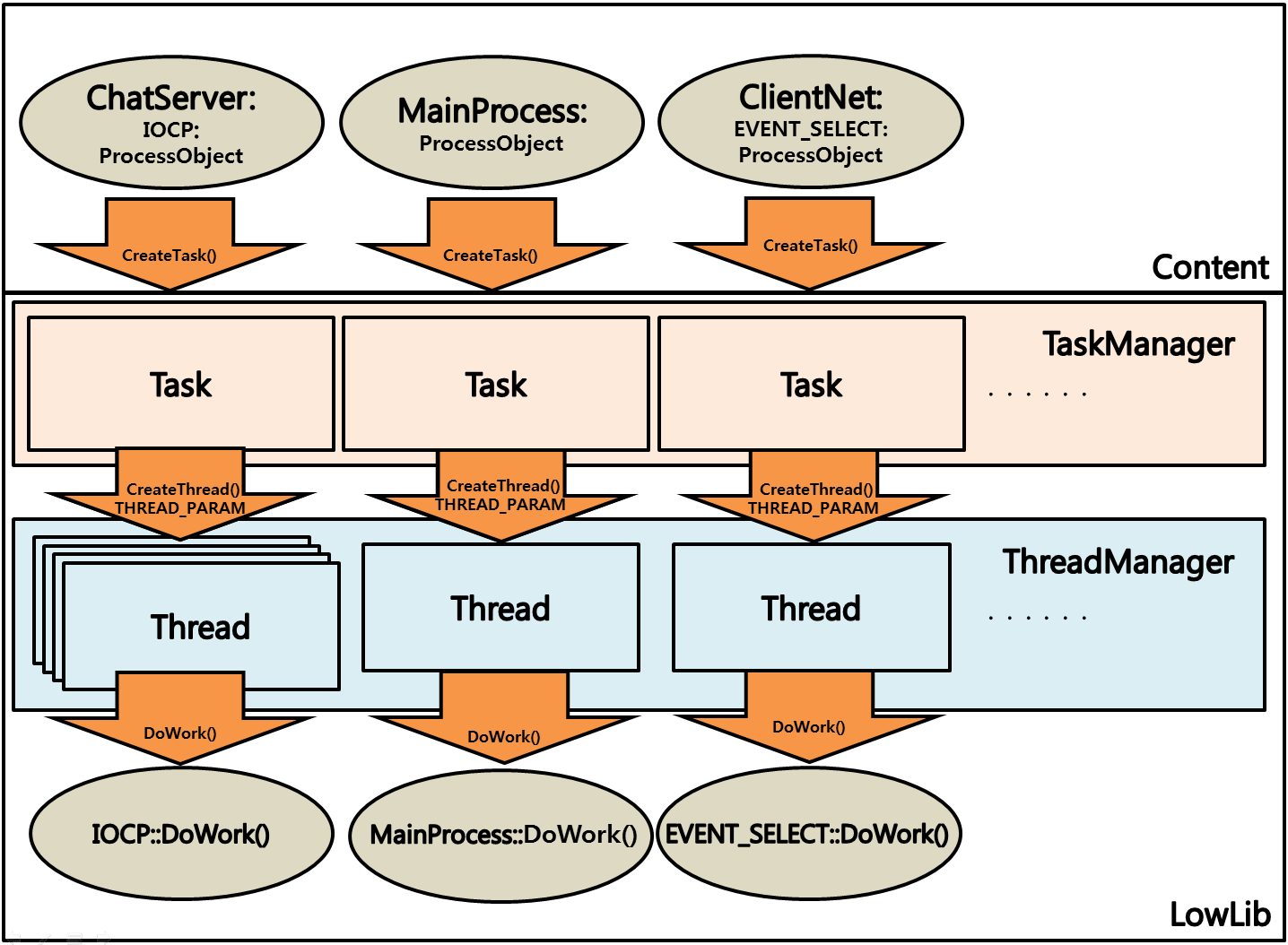
|  |  |
| --- | --- |
| **클래스 이름** | **역할** |
| ClientSession | - 접속 개체의 통신을 관리하는 클래스  - Msg의 송수신 및 수신 된 Msg의 핸들링을 담당 |
| ClientNet | - EventSelect를 상속 클라이언트의 IO처리를 담당  - ClientSession으로부터 Msg의 송신을 전달 받거나, ClientSession에게 수신된 Msg의 처리를 요청 |
| TestClientDlg | - TestClient 다이얼로그의 동작을 담당  - 텍스트를 로그 뷰에 입력하거나, 신규 접속 유저가 있다면 유저 리스트 뷰에 추가 하거나 하는 등의 UI 처리를 담당 |

3-4. CommonFile

|  |  |
| --- | --- |
| **파일 이름** | **역할** |
| Protocol | - 송수신된 프로토콜의 식별을 위한 ID enum 값을 관리  - REQ, ACK는 짝을 맞추어 존재해야 하며, 뒤에 아무것도 없는 경우 NOTIFY Msg로 간주 |
| PT\_ErrorCode | - Msg를 통해 처리된 작업의 결과를 error\_code 형식으로 전달  ex) PT\_ERROR\_NONE : 작업 성공, PT\_ERROR\_USER\_NOT\_FOUND : 유저를 찾을 수 없음 |
| PT\_Struct | - 수신된 Msg의 버퍼를 구조화 한 결과를 PT\_Format에 저장  - PT\_ReadMsg에 사용될 구조체 |
| PT\_ReadMsg | - 수신된 Msg로 부터 데이터를 구조화된 구조체(PT\_Struct/PT\_Format)에 담아 뽑아내는 함수  - ReadMsg만 있는 이유는 Msg에 데이터를 쓰는 과정은 작업자가 처리하기 쉬우나, Msg를 받아 데이터를 뽑는 과정에서는 어떤 자료형으로, 순서로 뽑는지 잘 알지 못한다. 때문에, ReadMsg에 추가된 함수를 이용하면 쉽게 데이터를 뽑아 다음 처리에 사용할 수 있음 |

**4. 주요 기능 FlowChart**

4-1. 태스크 등록 및 동작 FlowChart



태스크란 병렬로 처리될 작업(Job)을 포함하고 있는 개체를 뜻합니다. 사용자는 병렬로 처리될 객체에 한해

작업 객체(ProcessObject) 상속받은 클래스를 설계하고, TaskManager에게 태스크생성을 요청(CreateTask)해야 합니다. 태스크가 갖는 스레드의 개수는 CreateTask 요청 시 Parameter를 통해 전달된 값으로 결정 됩니다.

|  |
| --- |
| // ProcessObject를 상속 받은 IOCP 태스크 생성, 생성되는 스레드 0 = 물리코어 \* 2 개  if (TASK\_MANAGER.CreateTask(TASK\_IOCP, 0, &CHAT\_SERVER, 0, 0) == false)  return false;  // ProcessObject를 상속 받은 MainPorcess 태스크 생성, 생성되는 스레드 1 개  if (TASK\_MANAGER.CreateTask(TASK\_MAIN\_PROCESS, 1, &MAIN\_PROCESS, 0, 0) == false)  return false; |

TaskManager는 Singleton 패턴으로 설계되어 태스크의 생성 및 동작, 정리를 관리하게 되는 클래스이다.

CreateTask시 태스크는 ThreadManager를 통해 태스크에 할당될 스레드의 생성을 요청하게 된다.

이때, 작업 객체(ProcessObject)의 초기화 작업(Init())이 이루어 진다.

|  |
| --- |
| bool Task::Create(DWORD task\_id, DWORD create\_count, ProcessObject\* process\_object, DWORD stack\_size, int priority)  {  ...    if (THREAD\_MANAGER.AppendThread(create\_count, this, stack\_size, priority) == false)  return false;  is\_running\_ = true;  object\_->SetAllocThreadCount(create\_count);  object\_->Init();    return true;  } |

이때 생성된 스레드의 처음 스레드 상태는 대기(CREATE\_SUSPENDED) 상태로 생성된다. (이유는 한 태스크에 여러개의 스레드를 생성하는 경우 나머지 스레드 모두 생성될 때 까지 스레드 스케쥴링이 이루어 지지 않게 하기 위함이다.)

CreateTask의 동작이 완료 되면 StartTaskJob(TASK\_ID)를 통해 태스크를 동작 시킨다.

|  |
| --- |
| // IOCP 태스트 시작  if (TASK\_MANAGER.StartTaskJob(TASK\_IOCP) == false)  return false;  // MainProcess 태스트 시작  if (TASK\_MANAGER.StartTaskJob(TASK\_MAIN\_PROCESS) == false)  return false; |

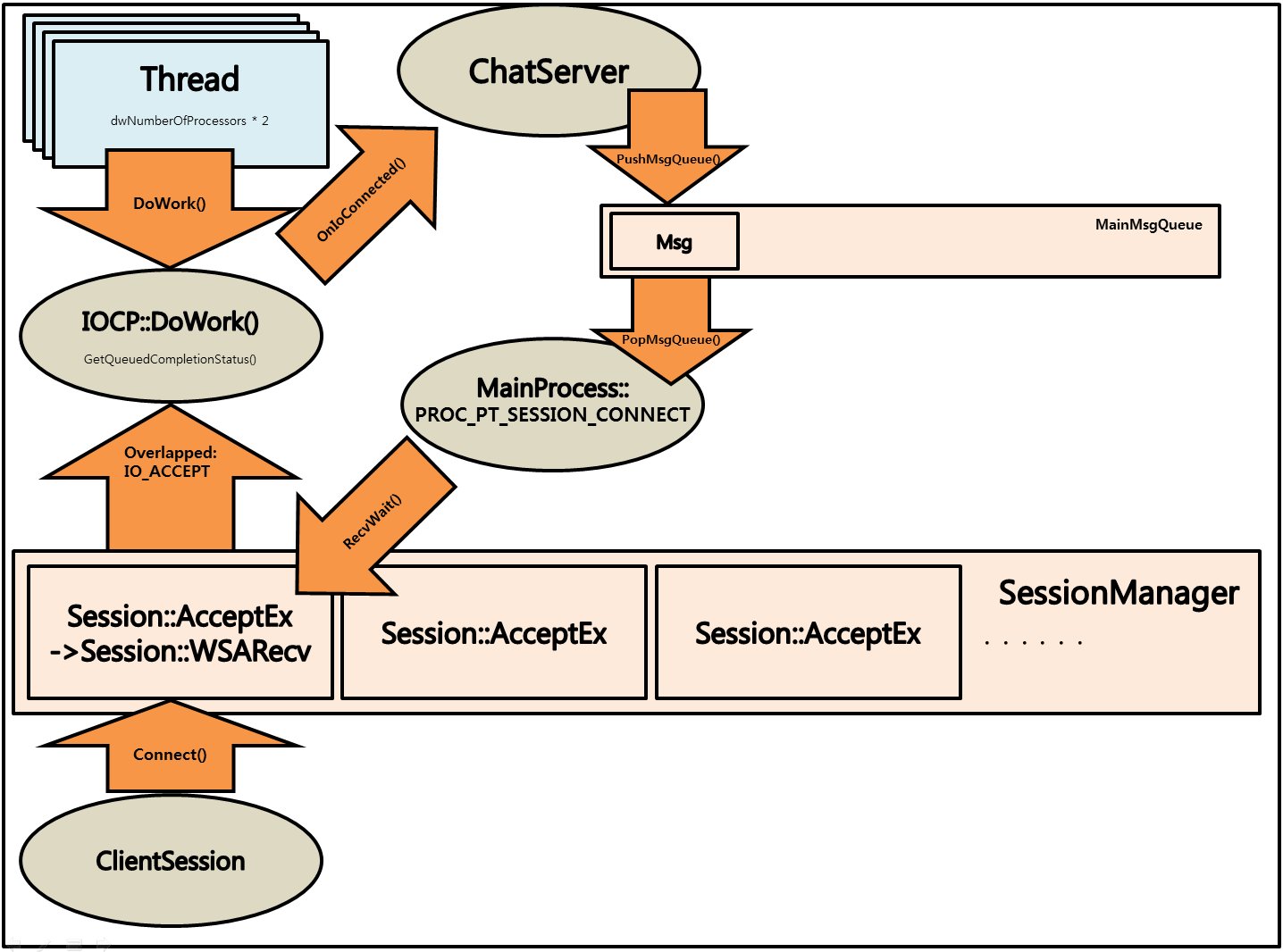
StartTaskJob를 수행하면 태스크 별로 맵핑 된 스레드는 태스크의 작업 객체(ProcessObject)의 DoWork()를 호출하며, 스레드는 Running 상태로 전환된다.

Server Solution 에서는 비동기 IO 처리를 담당하는 IOCP 태스크,

컨텐츠 Logic, 메인 Update를 담당하게 될 MainProcess 태스크 두 개의 태스크가 생성된다.

TestClient 에서는 EventSelect 태스트 하나가 생성 된다.

4-2. 태스크 등록 및 동작 FlowChart



연결(Connect)를 관리하고 Send, Recv 등 연결 개체로써의 역할을 수행할 세션의 동작 중 연결 작업이다.

SessionManager는 초기화(Initialize)단계 에서 ServerConfig.ini 파일에 입력되어 있는 INIT\_SESSION\_COUNT,

MAX\_SESSION\_COUNT 값일 읽어 들이며, 미리 Session 객체를 INIT\_SESSION\_COUNT 만큼 생성하고 AcceptEX 함수를 호출하여 AcceptPool을 생성하고 IOCP는 GetQueueCompletionStatus를 대기 하게 된다. (WSAAccept가 아닌 AcceptEx를 사용한 이유는 WSAAccept함수는 접속이 생기면 소켓을 생성하지만, AcceptEx는 미리 만들어 놓은 소켓을 사용하여 AcceptPool을 생성하는데 유리할 것이라고 생각하였다.)

|  |
| --- |
| while ((\*running))  {  success = GetQueuedCompletionStatus(iocp\_handle\_, &transfered\_byte\_count, &completion\_key, &overlapped, INFINITE);  if (!completion\_key)  break;  ...  if ((success == false) || (success && transfered\_byte\_count == 0))  {  if (overlapped\_ex->io\_type == IO\_ACCEPT)  OnIoConnected(object);  else  OnIoDisconnected(object);  continue;  }  ...  } |

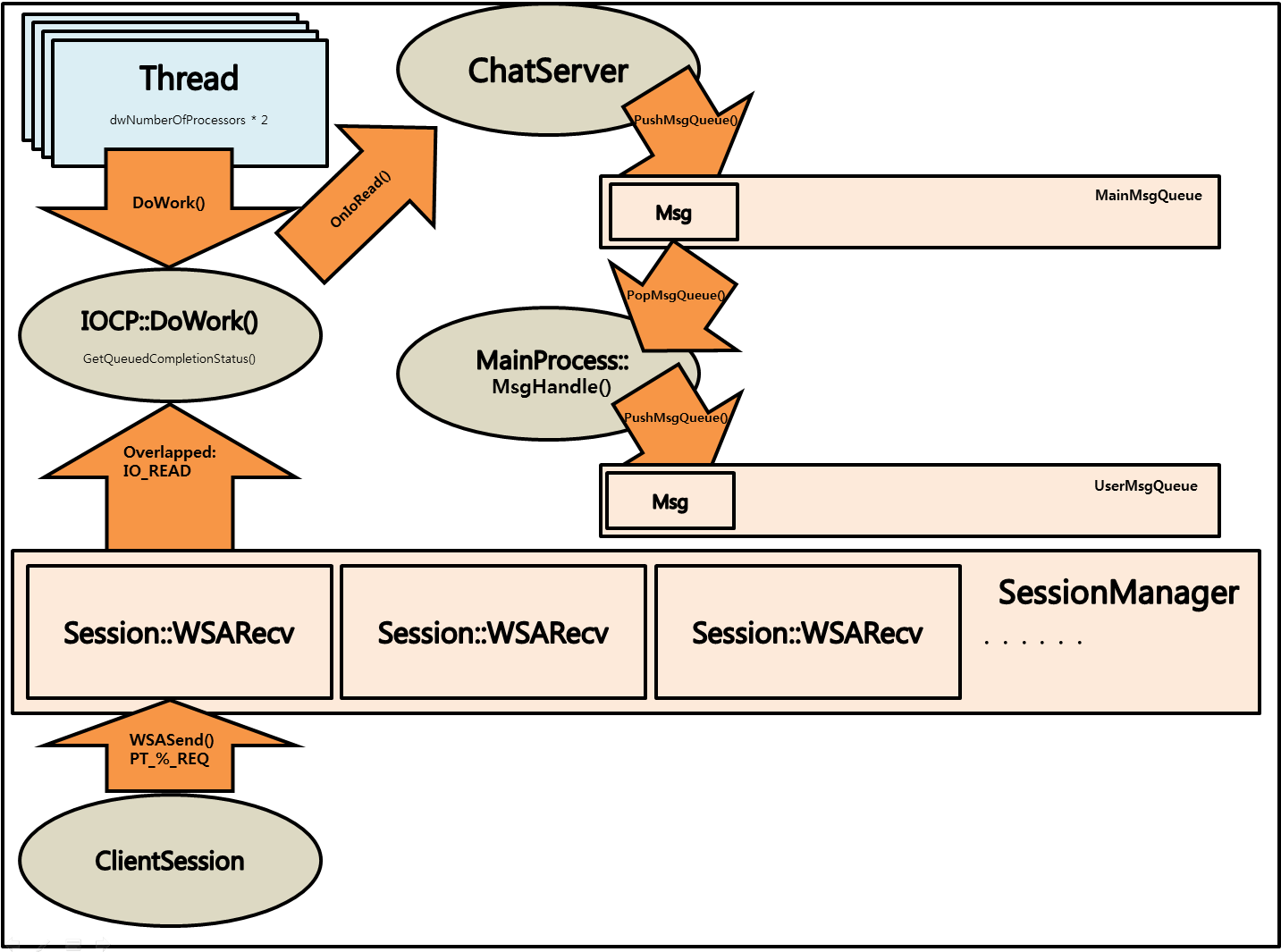
Client로부터 Connect 요청이 오면 GetQueueCompletionStatus로 부터 IO\_ACCEPT 가 담긴 overlapped\_ex 객체를 통해 ChatServer의 OnIoConnected() 함수를 호출하게 된다. ChatServer 클래스는 Client의 Connect 요청이 담긴 Msg를 MainMsgQueue에 Push하는 PushMsgQueue() 함수를 호출한다.

|  |
| --- |
| void ChatServer::OnIoConnected(void \*obejct)  {  Session \*session = reinterpret\_cast<Session\*>(obejct);    if (Iocp::RegistSocket(session->GetSocket(), reinterpret\_cast<ULONGLONG>(session)) == false)  return;  Msg \*msg = new Msg(PT\_SESSION\_CONNECT);  msg->SetSessionID(session->GetID());  MAIN\_PROCESS.PushMsgQueue(msg);  } |

MainMsgQueue는 Singleton을 상속한 MainProcess 멤버 컨테이너이므로 유일하며, Concurrent\_queue 형태로 멀티 스레드환경에서 push, try\_pop이 스레드 세이프 하다는 특징을 가진 켄테이너이다. MainProcess는 MainMsgQueue를 Tick단위로 감시하며, Msg가 있을 경우, MainProcess에서 처리해야 하는 이슈(Connect, DisConnect, KEEP\_ALIVE 등)라면 MainProcess 내에서 MsgHandle 처리를 하게 된다. (특정 유저의 Msg라면, User객체 내의 UserMsgQueue 컨테이너를 이용한다. 4-3.)

연결이 완료된 세션은 RecvWait()함수가 호출되며, WSARecv를 통해 연결을 비동기로 대기 하게된다. 또, 해당 세션의 연결이 끊어 지면, AcceptWait()함수를 호출에 다른 Client의 Connect를 대기한다.

4-3. 기본 로직 처리 FlowChart



이미 연결(Connect)과정이 완료된 세션은 WSARecv 상태가 되며, 본격적으로 Client의 IO를 완료를 대기하기 시작한다. Client로부터 **MsgID : PT\_%\_REQ**인 Msg의 IO가 완료되어 GetQueuedCompletionStatus 함수가 반환되었을 때, IO\_ACCEPT와 마찬가지로, io\_type을 통해 io를 구분 짓고 올바른 함수를 호출한다.

|  |
| --- |
| while ((\*running))  {  success = GetQueuedCompletionStatus(iocp\_handle\_, &transfered\_byte\_count, &completion\_key, &overlapped, INFINITE);  if (!completion\_key)  break;  ...  switch (overlapped\_ex->io\_type)  {  case IO\_READ:  OnIoRead(object, transfered\_byte\_count);  break;  case IO\_WRITE:  OnIoWrote(object, transfered\_byte\_count);  break;  }  } |

이렇게 호출된 OnIoRead()함수는 IO가 완료된 Msg를 세션으로부터 뽑아내어, MainMsgQueue에 Push하게 되며, 세션은 다시 RecvWait()를 호출해 WSARecv 상태로 전환된다.

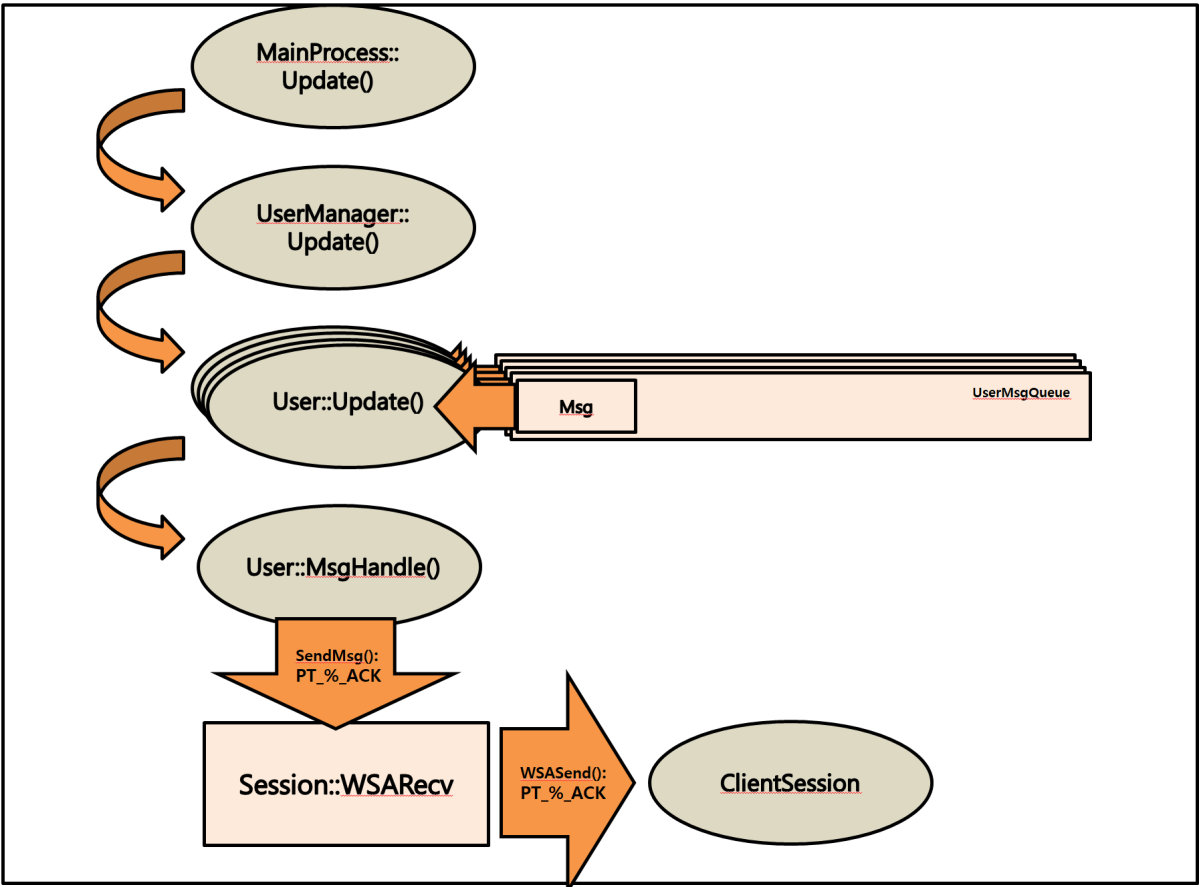
|  |
| --- |
| void ChatServer::OnIoRead(void \*obejct, DWORD data\_length)  {  Session \*session = reinterpret\_cast<Session\*>(obejct);  if (session->RecvByIocp(data\_length))  {  Msg\* msg = nullptr;  session->GetMsg(msg);  if (msg == nullptr)  return;  ...    msg->SetSessionID(session->GetID());  MAIN\_PROCESS.PushMsgQueue(msg);  }  if (session->RecvWait(listener\_->GetSocket()) == false)  ...  } |

MainProcess에서 PopMsgQueue를 통해 뽑아낸 Msg가 MsgHandle에 MsgID가 등록 되지 않았을 경우에는 Msg의 세션 ID를 통해 User 객체를 찾아, 해당 UserMsgQueue에 Push를 수행하며, User객체의 Update() 과정 중 수행되게 된다.

|  |
| --- |
| void MainProcess::MsgHandle(Msg \*msg)  {  if (msg->GetSessionID())  {  Session\* session = SESSION\_MANAGER.GetSession(msg->GetSessionID());  if (session == nullptr)  {  ...  }  switch (msg->GetID())  {  // MainProcess에서 처리될 Msg ID  case PT\_SESSION\_CONNECT:...break;  case PT\_SESSION\_DISCONNECT:...break;  case PT\_CHECK\_VERSION\_REQ:...break;  case PT\_REG\_USER\_REQ:...break;  case PT\_UNREG\_USER\_REQ:...break;  // User가 처리 해야 할 Msg ID는 User의 MsgQueue로 이동  default:  {  User\* user = USER\_MANAGER.FindUser(session->GetID(), true);  if (user)  user->PushMsgQueue(msg);  }  break;  }  }  } |

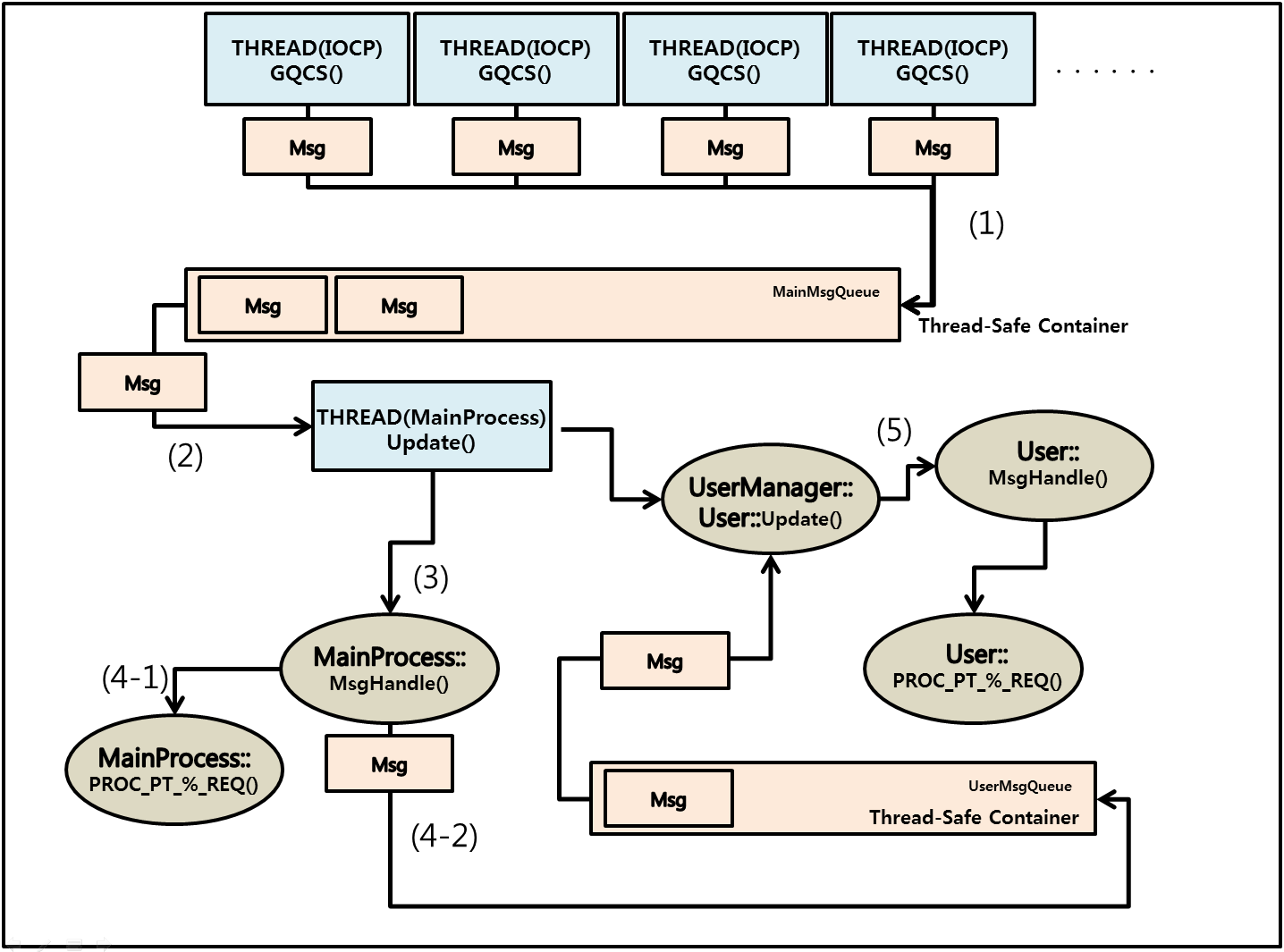
이렇게 넘어온 Msg는 User의 MsgHandle()을 통해 남은 작업을 처리하게 된다.

|  |
| --- |
| void User::MsgHandle(Msg\* msg)  {  switch (msg->GetID())  {  case PT\_CHAT\_REQ:  PROC\_PT\_CHAT\_REQ(msg);  break;  default:  ...  break;  }  } |



위 과정이 끝나면, Client에게 PT\_%\_ACK 메시지를 전송하며, 해당 Msg(ID:PT\_%\_ACK)의 처리를 완결 짓게 된다.

4-4. 멀티 스레드 IOCP와 MainProcess 처리 FlowChart



예제로 있는 채팅 서버는 두 개의 태스크를 등록하여 사용한다. 그 중 IOCP의 경우 CPU 물리 코어 \* 2 개의 개수 만큼 스레드를 가지는데 이는 멀티 스레드 환경에서 IO 완료 가 이루어 진다고 볼 수 있고, 서로 다른 스레드에서 Msg 가 전달되어 처리를 해야 한다는 것이다. 그 결과 잘못된 동기화 작업을 하게 되면, 예상치 못하는 결과 혹은, 데드락 같은 성능의 저하를 경험할 수 있다.

이 예제에서는 한가지 해결 책으로 멀티 스레드 환경에서 IOCP의 처리는 이루어 지되 Thread-Safe 한 컨테이너인 MainMsgQueue를 사용하여 Msg의 직렬화를 통해 해결했다.

|  |
| --- |
| Concurrency::concurrent\_queue<Msg\*> main\_msg\_queue\_; |

(1) concurrent\_queue는 push, try\_pop에 대해 Thread-Safe하므로, 이 컨테이너에 멀티 스레드 환경에서의 IOCP로 수신 받은 Msg를 Push하여 Msg의 줄을 세운다.

(2) 또 다른 태스크인 MainProcess 태스크는 1개만의 스레드를 가진다. 메인 로직을 처리하기 위한 태스크로 그림에서처럼 MainMsgQueue에 쌓인 Msg의 가장 첫 번째 Msg를 뽑아 내려는 시도(try\_pop)를 꾸준히(Update) 시도한다.

(3) Msg가 존재할 경우 try\_pop은 true를 반환하며, MainProcess::MsgHandle()을 수행한다.

(4) MainProcess 내에서 처리 되어야 할 내용들 (SESSION\_CONNECT, SESSION\_DISCONNECT) 은 MainProcess 내에서 처리되며, 처리가 완료된 Msg는 소멸한다.

(4-1) MainProcess 내에서 처리될 MsgID가 아닌 경우(유저 Content 처리)는 잘못된 MsgID로 리턴되거나 해당 유저를 찾아 유저 멤버 컨테이너인 UserMsgQueue에 Push하게된다.

(5) MainProcess::DoWork()는 MainProcess::Update() 뿐만 아니라 컨텐츠 로직에 사용되는 %\_MANAGER 객체의 Update() 호출하며, %\_MANAGER 는 자신이 관리하는 객체의 Update()를 호출하게 된다.

(USER\_MANAGER.Update()의 경우 관리 대상인 User의 Update()를 호출한다.)

|  |
| --- |
| void MainProcess::DoWork(void\* param)  {  bool \*running = (bool\*)param;  while ((\*running))  {  ULONGLONG current\_tick = ::GetTickCount64();  Update(current\_tick);  USER\_MANAGER.Update(current\_tick);  SESSION\_MANAGER.Update(current\_tick);  ...  }  } |
| void UserManager::Update(ULONGLONG current\_tick)  {  for each(const Users::value\_type& user in users\_)  (\*user.second).Update(current\_tick);  } |

User::Update() 과정에서는 UserMsgQueue에 담긴 Msg를 뽑아 내려 시도(try\_pop)을 시도하며, Msg가 존재하는 경우에 User::MsgHandle()을 호출하며, 해당 User가 Msg에 맞는 작업을 진행하고, 완료된 Msg를 제거한다.

4-5. Msg Read, Write

Msg는 Server와 Client 사이에서 통신될 데이터인 Packet을 조금 더 구조화 시킨 객체로써, Packet의 바이트 배열에 존재하는 Protocol ID, Data 등을 Msg 클래스의 멤버 형태로 표현하여 사용자가 Server, Client 에서 데이터를 송수신(WSASend, WSARecv)할 때 간편하게 사용할 수 있는 인터페이스를 제공한다.

|  |
| --- |
| class IMsg  {  public:  **...**  // -------------------- WRITE --------------------  IMsg& operator << (const DWORD data) …  IMsg& operator << (const ULONGLONG data) …  IMsg& operator << (const BYTE data) …  IMsg& operator << (BYTE \*data) …  IMsg& operator << (const SHORT data) …  IMsg& operator << (const int data) …  IMsg& operator << (const \_\_int64 data) …  IMsg& operator << (const float data) …  IMsg& operator << (const bool data) …  IMsg& operator << (const wchar\_t data) …  IMsg& operator << (const std::wstring& data) …  // -------------------- READ --------------------  IMsg& operator >> (DWORD \*data) …  IMsg& operator >> (ULONGLONG \*data) …  IMsg& operator >> (BYTE \*data) …  IMsg& operator >> (SHORT \*data) …  IMsg& operator >> (int \*data) …  IMsg& operator >> (\_\_int64 \*data) …  IMsg& operator >> (float \*data) …  IMsg& operator >> (bool \*data) …  IMsg& operator >> (wchar\_t \*data) …  IMsg& operator >> (std::wstring& data) …  };  class Msg : public MemoryPool<Msg>, public IMsg  {  public:  **...**  private:  DWORD session\_id\_;  DWORD offset\_;  DWORD protocol\_id\_;  BYTE \*buffer\_;  }; |

실제 데이터를 송신(WSASend)하는 방법은 다음과 같다.

|  |
| --- |
| void User::PROC\_PT\_CHAT\_REQ(Msg\* msg)  {  ...  Msg\* ack\_msg = new Msg(PT\_CHAT\_ACK);  \*ack\_msg << chat.text;  SendMsg(ack\_msg);  } |

생성된 Msg 객체 포인터를 대상으로 오버라이딩 된 << 연산자를 사용하여 Msg 버퍼에 chat.txt라는 문자열을 입력한 후 SendMsg함수를 사용하여 User와 맵핑된 Client Session에게 Msg를 송신한다. SendMsg 함수 내부에는 전달받은 Msg 객체에서 Msg ID, Data buffer를 통해 패킷을 만들어 송신하는 과정이 존재한다.

데이터를 수신(WSARecv)하는 방법은 다음과 같다.

|  |
| --- |
| void User::PROC\_PT\_CHAT\_REQ(Msg\* msg)  {  ...  CHAT chat;  READ\_PT\_CHAT(msg, chat);  ...  }  //PT\_ReadMsg.h  inline void READ\_PT\_CHAT(Msg \*msg, CHAT &parameter)  {  \*msg >> parameter.text;  } |

Msg의 Data를 읽어 들이는 과정은 작업자가 따로 PT\_ReadMsg.h 파일에 함수를 정의하여 사용하는 방법을 사용하였다. PT\_ReadMsg.h 파일만 존재하고 PT\_WriteMsg.h 같은 파일이 존재하지 않는 이유는 작업자가 Msg를 데이터를 쓰는 과정은 << 연산자를 연속적으로 사용하면 간단하게 해결 되지만, Msg에서 읽어 내는 과정에서는 쓰인 순서, 자료형의 크기 등 고려해야 할 사항이 많아 진다.

때문에 Msg에 데이터를 쓰는(Write) 작업자가PT\_ReadMsg.h 파일에 별도로 READ\_PT\_% 함수를 제공하는 방법을 고려해 본 것이다.

5. 개선사항

- DBConn 구현 (ODBC, ADODB ... )

- MsgHandle() 시 비동기로 처리되어야 할 작업을 처리할 수 있도록 (OverlappedJobManager or

AsyncJobManager) 구현

- MemoryPool 클래스 수정

- ThreadSync 수정 (현재는 SINGLE\_LOCK 만 존재, MULTI\_LOCK 구현)

- UDP 통신 소켓 추가