

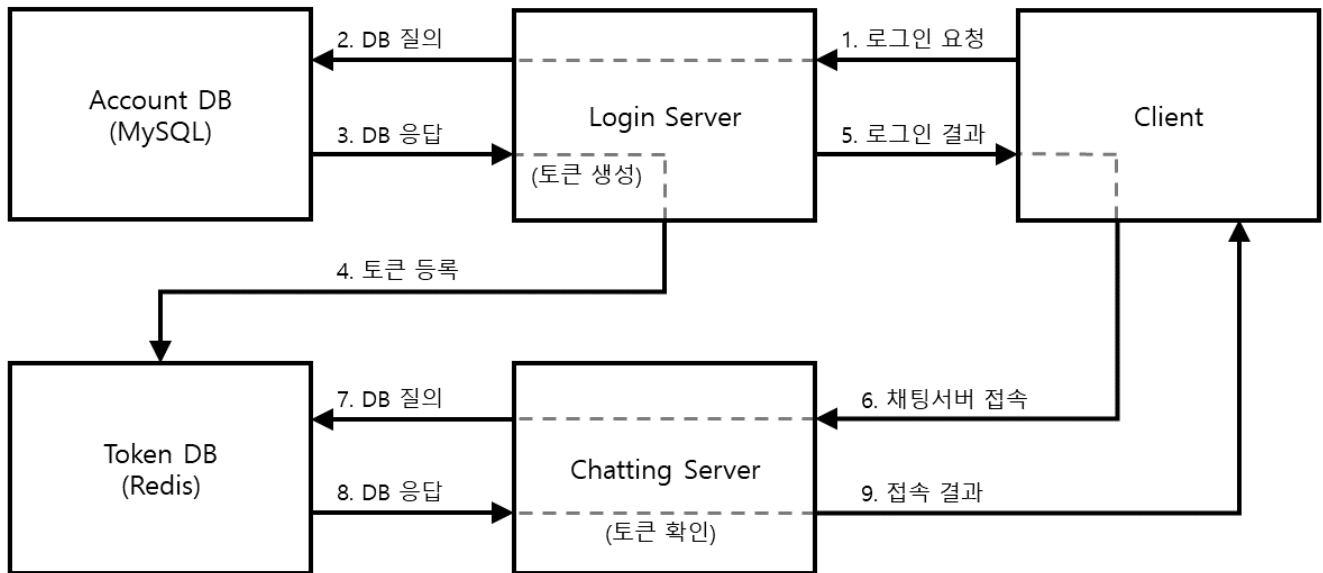
게임 서버 포트폴리오

최준영

목차

1. 서버 아키텍처 구조	3
2. 네트워크 라이브러리 소개	4
2-1. 네트워크 라이브러리 스레드 구조	6
2-2. 상세설명 및 기술적 요소	7
2-3. 구현하며 발생한 문제와 해결	11
3. MMO 프로젝트	13
3-1. MMO 싱글 스레드 채팅 서버	13
3-2. MMO 멀티 스레드 채팅 서버	19
3-3. 싱글/멀티 스레드 채팅 서버 성능 비교	22
3-4. 로그인 서버	23
3-5. 모니터링 서버	25
4. 락프리 자료구조	28
4-1. 락프리 스택	28
4-2. 락프리 메모리 풀	31
4-3. 락프리 큐	33
4-4. new 보다 13배 빠른 메모리풀	42
5. 서버 개발을 위해 구현한 클래스	45
5-1. DBConnector	45
5-2. DBConnectorTLS	46
5-3. Profiler	47
5-4. Parser	48

1. 서버 아키텍처 구조



완성된 서버 구조는 위와 같습니다.

1. 로그인 요청

클라이언트는 채팅 서버에 접속하기 위해 먼저 로그인 서버를 거쳐야합니다.
클라이언트는 로그인 서버로 AccountNo, Key가 담긴 로그인 요청 패킷을 송신합니다.

2, 3. Account DB 조회, 유효 클라이언트 여부 판단

로그인 서버는 클라이언트가 유효한 클라이언트인지 판단하기 위해, DB에 로그인 요청 Account에 해당하는 Key를 조회하여 로그인 요청패킷의 Key와 비교합니다.

4, 5. 토큰 등록, 로그인 결과 통지

채팅서버에서 클라이언트의 유효성을 판단하기 위한 토큰을 발급하여 Redis에 등록하고, 클라이언트에게 채팅서버의 주소정보, 토큰을 포함하는 로그인 결과 패킷을 회신합니다.

6. 채팅서버 접속

클라이언트는 로그인서버에서 받은 채팅서버 주소로 로그인 요청패킷을 송신합니다.

7, 8. Token DB 조회, 유효 클라이언트 여부 판단

채팅 서버는 클라이언트가 유효한 클라이언트인지 판단하기 위해, Token DB에 Account에 해당하는 토큰을 조회하여 로그인 요청패킷의 토큰과 비교합니다.

9. 접속 결과

채팅서버에서 클라이언트의 유효 판단 결과를 담은 로그인 응답패킷을 송신합니다.

※ 위 과정이 성공적으로 이루어졌다면, 콘텐츠 메시지를 주고 받습니다.

2. 네트워크 라이브러리 소개

동기화 객체를 사용하지 않은 네트워크 라이브러리를 구현했습니다.

라이브러리의 구현과 테스트 과정은 어려운 작업이었지만, 이 후 채팅서버, 로그인서버 등을 구현할 때 라이브러리를 사용하여 구현함으로써 생산성 향상을 체감했습니다. 또한, 네트워크 기능과 서버 콘텐츠 로직이 분리됨으로써 유지보수에 용이했습니다.

(github : https://github.com/dkdldjswkd/cpp_Project/tree/main/01%20Network/NetworkLib)

```
class NetServer {
public:
    NetServer(const char* systemFile, const char* server);
    virtual ~NetServer();

    // 중략 (private 멤버와 메서드)

protected:
    // 라이브러리 사용자 측 재정의 하여 사용
    virtual bool OnConnectionRequest(in_addr ip, WORD port) = 0;
    virtual void OnClientJoin(SessionId sessionId) = 0;
    virtual void OnRecv(SessionId sessionId, PacketBuffer* contentsPacket) = 0;
    virtual void OnClientLeave(SessionId sessionId) = 0;
    virtual void OnServerStop() = 0;

public:
    // 서버 ON/OFF
    void Start();
    void Stop();

    // 라이브러리 제공 API
    void SendPacket(SessionId sessionId, PacketBuffer* sendPacket);
    void Disconnect(SessionId sessionId);

    // Getter
    void UpdateTPS();
    DWORD GetSessionCount();
    DWORD GetAcceptTPS();
    DWORD GetAcceptTotal();
    DWORD GetSendTPS();
    DWORD GetRecvTPS();
};
```

네트워크 라이브러리 사용자는 추상 클래스인 NetServer를 상속받아, 사용자 서버 클래스를 구현합니다. 사용자는 네트워크 이벤트 시 호출되는 callback 함수를 재정의 하는 방법으로 사용자 서버를 구현하며, 네트워크 라이브러리에서 지원하는 SendPacket(), Disconnect() 등의 함수를 사용할 수 있습니다.

사용자 재정의 Callback 함수

1. bool OnConnectionRequest(in_addr ip, WORD port);

accept()의 성공적인 반환 시 호출되는 함수로, 클라이언트의 접속을 수락하거나 거절하는 작업을 수행하기 위한 함수입니다. false를 반환하면 클라이언트와 연결을 해제합니다.

2. void OnClientJoin(SessionId sessionId)

클라이언트 접속 승인 후 네트워크 라이브러리에서 클라이언트 관리를 위한 Session 배정 후 호출되는 함수입니다. 네트워크 라이브러리 사용자에게 클라이언트 접속을 알려, 클라이언트 접속 시 해야할 작업을 수행하기 위한 함수입니다. 네트워크 라이브러리 사용자는 해당 함수에서 session Id를 보관하거나, 클라이언트에게 접속 성공 메시지를 송신할 수 있습니다. Session Id는 클라이언트를 식별하는 고유한 값입니다.

3. void OnRecv(SessionId sessionId PacketBuffer* contentsPacket)

클라이언트의 메시지 수신 시 호출되는 함수로, 메시지 처리를 수행하기 위한 함수입니다. 매개변수로 Session Id, Packet을 갖기때문에 클라이언트를 식별하고 메시지 처리를 할 수 있습니다. 클라이언트로부터 받은 패킷이 온전히 메시지 단위로 조립될 수 있어야 호출됩니다.

4. void OnClientLeave(SessionId sessionId)

클라이언트와의 연결이 종료되었을 때 호출되는 함수로, 클라이언트 연결 종료 시의 작업을 수행하기 위한 함수입니다. 매개변수로 Session Id를 갖기 때문에 해당 클라이언트에대한 리소스 정리하는 작업을 수행 할 수 있습니다.

5. void OnServerStop()

라이브러리의 Stop()을 사용하여 서버 종료 시 라이브러리의 리소스 정리 후 Stop()에서 호출되는 함수로, 사용자 서버의 리소스 정리 작업을 수행하기 위한 함수입니다. 사용자 서버에서 생성한 스레드등의 자원을 정리하는 작업을 수행할 수 있습니다.

라이브러리 제공 API

1. void Start()

listen()을 호출, Accept Thread/Timeout Thread를 생성하며 서버를 시작합니다.

2. void Stop()

라이브러리의 리소스를 정리하고 서버를 종료합니다.

3. void SendPacket(SessionId sessionId, PacketBuffer* sendPacket)

클라이언트에게 메시지를 송신하는 함수입니다. 내부적으로 WSASend를 호출하여 메시지를 송신합니다.

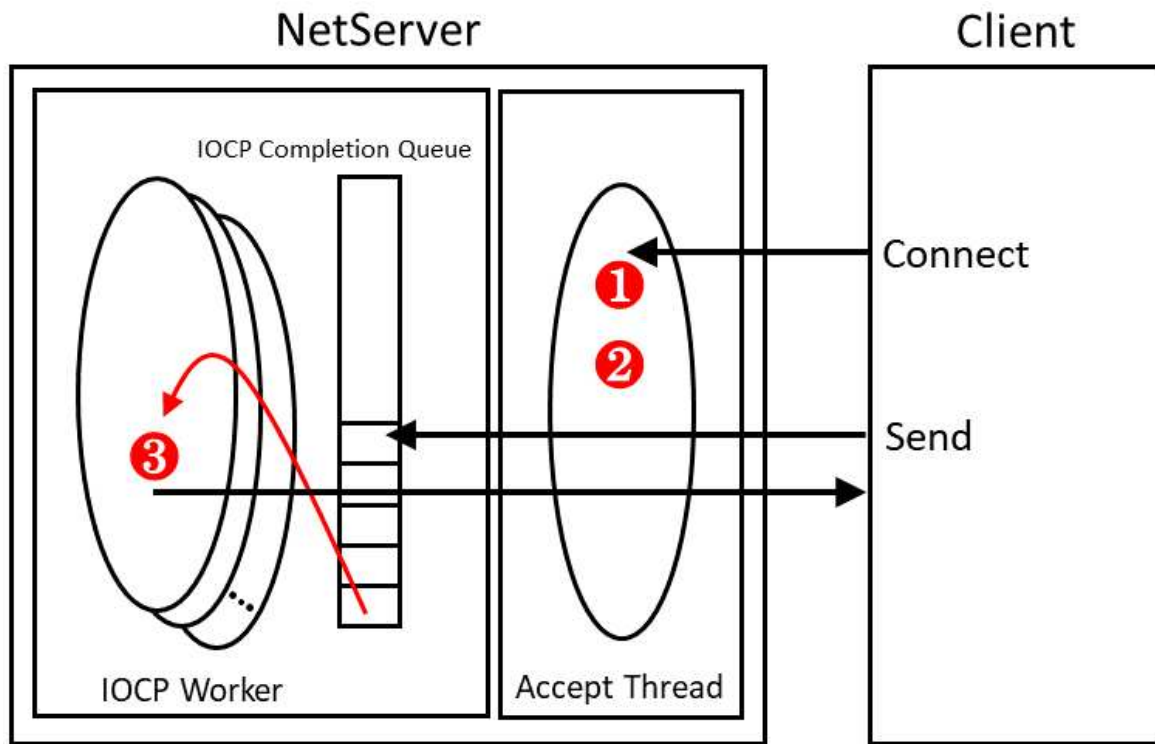
4. void Disconnect(SessionId sessionId)

클라이언트와의 연결을 끊기위해 호출하는 함수입니다.

5. Getter

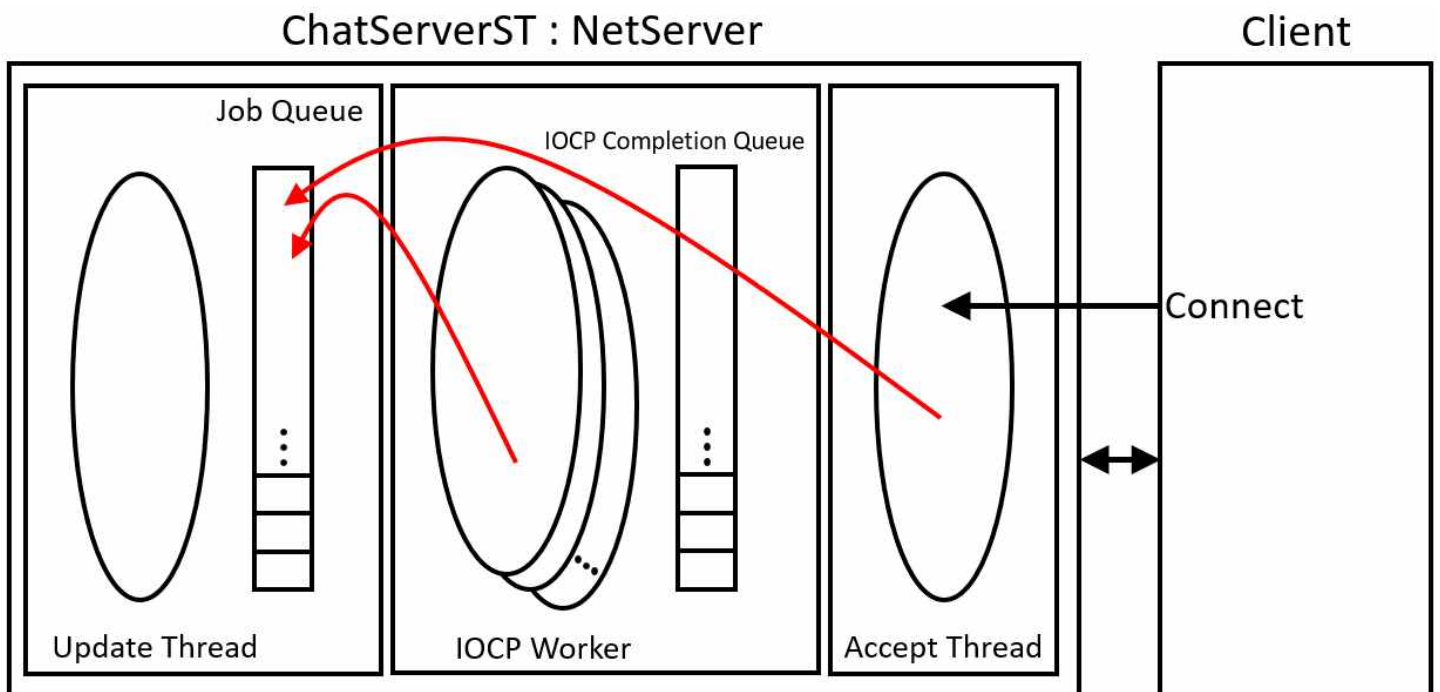
서버에서 카운팅한 '서버에 접속한 클라이언트 수', '송/수신 메시지 수' 등의 모니터링 데이터를 반환하는 함수입니다. UpdateTPS()를 호출 하여 데이터를 최신화 후 사용해야합니다.

네트워크 라이브러리 스레드 구조



(위의 그림은 네트워크 라이브러리의 구조를 간단하게 시각화 한 모습입니다.)

- ❶ : accept()의 성공적인 반환 후 위에서 설명한 OnConnectionRequest()의 호출 시점입니다.
- ❷ : 라이브러리에서 연결 클라이언트에 대한 Session 배정 후 OnClientJoin()의 호출 시점입니다.
- ❸ : 클라이언트로부터 메시지를 수신하여 OnRecv()가 호출되는 시점입니다.
- ❹ : 여기서 클라이언트로부터 받은 메시지를 IOCP Worker에서 즉시 처리할 수도 있고, 외부 사용자 생성 스레드로 넘겨 처리할 수도 있습니다.



(위의 그림은 ❷, ❸에서 즉시 '로그인, 메시지처리'를 하지않고 사용자 스레드로 넘겨서 처리하는 예시입니다.)

상세설명 및 기술적 요소

1. 클라이언트에 대한 Recv/Send를 1회로 제한했습니다.

제한 이유 : Recv/Send는 syscall로 비용이 높은 함수입니다. 즉, 호출 주기가 짧아진다면 **성능저하의 원인**이 될 수 있습니다. 또한, Recv/Send 호출 시 사용되는 Overlapped 구조체는 완료통지 시 까지 보존되어야하는데 **Overlapped 구조체 관리또한 번거로운 작업**입니다.

Recv : IOCP의 I/O Completion Queue에 쌓이는 I/O 완료통지는 Recv의 작업 순서를 보장합니다. 다만, **Worker에서의 작업순서는 보장되지 않으므로 Recv를 1회로 제한했습니다.** 연결된 클라이언트에 대해 Recv를 호출하며, Recv 완료통지 시 Recv를 재등록하는 하는 방법으로 **클라이언트 당 Recv 1회를 유지**했습니다.

Send : 제한 이유는 위와 같습니다. 라이브러리에서 클라이언트에 대한 Send 처리 전, Send 진행 여부를 확인합니다. 만약 **Send가 진행중이라면 Session의 SendQ에 메시지를 큐잉**하고, Send 완료처리 시 **sendQ를 확인하여 다시 쌓인 메시지를 전부 Send** 합니다. 해당 방법으로 클라이언트 당 Send 1회를 유지했습니다.

2. Session 정리 조건과 IOCount 도입 배경

Session을 정리하기 위한 조건은 '클라이언트와 연결해제 시'가 아닌, '클라이언트에 대한 모든 I/O 완료통지 이후'로 판단했습니다. 이유는 연결이 끊어졌다고 해도, 아직 완료통지를 받지 못한 I/O가 있다면 Worker의 I/O 완료통지 코드에서 Session에 접근하기 때문입니다. 만약 연결이 끊어졌을 때 Session을 정리했다면, 이는 정리된 리소스에 접근하는 아주 위험한 상황이 되어버립니다. 이를 위해 세션의 I/O 진행상황 및 완료통지 처리 여부를 판단하기위해 Session의 멤버로 'IOCount'를 추가했습니다.

IOCount 증감의 조건은 다음과 같습니다. Recv/Send 호출 전 IOCount를 증가하며, 'I/O에 대한 완료통지 작업 후' 또는 'Recv/Send 호출 실패 시' IOCount를 감소합니다. 라이브러리에서는 Recv 완료 통지 작업 후 해당 클라이언트에 대해 다시 Recv를 요청하기 때문에 정상적인 세션의 경우 IOCount는 항상 1 이상으로 유지됩니다.

Session의 IOCount가 0이 되는 경우, 네트워크 라이브러리는 클라이언트에 대한 리소스를 정리합니다. Session IOCount 0이 의미하는것은 연결 해제 및 클라이언트에 대한 모든 I/O 완료통지 코드가 진행되었음을 뜻합니다. (예외적으로 서버에서 클라이언트를 먼저 끊는경우 의도적으로 IOCount를 0으로 유도합니다.)

3. SendPacket(), Disconnect() 제약과 해결

Worker의 완료통지 코드에서 접근하는 Session은 Release Session 일 수 없습니다. Recv/Send 호출 전 IOCount를 증가시키며, Worker의 완료통지 코드 진행 후 IOCount를 감소시키기 때문에 최소한 완료통지 코드가 전부 진행 되어야 IOCount 0으로 인한 Session Release가 진행될 수 있습니다.

하지만, **Worker 외 제 3 스레드에서 접근하는 Sessoin은 Release Session이 아님을 보장하지 못합니다.** Release된 Session 일 수도 있고, Release 후 새로 접속한 클라이언트에 의해 사용(재사용) 중 일 수 있습니다. 재사용되는 경우 Disconnect() 호출 시 엉뚱한 클라이언트를 끊어버리게 되며, SendPacket() 호출 시 엉뚱한 클라이언트에게 메시지를 보내는 등 큰 문제가 생길 수 있습니다.

즉, 라이브러리 제공함수 SendPacket()과 Disconnect()는 Worker에서 호출되는 OnRecv() 내부에서만 사용가능하다는 뜻이 됩니다. 이는 서버의 능동적인 연결해제 및 메시지 송신이 제한된다는 의미입니다.

```
NetServer.cpp // NetServer::ValidateSession()

Session* NetServer::ValidateSession(SessionId sessionId) {
    Session* p_session = &sessionArray[sessionId.sessionIndex];
    IncrementIOCount(p_session); SessionId 에서 추출한 Index로 Session Find
    * 다른 스레드에서 Release 되지 않도록 IOCount 증가
    // 세션 릴리즈 상태
    if (true == p_session->releaseFlag) {
        DecrementIOCountPQCS(p_session); Release 여부 체크
        return nullptr; (IOCount 증가전 Release 상태였을 수 있음)
    }
    // 세션 재사용
    if (p_session->sessionId != sessionId) {
        DecrementIOCountPQCS(p_session); 재사용 여부 체크
        return nullptr; (재사용 Session을 대상으로 IOCount 증가 시켰을 수 있음)
    }

    // 재사용 가능성 제로
    return p_session;
} 위의 조건을 모두 통과했다면, 재사용 Session이 아니며 Release 될 수 없음
```

위의 VaildateSession()은 이 문제를 해결하기 위한 함수입니다. SendPacket(), Disconnect()가 Worker 외 스레드에서 사용제약이 있는 이유는 Session의 사용 중 Release의 가능성이 있기 때문입니다. 풀어 말하자면, IOCount를 물고있는채로 작업하지 않기 때문입니다.

Worker 외 스레드에서 Sessoin 사용 시 IOCount를 먼저 증가시킨다면, Session이 Release 될 수 없으므로 해당 문제를 해결할 수 있다고 판단 했습니다.

ValidateSession()는 IOCount를 증가시키고, Session의 상태가 유효하다면 Session 사용을 허가하는 함수입니다. 이로인해 Worker에서 IOCount를 감소 시킨다 하더라도 IOCount가 0인 경우를 만들지 않음으로써 Session Release 실행이 제한됩니다. 해당 함수의 구현으로 인해, 서버에서 능동적으로 클라이언트를 끊거나 메시지를 송신할 수 있게 되었습니다.

4. ReleaseSession()

NetServer.cpp // NetServer::ReleaseSession()

```
bool NetServer::ReleaseSession(Session* p_session) {
    if (0 != p_session->ioCount)
        return false;

    // * releaseFlag(0), iocount(0) -> releaseFlag(1), iocount(0)
    if (0 == InterlockedCompareExchange64((long long*)&p_session->releaseFlag, 1, 0)) {
        // 리소스 정리 (소켓, 패킷)
        closesocket(p_session->sock);
        PacketBuffer* packet;
        while (p_session->sendQ.Dequeue(&packet)) {
            PacketBuffer::Free(packet);
        }
        for (int i = 0; i < p_session->sendPacketCount; i++) {
            PacketBuffer::Free(p_session->sendPacketArr[i]);
        }
        Session 리소스 정리
        // 사용자 리소스 정리
        OnClientLeave(p_session->sessionId);
        // 세션 반환
        sessionIdxStack.Push(p_session->sessionId.session_index);
        InterlockedDecrement((LONG*)&sessionCount);
        return true;
    }
    return false;
}
```

해당 CAS 연산 성공 시 Sessoin Release 진행
다른 스레드에서 IOCount 증가 시 해당 연산 실패

사용자 정의 Callback 함수 OnClientLeave() 호출

Sessoin Index 반환, 다음 접속 클라이언트는
지금 반환된 Index의 Session 메모리에 할당됨

네트워크 라이브러리에서는 Sessoin의 IOCount 감소 후 값을 판단하여, 0이라면 ReleaseSession()을 호출합니다. 위의 코드를 보면 releaseFlag 뿐만 아니라 IOCount가 0인지 재차 확인하고 있습니다. 이는 IOCount가 0으로 감소했지만, 찰나의 순간에 외부 스레드에서 Session 사용을 위해 ValidateSession()을 호출하여 IOCount를 증가 시킨 경우입니다. 이때는 리소스를 정리하여 발생하는 문제를 막기 위해 Release 수행을 하지않고 반환하며, IOCount를 증가시킨 쪽에서 Session에 대한 Release 책임을 갖습니다.

5. 클라이언트의 Timeout 처리

NetServer.cpp // NetServer::TimeoutFunc()

```
void NetServer::TimeoutFunc() {
    for (;;) {
        Sleep(timeoutCycle);
        INT64 cur_time = timeGetTime(); Timeout 기준 시간 얻기
        for (int i = 0; i < maxSession; i++) {
            Session* p_session = &sessionArray[i];
            SessionId id = p_session->sessionId; Session 전체 순회

            // 타임아웃 처리 대상 아님 (Interlocked 연산 최소화하기 위해)
            if (sessionArray[i].releaseFlag || (timeOut > cur_time - sessionArray[i].lastRecvTime))
                continue;
            // 마지막 수신 시간이 Timeout 주기를 초과하지 않음
            // -> 타임아웃 처리 대상이 아님, continue

            // 타임아웃 처리 대상일 수 있음
            IncrementIOCount(p_session);
            // 릴리즈 세션인지 판단. Release 되지 않기 위해, IOCount 증가
            if (true == p_session->releaseFlag) {
                DecrementIOCount(p_session); IOCount 증가전, Release됐을 수 있으므로 *Release더블체크
                continue;
            }

            // 타임아웃 조건 판단.
            if (timeOut > cur_time - sessionArray[i].lastRecvTime) {
                DecrementIOCount(p_session); 재사용 Session일 수 있으므로 *Timeout여부더블체크
                continue;
            }

            // 타임아웃 처리
            DisconnectSession(p_session);
            DecrementIOCount(p_session); Session Timeout
        }
    }
}
```

system file의 TIME_OUT_FLAG 설정 시 실행되는 Timeout 함수입니다. timeoutCycle의 주기로 깨어나, Session Array 전체를 순회하며 Timeout 처리를 합니다. Release 여부와 Timeout 조건을 체크하며 Timeout 조건에 부합된다면, ValidateSession()의 로직과 유사한 방식으로 Session을 얻고 Disconnect 처리합니다.

6. Send 시 데이터 복사 최소화

네트워크 라이브러리는 Session의 SendQ에 패킷의 주소를 SendQ에 큐잉하는 방식을 사용합니다. 이는 Send를 위한 로직에서 데이터 복사를 최소화하기 위함입니다. Unicast 일때 효과가 미비할 수 있지만, Broadcast 시 효과가 극대화 됩니다. 예를들어 100byte 메시지를 100명에게 송신 시 100명의 SendBuffer에 메시지를 복사하게된다면 $100 \times 100 = 10000\text{byte}$ 의 복사가 발생합니다. 하지만, 100명에게 패킷의 주소를 복사한다면, $8 \times 100 = 800\text{byte}$ 로 복사량이 현저히 줄어들게되어 이로인한 성능개선을 기대할 수 있습니다.

Send 완료통지에서는 송신에 성공한 패킷을 SendQ에서 디큐잉하고, 패킷의 참조 카운트를 감소하는 방식으로 패킷의 생명을 관리합니다.

구현하며 발생한 문제와 해결

네트워크 라이브러리 구현 중 어려움이 많았습니다. 기억에 남는 문제상황과 해결과정을 적어보았습니다.

1. Session 할당 후 세션이 끊어지던 문제 (해결방법 : Session의 생성 IOCount 증가)

```
Session.cpp // Session::Set()
void Session::Set(SOCKET sock, in_addr ip, WORD port, SessionId session_id) {
    this->sock = sock;
    this->ip = ip;
    this->port = port;
    this->sessionId = session_id;
    recvBuf.Clear();
    sendFlag = false;
    disconnectFlag = false;
    sendPacketCount = 0;
    lastRecvTime = timeGetTime();

    // 생성하자마자 릴리즈 되는것을 방지
    InterlockedIncrement(&ioCount);
    this->releaseFlag = false; // 생성 IOCount 증가
}
```

클라이언트 접속 시 Session을 할당 후 호출하는 함수 Session::Set()입니다.

Session 멤버 초기화 후 releaseFlag를 false로 바꾸는 순서를 갖는데, 이때 relaseFlag 초기화 전 꼭 IOCount를 올려줘야합니다. 만약, releaseFlag와 IOCount 증가의 순서가 바뀐다면 문제가 발생합니다. 제 3 스레드에서 Session의 IOCount 증감으로 인해 Release가 호출되었고, Set()에서 releaseFlag를 false로 초기화 했다면, 클라이언트가 접속하자마자 Release로 인해 연결이 끊어집니다. IOCount를 releaseFlag 보다 먼저 증가함으로써 이 문제를 해결 했습니다. 세션 할당 후 IOCount를 증가시켜주지 않으면 생기는 두 번째 문제는 OnClientJoin()의 호출에서 SendPacket() 호출 시 IOCount가 증감되기 때문에 연결된 클라이언트에 대해, 메시지 송신 후 바로 연결이 끊켜 버리는 문제가 발생할 수 있습니다. 이를 방지하기 위해 생성 IOCount를 증가 시키고, Recv 호출 후 생성 IOCount를 감소시킵니다.

2. 메시지 송신이 안되던 문제 (0 Byte Send 버그)

NetServer.cpp // NetServer::SendPost()

```
bool NetServer::SendPost(Session* p_session) {
    // Empty return
    if (p_session->sendQ.GetUseCount() <= 0)
        return false; // 보낼 메시지가 없다면 송신하지 않음 (다른 스레드에서 송신 한 경우)

    // Send 1회 체크 (send flag, true 시 send 진행 중)
    if (p_session->sendFlag == true)
        return false;

    if (InterlockedExchange8((char*)&p_session->sendFlag, true) == true)
        return false; // 이미 Send 중 이라면 송신하지 않음 (다른 스레드에서 송신 한 경우)

    // Empty continue
    if (p_session->sendQ.GetUseCount() <= 0) {
        InterlockedExchange8((char*)&p_session->sendFlag, false);
        return SendPost(p_session); // * 보낼 메시지가 있는지 더블체크(중요).
    } // 다른 스레드에서 호출된 SendPost() 에서 송신한 경우 0 Byte Send

    AsyncSend(p_session);
    return true;
}
```

위의 함수 SendPost()는 네트워크 라이브러리에서 Send를 1회로 제한하기 위해 WSASend()를 래핑한 함수입니다. 버그 발생 시나리오는 이렇습니다.

1. A 스레드에서 SendPost() 호출 후 SendQ에 패킷이 있다고 판단합니다.

2. B 스레드에서도 SendPost()를 호출했고, Send 성공 후 SendQ를 전부 비웁니다. (sendFlag == false)

3. A 스레드에서 sendFlag를 확인, Send 진행중이 아니기 때문에 Send 합니다. * 0 Byte Send 발생

* 위 시나리오 발생 시 완료통지 부의 transferr는 0 이므로, 연결 끊김으로 판단하고 SendFlag의 초기화가 진행되지 않습니다. 그렇기 때문에 해당 클라이언트에게 Send 시 SendFlag는 계속 true이므로 송신이 되지 않습니다.

해당 버그 발견 후 SendPost()에 SendQ Size를 더블체크하는 코드가 들어갔으며, Q Size가 0일 시 sendFlag를 false로 초기화 후 재귀 호출하는 방법으로 해결했습니다. (더블체크 중 다른스레드에서 sendQ에 Enqueue 후 sendFlag가 true이기 때문에 Send 하지 않고 반환 했을 수 있으므로 꼭 재귀호출 해야함)

3. 싱글 스레드 서버 퍼포먼스 저하 문제 해결

싱글 스레드 채팅서버 스레드 설계는 UpdateThread에 작업을 직렬화 시켜 처리하는 방식을 사용했습니다. 서버 모니터링 항목 중 Update Q의 Size를 보았을 때 Q에 작업이 쌓이는 것을 확인했습니다. 서버의 성능 개선을 위해 UpdateThread의 메시지 타입 별 소요 시간을 측정 했고, SendPacket()을 호출하는 부분의 점유시간이 압도적이었습니다. 이를 개선하기 위해 PQCS()를 사용하여 Worker로 Send call을 우회하는 방식을 채택했습니다. 이러한 방법으로 1분 평균 Update Q의 Size를 '100~200'에서 '0~1'로 성능을 향상 시킬 수 있었습니다.

4. SRWLOCK 사용 시 재귀 Lock 으로 인한 데드락 발생

멀티스레드 채팅서버를 구현하는 과정에서, OnRecv() 함수에서 Lock을 사용하여 메시지 처리를 했습니다. 메시지를 보내려는 Sector를 잠구고, Sector에 속한 Session들에게 SendPacket()을 호출하는 과정에서 IOCount가 0으로 감소되었고, Session의 Release 시 호출되는 OnClientLeave() 함수에서 해당 Session이 속한 Sector를 잠구고 erase하는 코드가 실행되어, 재귀락으로 인한 데드락이 발생했습니다. 처음의 해결방안으로는 SRWLOCK을 래핑하여 재귀락을 지원하는 클래스를 구현했지만, 네트워크 라이브러리에서 SRWLOCK 사용의 제약은 여전하다고 판단하여, Session의 Release는 Worker에서 실행 될 수 있게 PQCS()를 사용하여 스레드를 우회하는 방법을 선택했습니다.

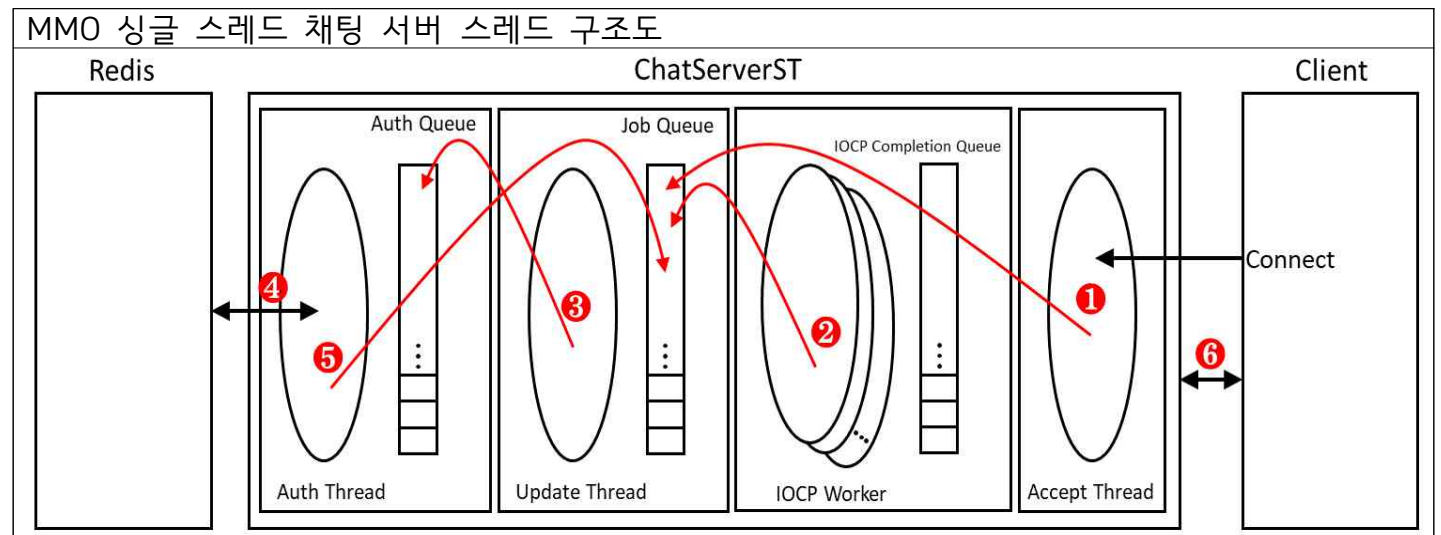
3. MMO 프로젝트

앞서 소개한 네트워크 라이브러리를 사용해 '로그인 서버, MMO 채팅 서버(싱글/멀티 스레드), 모니터링 서버'를 구현했습니다.

MMO 싱글 스레드 채팅 서버

클라이언트로부터 받은 메시지를 채팅 서버의 업데이트 스레드에 직렬화 하여 처리합니다.

(github : https://github.com/dkdldjswkd/cpp_Project/tree/main/01%20Network/ChatServerST)



❶ : 클라이언트의 접속으로 인해 AcceptThread에서 채팅서버 구현 시 오버라이딩한 OnClientJoin()가 호출됩니다. OnClientJoin()에서는 CLIENT_JOIN Type의 Job을 생성하여 UpdateThread의 Job Queue에 큐잉합니다.

```
void ChatServerST::OnClientJoin(SessionId sessionId) {
    JobQueueing(sessionId, JOB_TYPE_CLIENT_JOIN);
}

void ChatServerST::JobQueueing(SessionId sessionId, WORD type) {
    Job* p_job = jobPool.Alloc();
    p_job->Set(sessionId, type);
    jobQ.Enqueue(p_job);
    SetEvent(updateEvent);
}
```

❷ : 클라이언트의 메시지 송신으로 인해 Worker에서 채팅서버 구현 시 오버라이딩한 OnRecv()가 호출됩니다. OnRecv()에서는 메시지의 Type에 해당하는 Job을 생성하여 UpdateThread의 Job Queue에 큐잉합니다.

```
void ChatServerST::OnRecv(SessionId sessionId, PacketBuffer* csContentsPacket) {
    WORD type;
    try {
        *csContentsPacket >> type;
    }
    catch (const PacketException& e) {
        LOG("ChatServerST", LOG_LEVEL_WARN, "Disconnect // impossible : >> type");
        Disconnect(sessionId);
        return;
    }

    // 중략 (INVALID PACKET TYPE 이라면, Disconnect 합니다.)

    csContentsPacket->IncrementRefCount();
    JobQueueing(sessionId, type, csContentsPacket);
}
```

③ : 채팅서버 UpdateThread의 메시지 처리 프로세스 중 메시지 타입이 REQ_LOGIN인 경우입니다. 유효 클라이언트 여부를 판단하기위해 AuthThread의 Auth Queue에 큐잉합니다.

```
case en_PACKET_CS_CHAT_REQ_LOGIN: {
    // session id를 통해 player 컨테이너에서 player를 찾습니다.
    auto iter = playerMap.find(sessionId);
    if (iter == playerMap.end()) {
        PacketBuffer::Free(csContentsPacket);
        break;
    }
    Player* p_player = iter->second;

    // 중략 ( 중복로그인을 판단하여, Disconnect 합니다.)

    AccountToken* p_at = tokenPool.Alloc();
    // 중략 (수신 패킷에서 Player 정보, 토큰정보를 꺼내어 player와 p_at(Auth Job)을 셋팅합니다.)

    // AuthThread의 Auth Queue에 클라이언트 유효판단을 위해 큐잉합니다.
    tokenQ.Enqueue(p_at);
    SetEvent(authEvent);

    PacketBuffer::Free(csContentsPacket);
    break;
}
```

④, ⑤ : Redis에서 Token을 조회합니다. 클라이언트에서 송신한 로그인 요청 메시지에 포함된 토큰 과 Redis에 저장된 토큰이 일치한다면 유효 클라이언트로 판단하여 LOGIN_SUCCESS Type의 Job을 생성하여 UpdateThread의 Job Queue에 큐잉합니다. (실패 시 LOGIN_FAIL Type을 큐잉합니다.)

```
for (;;) {
    // 이벤트 Signal을 대기합니다.
    WaitForSingleObject(authEvent, INFINITE);
    if (authStop) break;

    // 이벤트 Signal로 인해 깨어났다면, Queue에 쌓인 모든 Job을 처리합니다.
    while (tokenQ.Dequeue(&p_at)) {
        char chattingKey[100];
        snprintf(chattingKey, 100, "%lld.chatting", p_at->accountNo);

        // Redis에서 Key를 조회합니다.
        std::future<cpp_redis::reply> futureReply = redisClient.get(chattingKey);
        redisClient.sync_commit();
        cpp_redis::reply reply = futureReply.get();
        if (reply.is_string()) memcpy((char*)&redisToken, reply.as_string().c_str(), sizeof(Token));

        // 토큰의 일치를 판단합니다. (토큰 일치 시 유효 세션입니다.)
        // 성공 시 SUCCESS, 실패 시 FAIL Type의 Job을 생성하여 UpdateThread의 JobQueue에 큐잉합니다.
        if (0 == strncmp(redisToken.buf, p_at->token.buf, 64)) {
            redisClient.del({ chattingKey });
            redisClient.sync_commit();
            JobQueuing(p_at->sessionId, JOB_TYPE_CLIENT_LOGIN_SUCCESS);
        }
        else {
            JobQueuing(p_at->sessionId, JOB_TYPE_CLIENT_LOGIN_FAIL);
        }
        tokenPool.Free(p_at);
    }
}
```

⑥ : 위의 작업이 성공적으로 진행된 클라이언트에 한하여, 컨텐츠 메시지를 주고받습니다. (메시지 타입과 메시지 타입 별 처리 프로세스는 다음장에 이어서 설명하겠습니다.)

메시지 타입 별 처리 프로세스

0. UpdateThread

```
for (::) {
    WaitForSingleObject(updateEvent, INFINITE);
    if (updateStop) break;
    for (::) {
        if (jobQ.GetUseCount() < 1) break;
        jobQ.Dequeue(&p_job);

        SessionId sessionId = p_job->sessionId;
        PacketBuffer* csContentsPacket = p_job->p_packet;

        switch (p_job->type) {
            case JOB_TYPE_CLIENT_JOIN:
            case JOB_TYPE_CLIENT_LEAVE:
            case en_PACKET_CS_CHAT_REQ_LOGIN:
            case en_PACKET_CS_CHAT_REQ_SECTOR_MOVE:
            case en_PACKET_CS_CHAT_REQ_MESSAGE:
            case JOB_TYPE_CLIENT_LOGIN_SUCCESS:
            case JOB_TYPE_CLIENT_LOGIN_FAIL:
            default:
                }
        jobPool.Free(p_job);
    }
}
```

UpdateThread의 대략적인 모양은 위와 같습니다, Job Queue에서 디큐잉한 Job을 Type에 맞게 처리합니다.

1. Job Type 'JOB_TYPE_CLIENT_JOIN'

```
case JOB_TYPE_CLIENT_JOIN: {
    Player* p_player = playerPool.Alloc();
    p_player->Set(sessionId);
    playerMap.insert({ sessionId, p_player });
    break;
}
```

OnClientJoin()에서 직렬화 되는 Job입니다. 접속한 클라이언트를 관리하기 위해 Player의 컨테이너 PlayerMap 등록하는 작업을 수행합니다. (Player는 사용자단에서 관리하는 세션 단위입니다.)

2. Job Type 'JOB_TYPE_CLIENT_LEAVE'

```
case JOB_TYPE_CLIENT_LEAVE: {
    // Player 검색
    auto iter = playerMap.find(sessionId);
    Player* p_player = iter->second;

    // 컨테이너에서 삭제
    playerMap.erase(iter);

    // Sector 에서 삭제
    if (!p_player->sectorPos.IsValid()) {
        sectorSet[p_player->sectorPos.y][p_player->sectorPos.x].erase(p_player);
    }

    // Player 반환
    if (p_player->isLogin) playerCount--;
    p_player->Reset();
    playerPool.Free(p_player);
    break;
}

void ChatServerST::OnClientLeave(SessionId sessionId) {
    JobQueuing(sessionId, JOB_TYPE_CLIENT_LEAVE);
}
```

OnClientLeave()에서 직렬화 되는 Job입니다. 연결 해제된 Player를 주 컨테이너 'PlayerMap'과 부 컨테이너 'SectorSet'에서 erase하는 작업을 수행합니다.

3. Job Type 'en_PACKET_CS_CHAT_REQ_LOGIN'

```
case en_PACKET_CS_CHAT_REQ_LOGIN: {
    // session id를 통해 player 컨테이너에서 player를 찾습니다.
    auto iter = playerMap.find(sessionId);
    if (iter == playerMap.end()) {
        PacketBuffer::Free(csContentsPacket);
        break;
    }
    Player* p_player = iter->second;

    // 종락 ( 중복로그인을 판단하여, Disconnect 합니다.)

    AccountToken* p_at = tokenPool.Alloc();
    // 종락 (수신 패킷에서 Player 정보, 토큰정보를 꺼내어 player와 p_at(Auth Job)을 셋팅합니다.)

    // AuthThread의 Auth Queue에 클라이언트 유효판단을 위해 큐잉합니다.
    tokenQ.Enqueue(p_at);
    SetEvent(authEvent);

    PacketBuffer::Free(csContentsPacket);
    break;
}
```

Worker의 OnRecv()에서 직렬화 되는 Job입니다. 유효 클라이언트 여부를 판단하기위해 AuthThread의 Auth Queue에 큐잉합니다. (이전, 'MMO 채팅 서버 접속 및 메시지 처리 과정' 챕터에서도 기재했던 내용입니다.)

4. Job Type 'en_PACKET_CS_CHAT_REQ_SECTOR_MOVE'

```
case en_PACKET_CS_CHAT_REQ_SECTOR_MOVE: {
    // 종락 (session id를 통해 player 컨테이너에서 player를 찾습니다.)
    // 종락 (해당 메시지를 보낸 클라이언트가 로그인 상태가 아닐 시 Disconnect 합니다.)
    // 수신받은 패킷에서 accountNo, Sector x/y의 데이터를 꺼냅니다.
    INT64 accountNo;
    Sector curSector;
    try {
        *csContentsPacket >> accountNo;
        *csContentsPacket >> curSector.x;
        *csContentsPacket >> curSector.y;
    }
    catch (const PacketException& e) {
        Disconnect(sessionId);
        PacketBuffer::Free(csContentsPacket);
        break;
    }

    // 종락 (클라이언트가 송신한 패킷에 담긴 섹터 x,y 좌표가 유효하지 않은 좌표일 경우 Disconnect 합니다.)
    // 플레이어의 이전위치 Sector에서 플레이어를 삭제하고, 이동 요청 Sector로 최신화 합니다.
    if (!p_player->sectorPos.IsValid()) {
        sectorSet[p_player->sectorPos.y][p_player->sectorPos.x].erase(p_player);
    }
    p_player->SetSector(curSector);
    sectorSet[curSector.y][curSector.x].insert(p_player);

    // 최신화된 Sector 좌표를 포함한 RES_SECTOR_MOVE 패킷을 회신합니다.
    PacketBuffer* p_packet = PacketBuffer::Alloc();
    *p_packet << (WORD)en_PACKET_CS_CHAT_RES_SECTOR_MOVE;
    *p_packet << (INT64)accountNo;
    *p_packet << (WORD)curSector.x;
    *p_packet << (WORD)curSector.y;
    SendPacket(sessionId, p_packet);
    PacketBuffer::Free(p_packet);
    PacketBuffer::Free(csContentsPacket);
    break;
}
```

Worker의 OnRecv()에서 직렬화 되는 Job입니다. Sector 이동 요청 메시지로, 채팅서버의 콘텐츠 메시지입니다. Player의 이전 좌표 SectorSet에서 Player를 erase하고, 이동 요청 좌표의 SectorSet에 Player를 insert하여 Player의 Sector 이동을 반영합니다.

5. Job Type 'en_PACKET_CS_CHAT_REQ_MESSAGE'

```
case en_PACKET_CS_CHAT_REQ_MESSAGE: {
    // 중략 (session id를 통해 player 컨테이너에서 player를 찾습니다.)
    // 중략 (해당 메시지를 보낸 클라이언트가 로그인 상태가 아닐 시 Disconnect 합니다.)
    // 수신받은 패킷에서 accountNo와 채팅 메시지를 꺼냅니다.
    INT64 accountNo;
    WORD msgLen;
    WCHAR msg[MAX_MSG]; // null 미포함
    try {
        *csContentsPacket >> accountNo;
        *csContentsPacket >> msgLen;
        csContentsPacket->GetData((char*)msg, msgLen);
    }
    catch (const PacketException& e) {
        LOG("ChatServerST", LOG_LEVEL_WARN, "Disconnect // impossible : >> chatting packet");
        Disconnect(sessionId);
        PacketBuffer::Free(csContentsPacket);
        PRO_END("UpdateFunc::en_PACKET_CS_CHAT_REQ_MESSAGE");
        break;
    }
    msg[msgLen / 2] = 0;

    // 채팅 패킷을 생성합니다. 채팅 패킷에는 채팅 메시지와 채팅 메시지를 송신한 Player의 id, nickname도 포함됩니다.
    PacketBuffer* p_packet = PacketBuffer::Alloc();
    *p_packet << (WORD)en_PACKET_CS_CHAT_RES_MESSAGE;
    *p_packet << (INT64)accountNo;
    p_packet->PutData((char*)p_player->id, ID_LEN * 2);
    p_packet->PutData((char*)p_player->nickname, NICKNAME_LEN * 2);
    *p_packet << (WORD)msgLen;
    p_packet->PutData((char*)msg, msgLen);

    // 위에서 만든 채팅 패킷을 채팅 메시지를 송신한 Player가 속한 Sector와 인접한 Sector에 송신합니다.
    SendSectorAround(p_player, p_packet);
    PacketBuffer::Free(p_packet);
    PacketBuffer::Free(csContentsPacket);
    break;
}
```

Worker의 OnRecv()에서 직렬화 되는 Job입니다. 채팅 메시지로, 채팅서버의 콘텐츠 메시지입니다. 채팅 메시지를 송신한 Player가 속한 Sector와 인접한 Sector에 포함되는 플레이어들에게 채팅 메시지를 송신합니다.

6. Job Type 'JOB_TYPE_CLIENT_LOGIN_SUCCESS'

```
case JOB_TYPE_CLIENT_LOGIN_SUCCESS: {
    // 중략 (session id를 통해 player 컨테이너에서 player를 찾습니다.)

    // 로그인 성공
    p_player->isLogin = true;
    playerCount++;

    // 로그인 성공 패킷 회신
    PacketBuffer* p_packet = PacketBuffer::Alloc();
    *p_packet << (WORD)en_PACKET_CS_CHAT_RES_LOGIN;
    *p_packet << (BYTE)p_player->isLogin;
    *p_packet << (INT64)p_player->accountNo;
    SendPacket(sessionId, p_packet);
    PacketBuffer::Free(p_packet);
    break;
}
```

Auth Thread에서 직렬화 되는 Job으로, Auth Thread에서 유효한 Player로 판단한 경우 직렬화됩니다. Player의 로그인 상태를 true로 전환하고 로그인 성공 패킷을 회신합니다.

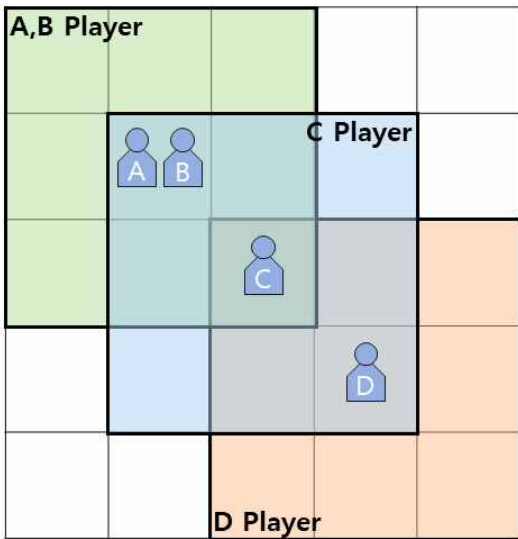
7. Job Type 'JOB_TYPE_CLIENT_LOGIN_FAIL'

```
case JOB_TYPE_CLIENT_LOGIN_FAIL: {  
    // 종락 (session id를 통해 player 컨테이너에서 player를 찾습니다.)  
    // 로그인 실패 패킷 회신  
    PacketBuffer* p_packet = PacketBuffer::Alloc();  
    *p_packet << (WORD)en_PACKET_CS_CHAT_RES_LOGIN;  
    *p_packet << (BYTE>false;  
    *p_packet << (INT64)p_player->accountNo;  
    SendPacket(sessionId, p_packet);  
    PacketBuffer::Free(p_packet);  
    break;  
}
```

Auth Thread에서 직렬화 되는 Job으로, Auth Thread에서 유효하지 않은 Player로 판단한 경우 직렬화됩니다. 로그인 실패 패킷을 회신합니다.

MMO 채팅서버의 Sector

채팅서버에서는 Sector 개념을 사용합니다. 채팅 메시지 송신 단위를 채팅 메시지 요청 Player와 주변 Player들로 제한하기 위함입니다. 채팅 메시지 수신 시 서버는 Player가 속한 Sector의 Player들과 인접한 Sector에 속한 Player들에게 채팅 메시지를 송신하며, 섹터 이동 메시지 수신 시 Player의 Sector를 최신화합니다.



왼쪽의 그림에서 한칸 한칸이 Sector입니다.

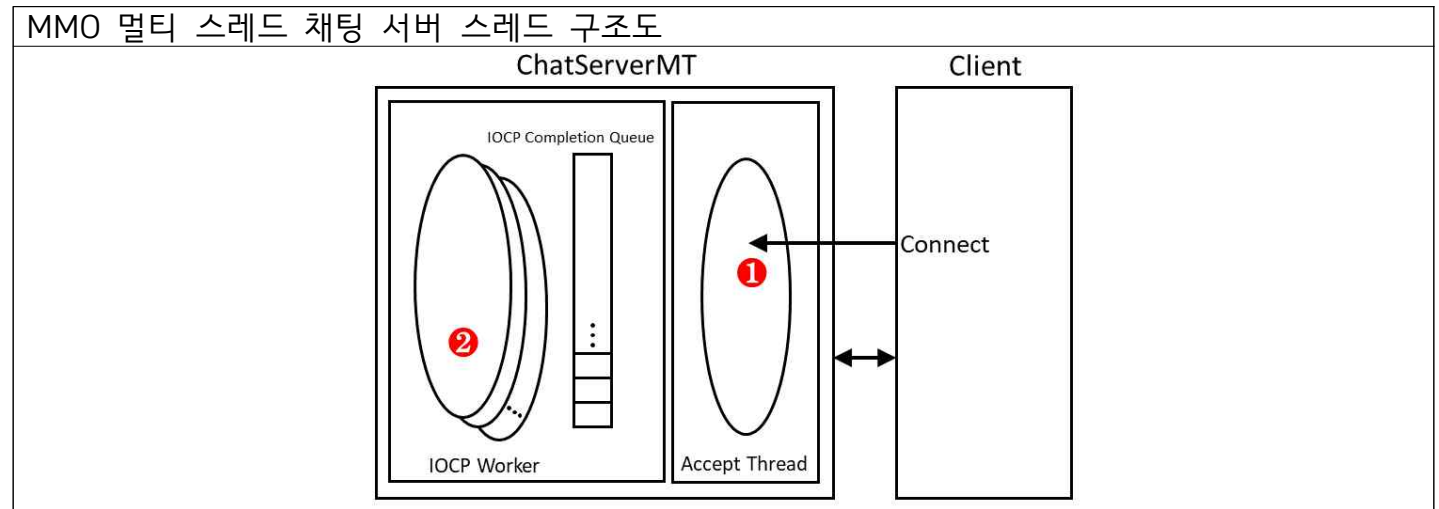
Sector는 Player의 보조 컨테이너로, 서버에서 채팅 메시지 송신 시 Sector에서 Player를 검색하여 채팅 메시지를 송신합니다.

왼쪽의 경우 A Player가 채팅 메시지를 보냈다고 할 때, 서버에서는 A, B, C Player에게 A가 보낸 채팅 메시지를 송신합니다. C Player의 채팅 메시지의 경우 A, B, C, D의 모든 Player에게 채팅 메시지가 송신되며, D Player의 경우 C, D Player에게 채팅 메시지가 송신됩니다.

정리하자면 A, B Player의 채팅 메시지를 D Player가 받을 수 없고, D Player의 채팅 메시지는 A, B Player가 받을 수 없으며, C Player의 채팅 메시지는 모든 Player가 C Player의 인접한 Sector에 위치하므로 모두에게 송신됩니다.

MMO 멀티 스레드 채팅 서버

Update Thread 등 외부 스레드로 메시지 처리를 우회하지 않고, OnRecv()가 호출되는 Worker Thread에서 메시지 즉시 처리합니다. 싱글 스레드 채팅서버와 퍼포먼스를 비교하기 위한 목적의 서버로, Auth 과정이 생략되어 있습니다. (github : https://github.com/dkdldjswkd/cpp_Project/tree/main/01%20Network/ChatServerMT)



❶ : 클라이언트의 접속으로 인해 AcceptThread에서 채팅서버 구현 시 오버라이딩한 OnClientJoin()가 호출됩니다. OnClientJoin()에서는 접속한 클라이언트를 관리하기 위해 Player의 컨테이너 PlayerMap 등록하는 작업을 수행합니다.

```
void ChatServerMT::OnClientJoin(SessionId sessionId) {
    Player* p_player = playerPool.Alloc();
    p_player->Set(sessionId);

    playerMapLock.Lock();
    playerMap.insert({ sessionId, p_player });
    playerMapLock.Unlock();
}
```

❷ : 클라이언트의 메시지 송신으로 인해 Worker에서 채팅서버 구현 시 오버라이딩한 OnRecv()가 호출됩니다. OnRecv()의 대략적인 모양은 아래와 같습니다. Packet Type에 맞게 메시지를 처리합니다. (멀티스레드 채팅 서버는 싱글 스레드 채팅서버와 퍼포먼스 비교 목적으로 구현된 서버로, Auth 과정이 생략되어 있습니다)

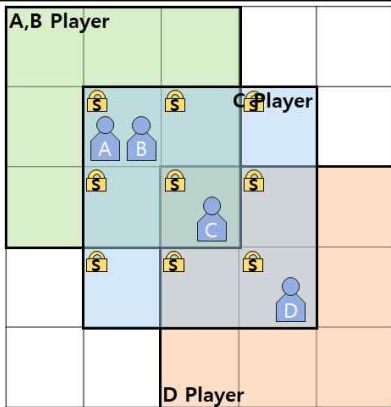
```
void ChatServerMT::OnRecv(SessionId sessionId, PacketBuffer* csContentsPacket) {
    WORD type;
    try {
        *csContentsPacket >> type;
    }
    catch (const PacketException& e) {
        Disconnect(sessionId);
        return;
    }
    switch (type) {
        case en_PACKET_CS_CHAT_REQ_LOGIN:
        case en_PACKET_CS_CHAT_REQ_SECTOR_MOVE:
        case en_PACKET_CS_CHAT_REQ_MESSAGE:
        default:
    }
    InterlockedIncrement((LONG*)&updateCount);
}
```

메시지 타입 별 처리 프로세스

0. 송신 메시지 동기화

```
void ChatServerMT::SendSectorAround(Player* p_player, PacketBuffer* send_packet) {
    // 9방향 Lock
    for (int i = 0; i < p_player->sectorAround.count; i++) {
        sectorLock[p_player->sectorAround.around[i].y][p_player->sectorAround.around[i].x].SharedLock();
    }
    for (int i = 0; i < p_player->sectorAround.count; i++) {
        SendSector(send_packet, p_player->sectorAround.around[i]);
    }
    // 9방향 Unlock
    for (int i = 0; i < p_player->sectorAround.count; i++) {
        sectorLock[p_player->sectorAround.around[i].y][p_player->sectorAround.around[i].x].ReleaseSharedLock();
    }
}

void ChatServerMT::SendSector(PacketBuffer* send_packet, Sector sector) {
    for (auto iter = sectorSet[sector.y][sector.x].begin(); iter != sectorSet[sector.y][sector.x].end(); iter++) {
        Player* p_player = *iter;
        if (p_player->isLogin) {
            SendPacket(p_player->sessionId, send_packet);
        }
    }
}
```



왼쪽의 그림은 Player C의 REQ_MESSAGE 메시지 처리시 호출되는 위의 함수 SendSectorAround()의 작동을 이미지 화 한 것입니다. Player C가 속한 Sector와 인접 Sector, 최대 9개의 SharedLock을 획득하고 Sector에 속한 Player들에게 Player C의 채팅 메시지를 송신하는 방법을 사용했습니다.

1. Packet Type 'en_PACKET_CS_CHAT_REQ_MESSAGE'

```
case en_PACKET_CS_CHAT_REQ_MESSAGE: {
    // 수신받은 패킷에서 accountNo와 채팅 메시지를 꺼냅니다.
    INT64 accountNo;
    WORD msgLen;
    WCHAR msg[MAX_MSG]; // null 미포함
    try {
        *csContentsPacket >> accountNo;
        *csContentsPacket >> msgLen;
        csContentsPacket->GetData((char*)msg, msgLen);
        msg[msgLen / 2] = 0;
    }
    catch (const PacketException& e) {
        Disconnect(sessionId);
        return;
    }

    // Player를 검색합니다.
    playerMapLock.SharedLock();
    Player* p_player = playerMap.find(sessionId)->second;
    playerMapLock.ReleaseSharedLock();

    // 채팅 메시지를 생성하고 SendSectorAround()를 호출하여, 인접한 Player들에게 채팅메시지를 송신합니다.
    PacketBuffer* p_packet = PacketBuffer::Alloc();
    *p_packet << (WORD)en_PACKET_CS_CHAT_RES_MESSAGE;
    *p_packet << (INT64)accountNo;
    p_packet->PutData((char*)p_player->id, ID_LEN * 2);
    p_packet->PutData((char*)p_player->nickname, NICKNAME_LEN * 2);
    *p_packet << (WORD)msgLen;
    p_packet->PutData((char*)msg, msgLen);
    SendSectorAround(p_player, p_packet);
    PacketBuffer::Free(p_packet);
    break;
}
```

채팅패킷을 생성하고, SendSectorAround()를 호출하여 인접한 Player들에게 채팅 메시지를 송신합니다.

2. Packet Type 'en_PACKET_CS_CHAT_REQ_SECTOR_MOVE'

```

case en_PACKET_CS_CHAT_REQ_SECTOR_MOVE: {
    // 중략 (수신받은 패킷에서 데이터를 꺼냅니다.)

    // 중략 (이동 섹터 좌표가 유효한 좌표인지 검사하여, 유효하지 않은 좌표라면 Disconnect 합니다.)

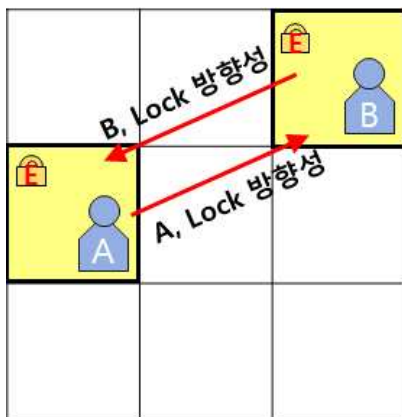
    // Player를 검색합니다.
    playerMapLock.SharedLock();
    Player* p_player = playerMap.find(sessionId)->second;
    playerMapLock.ReleaseSharedLock();

    // Player의 Prev Sector를 백업하고, Player가 알고있는 자신의 Sector를 최신화합니다.
    Sector prevSector = p_player->sectorPos;
    p_player->SetSector(curSector);

    // 최초로 수신한 SECTOR_MOVE 패킷인 경우입니다. SECTOR 이동이 아닌, 등록 작업을 수행합니다.
    if (prevSector.IsValid()) {
        sectorLock[curSector.y][curSector.x].Lock();
        sectorSet[curSector.y][curSector.x].insert(p_player);
        sectorLock[curSector.y][curSector.x].Unlock();
    }
    // SECTOR 이동 작업을 수행합니다.
    else if (prevSector != p_player->sectorPos) {
        // 동기화 객체의 주소 크기 순서로 Lock을 획득 합니다. (데드락을 방지하기 위함입니다.)
        if (&sectorLock[prevSector.y][prevSector.x] < &sectorLock[curSector.y][curSector.x]) {
            sectorLock[prevSector.y][prevSector.x].Lock();
            sectorLock[curSector.y][curSector.x].Lock();
        }
        else {
            sectorLock[curSector.y][curSector.x].Lock();
            sectorLock[prevSector.y][prevSector.x].Lock();
        }
        // Player Sector 이동처리
        sectorSet[prevSector.y][prevSector.x].erase(p_player);
        sectorSet[curSector.y][curSector.x].insert(p_player);
        sectorLock[prevSector.y][prevSector.x].Unlock();
        sectorLock[curSector.y][curSector.x].Unlock();
    }

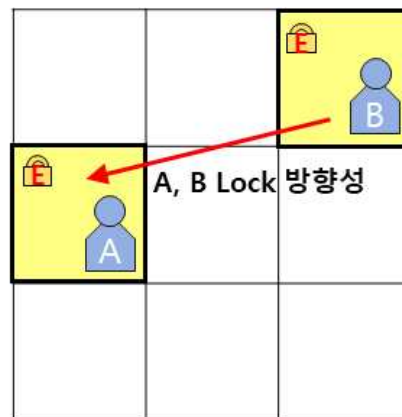
    // SECTOR 이동 결과 패킷을 송신합니다.
    PacketBuffer* p_packet = PacketBuffer::Alloc();
    *p_packet << (WORD)en_PACKET_CS_CHAT_RES_SECTOR_MOVE;
    *p_packet << (INT64)accountNo;
    *p_packet << (WORD)curSector.x;
    *p_packet << (WORD)curSector.y;
    SendPacket(sessionId, p_packet);
    PacketBuffer::Free(p_packet);
    break;
}

```



X

데드락 발생



O

데드락 발생 X

SECTOR_MOVE를 처리하기 위해서는 '이동 전 Sector, 이동 후 Sector' 두 개 Sector의 락을 획득하는 과정이 필요합니다. 락을 획득하는 과정에서 방향성에 대한 규칙이 없다면 왼쪽 그림처럼 데드락이 발생할 수 있다고 판단했습니다. 데드락을 방지하기 위해 '동기화 객체 주소 크기 순서대로 락을 획득'이라는 방향성 규칙을 두어, 락을 획득하는 방법을 사용했습니다.

싱글/멀티 스레드 채팅 서버 성능 비교

각각의 채팅 서버는 여러 가지 항목의 모니터링 데이터를 모니터링 서버로 송신하였으며, 모니터링 서버에서는 수신받은 모니터링 데이터들을 DB에 기록했습니다.

싱글 스레드 채팅 서버 스레드 구조 (Auth Off)

ChatServerST

Job Queue

IOCP Completion Queue

Update Thread

IOCP Worker

Accept Thread

Client

Connect

멀티 스레드 채팅 서버 스레드 구조

ChatServerMT

IOCP Completion Queue

IOCP Worker

Accept Thread

Client

Connect

싱글 스레드 채팅 서버 초당 메시지 처리량

	no	logtime	serverno	type	avr	min	max
	123	2023-04-13 19:03:46	1	35	20115	0	45022
	136	2023-04-13 19:08:45	1	35	26479	16734	54872
	149	2023-04-13 19:13:45	1	35	27096	16243	77621
	162	2023-04-13 19:18:45	1	35	31299	19765	53849
	175	2023-04-13 19:23:45	1	35	31390	19902	61565
	188	2023-04-13 19:28:46	1	35	31361	20436	58205
	201	2023-04-13 19:33:45	1	35	32637	19937	61184
	214	2023-04-13 19:38:45	1	35	29698	16742	56631
	227	2023-04-13 19:43:45	1	35	31553	18780	55910
	240	2023-04-13 19:48:45	1	35	31089	19131	55387
	253	2023-04-13 19:53:45	1	35	30794	20437	65538
	266	2023-04-13 19:58:45	1	35	31681	19435	60088
	279	2023-04-13 20:03:45	1	35	30085	21000	58241

멀티 스레드 채팅 서버 초당 메시지 처리량

	no	logtime	serverno	type	avr	min	max
▶	305	2023-04-13 20:21:26	2	35	33525	0	51294
	317	2023-04-13 20:26:26	2	35	49848	30435	92603
	329	2023-04-13 20:31:26	2	35	48516	31109	71993
	341	2023-04-13 20:36:26	2	35	49665	31603	83142
	353	2023-04-13 20:41:26	2	35	49967	29512	72716
	365	2023-04-13 20:46:27	2	35	48830	30780	71865
	377	2023-04-13 20:51:26	2	35	48795	30610	72332
	389	2023-04-13 20:56:26	2	35	48610	30076	73193
	401	2023-04-13 21:01:26	2	35	48319	29435	71573
	413	2023-04-13 21:06:26	2	35	48504	31050	72652
	425	2023-04-13 21:11:26	2	35	48686	31144	97739
	437	2023-04-13 21:16:27	2	35	47106	31276	71245
	449	2023-04-13 21:21:26	2	35	48004	27217	72687

테스트 환경 : LAN, 더미 머신 2대(머신 당 더미 5000명), 재접속 x, 서버 코어 개수 4, 하이퍼 스레딩 지원 x, 더미는 이전 보낸 메시지의 결과를 받지 않더라도 메시지를 계속 송신합니다.

싱글 스레드 채팅서버의 경우 Update Thread 1개, IOCP Worker Thread 3개로 설정했습니다.
멀티 스레드 채팅서버의 경우 Worker Thread 4개로 설정했습니다.

멀티 스레드 채팅 서버는 메시지 처리를 다수의 스레드에서 병렬적으로 수행하기 때문에 예상대로 처리량이 더 높게 측정되었습니다. 하지만, 메시지 처리 스레드 개수(4배) 만큼의 차이가 나타나지는 않았습니다. 이유는 두가지로 추측해보았습니다.

- 1. 멀티 스레드 채팅 서버 작업 시 동기화 객체에 대한 경합발생
- 2. 네트워크 라이브러리 API 'SendPacket()'에서 PQCS()를 사용하여 Worker로 Send call 오버헤드 전달

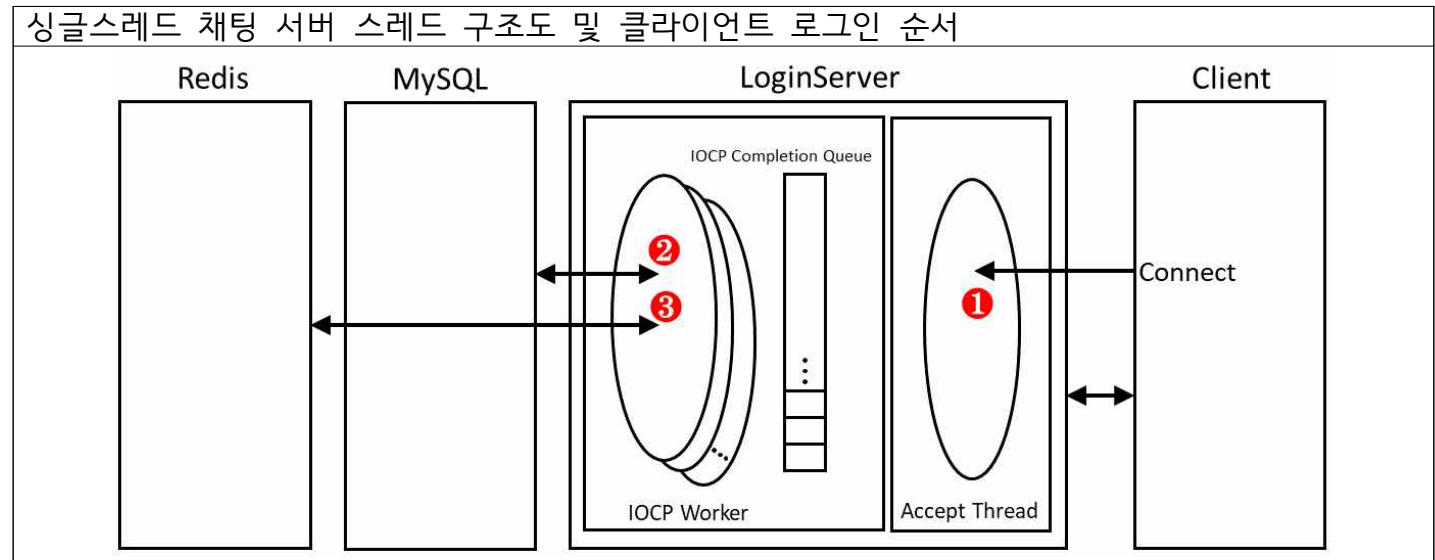
퍼포먼스를 비교하자면 단연, 병렬적으로 작업을 하는 멀티 스레드 서버가 더 효율적이라 생각합니다. 다만, 싱글 스레드 채팅 서버의 경우 'UpdateThread' 한 개의 스레드로 Job이 직렬화되기 때문에 동기화 고려가 필요하지 않았고, 멀티 스레드 채팅 서버보다 구현이 쉽고 빨랐습니다.

또한, 싱글 스레드 채팅 서버 퍼포먼스를 개선하며 사용한 'Send call 우회'의 방법으로 인해 싱글 스레드 구조의 경우 비동기적으로 수행가능한 작업을 다른 스레드로 우회하는 방법을 사용한다면 단점인 성능의 부분도 많은 개선이 가능하다는 점을 느꼈습니다.

정리하자면, 성능과 반응성이 중요한 경우 멀티스레드 서버 구조를 채택하는 것이 좋다고 판단됩니다. 그러나, 처리해야할 작업이 많지않고 안정성이 중요하다면 싱글 스레드 구조도 채택을 고려해볼 수 있다고 생각합니다.

로그인 서버

유효한 로그인 요청에 대해 토큰 발급 및 로그인 성공 패킷을 회신하는 로그인 서버입니다.
 (github : https://github.com/dkdldjswkd/cpp_Project/tree/main/01%20Network/LoginServer)



① : 클라이언트의 접속으로 인해 AcceptThread에서 로그인 서버 구현 시 오버라이딩한 OnClientJoin()가 호출됩니다. OnClientJoin()에서는 접속한 클라이언트를 관리하기 위해 컨테이너에 등록하는 작업을 수행합니다.

```
void LoginServer::OnClientJoin(SessionId sessionId) {
    User* p_player = userPool.Alloc();
    p_player->Set(sessionId);

    userMapLock.Lock();
    userMap.insert({ sessionId, p_player });
    userMapLock.Unlock();
}
```

②, ③ : 클라이언트의 메시지 송신으로 인해 Worker에서 로그인 서버 구현 시 오버라이딩한 OnRecv()가 호출됩니다. OnRecv()의 대략적인 모양은 아래와 같습니다. Packet Type은 'REQ_LOGIN' 한 개가 존재합니다.

```
void LoginServer::OnRecv(SessionId sessionId, PacketBuffer* csContentsPacket) {
    WORD type;
    try {
        *csContentsPacket >> type;
    }
    catch (const PacketException& e) {
        LOG("LoginServer", LOG_LEVEL_WARN, "Disconnect // impossible : >>");
        Disconnect(sessionId);
        return;
    }
    switch (type) {
        case en_PACKET_CS_LOGIN_REQ_LOGIN:
    }
}
```

메시지 타입 별 처리 프로세스

1. Packet Type 'en_PACKET_CS_LOGIN_REQ_LOGIN'

```
case en_PACKET_CS_LOGIN_REQ_LOGIN: {
    // 중략 (수신받은 패킷에서 토큰과 accountNo를 꺼냅니다.)
    // 인증 절차
    sqlResult = p_connectorTLS->Query("SELECT sessionkey, userid, usernick FROM sessionkey S
    JOIN account A ON S.accountno = A.accountno where S.accountno = %lld", accountNo);
    sqlRow = mysql_fetch_row(sqlResult);

    // 튜플이 존재하지 않음
    if (NULL == sqlRow) {
        Disconnect(sessionId);
        mysql_free_result(sqlResult);
        return;
    }

    // Session Key 대조
    // (일치 가정)

    // id, nick 추출
    WCHAR id[20];
    UTF8ToUTF16(sqlRow[1], id);
    WCHAR nickname[20];
    UTF8ToUTF16(sqlRow[2], nickname);

    // result 반환
    mysql_free_result(sqlResult);

    // 중략 (로그인 응답 패킷에 신어보낼 채팅서버 IP,Port를 셋팅합니다.)
    // 로그인 응답 패킷 생성합니다.
    PacketBuffer* p_packet = PacketBuffer::Alloc();
    *p_packet << (WORD)en_PACKET_CS_LOGIN_RES_LOGIN;
    *p_packet << (INT64)accountNo;
    *p_packet << (BYTE)dfLOGIN_STATUS_OK;
    p_packet->PutData((char*)id, 40);
    p_packet->PutData((char*)nickname, 40);
    p_packet->PutData((char*)GameServerIP, 32);
    *p_packet << (USHORT)GameServerPort;
    p_packet->PutData((char*)ChatServerIP, 32);
    *p_packet << (USHORT)ChatServerPort;

    // Token DB에 인증토큰을 삽입합니다. (토큰의 생명은 10초입니다.)
    char GameKey[100];
    char chattingKey[100];
    snprintf(chattingKey, 100, "%lld.chatting", accountNo);
    snprintf(GameKey, 100, "%lld.game", accountNo);
    connectorRedis.setex(chattingKey, 10, (char*)&sessoinKey);
    connectorRedis.setex(GameKey, 10, (char*)&sessoinKey);
    connectorRedis.sync_commit();

    // 로그인 응답 패킷 송신
    SendPacket(sessionId, p_packet);
    PacketBuffer::Free(p_packet);
    break;
}
```

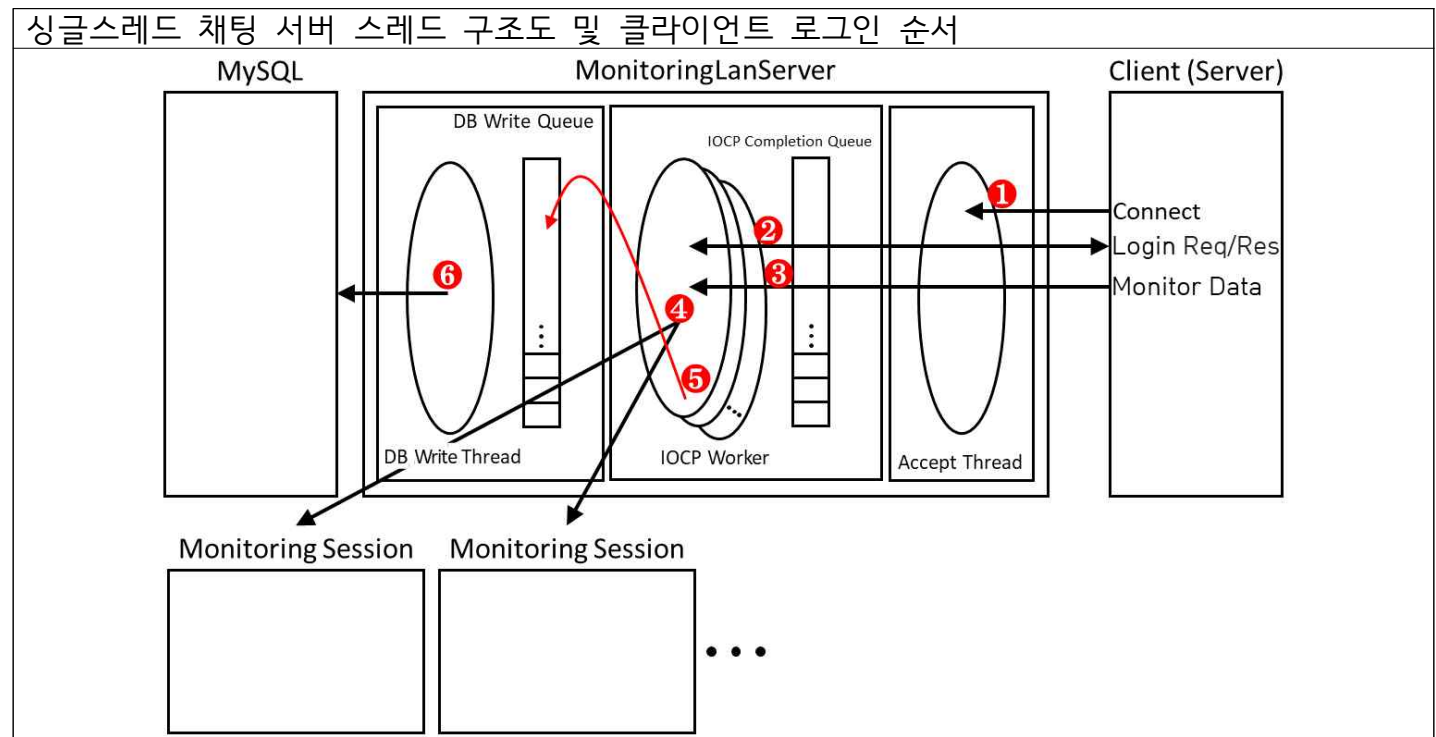
account DB에서 로그인 요청 클라이언트의 accountNo에 대한 sessoinKey를 조회합니다.

로그인 요청 패킷의 Session Key와 일치한다면 유효한 클라이언트로 판단하여, 10초의 유효기간을 갖는 로그인 인증 토큰을 발급하여 Token DB(Redis)에 삽입합니다. 마지막으로, 클라이언트에게 채팅서버 IP, port 정보와 발급 토큰이 담긴 로그인 응답 패킷을 송신합니다.

모니터링 서버

모니터 대상 서버로부터 모니터링 데이터를 받아 DB에 기록하며, 로그인된 모니터링 세션들에게 실시간 모니터링 정보를 전달합니다.

(github : https://github.com/dkldjswkd/cpp_Project/tree/main/01%20Network/MonitoringServer)



❶ : 클라이언트의 접속으로 인해 AcceptThread에서 모니터링 서버 구현 시 오버라이딩한 OnClientJoin()가 호출됩니다. OnClientJoin()에서는 접속한 클라이언트를 관리하기 위해 컨테이너에 등록하는 작업을 수행합니다.

```
void MonitoringLanServer::OnClientJoin(SessionId sessionId) {
    // User Alloc
    ServerSession* p_user = serverSessionPool.Alloc();
    p_user->Set(sessionId);

    // UserMap Insert
    serverSessionMapLock.Lock();
    serverSessionMap.insert({ sessionId, p_user });
    serverSessionMapLock.Unlock();
}
```

❷ ~ ❺ : 클라이언트의 메시지 송신으로 인해 Worker에서 모니터링 서버 구현 시 오버라이딩한 OnRecv()가 호출됩니다. OnRecv()의 대략적인 모양은 아래와 같습니다. Packet Type에 맞게 메시지를 처리합니다.

```
void MonitoringLanServer::OnRecv(SessionId sessionId, PacketBuffer* csContentsPacket){
    WORD type;
    try {
        *csContentsPacket >> type;
    }
    catch (const PacketException& e) {
        LOG("MonitoringServer", LOG_LEVEL_WARN, "Disconnect // impossible : >>");
        Disconnect(sessionId);
        return;
    }
    switch (type) {
        case en_PACKET_SS_MONITOR_LOGIN:
        case en_PACKET_SS_MONITOR_DATA_UPDATE:
        default:
    }
}
```

메시지 타입 별 처리 프로세스

1. Packet Type 'en_PACKET_SS_MONITOR_LOGIN'

```
case en_PACKET_SS_MONITOR_LOGIN: {
    int serverNo;

    // 중략 (패킷에서 serverNo를 꺼냅니다.)

    // User Find
    serverSessionMapLock.SharedLock();
    auto iter = serverSessionMap.find(sessionId);
    ServerSession* p_user = iter->second;
    serverSessionMapLock.ReleaseSharedLock();

    // 중략 ( 중복로그인을 판단하여, Disconnect 합니다.)

    // 로그인 성공
    p_user->Login(serverNo);
    return;
}
```

클라이언트에 해당하는 serverSession을 찾아, 로그인 처리합니다.

2. Packet Type 'en_PACKET_SS_MONITOR_DATA_UPDATE'

```
case en_PACKET_SS_MONITOR_DATA_UPDATE: {
    BYTE dataType; // 모니터링 데이터 타입
    int dataValue;
    int timeStamp;

    // 중략 (패킷에서 dataType, dataValue, timeStamp를 꺼냅니다.)

    // 중략 (User Find)

    // 중략 ( 중복로그인을 판단하여, Disconnect 합니다.)

    // * MonitoringNetSever로 모니터링 톨에게 모니터링 데이터 송신
    ((MonitoringNetServer*)monitoringNetServer)->BroadcastMonitoringData(p_user->serverNo, dataType, dataValue, timeStamp);
    SaveMonitoringData(p_user->serverNo, dataType, dataValue, timeStamp);
    return;
}

void MonitoringLanServer::SaveMonitoringData(int serverNo, int dataType, int data, int timeStamp){
    // serverNo에 해당하는 서버의 모니터링 정보를 찾아 최신화 합니다.
    MonitorData* p_data;
    // 중략 (serverNo에 해당하는 MonitorData Find)
    p_data = iter->second;
    p_data->updateData(data);

    // 5분 주기로 DB 스레드에 큐잉합니다.
    if (p_data->lastWriteTime + monitorLogTime <= timeStamp) {
        DBJobQueueing(p_data);
        p_data->Init(timeStamp);
    }
}
```

MONITOR_DATA_UPDATE 패킷의 모니터링 정보를 monitoringNetServer 객체(해당 객체는 모니터링 세션들과 연결하며, 실시간 모니터링 정보를 전달하는 서버 객체입니다)의 멤버함수 BroadcastMonitoringData()를 호출하여 모니터링 세션들에게 브로드 캐스트하며, SaveMonitoringData()를 호출하여 MONITOR_DATA_UPDATE 패킷을 보낸 서버의 모니터링 정보를 최신화 하고, 모니터링 데이터를 DB에 저장한지 일정 주기가 되었다면 DB에 모니터링 정보를 저장하기 위해 DB Queue에 모니터링 정보를 큐잉합니다.

3. MonitoringNetServer::BroadcastMonitoringData

```
void MonitoringNetServer::BroadcastMonitoringData(BYTE serverNo, BYTE dataType, int dataValue, int timeStamp){
    PacketBuffer* p_packet = PacketBuffer::Alloc();
    *p_packet << (WORD)en_PACKET_CS_MONITOR_TOOL_DATA_UPDATE;
    *p_packet << (BYTE)serverNo;
    *p_packet << (BYTE)dataType;
    *p_packet << (int)dataValue;
    *p_packet << (int)timeStamp;

    // 연결된 로그인 세션들에게 모니터링 정보 브로드 캐스트
    userMapLock.SharedLock();
    auto userMapEnd = userMap.end();
    for (auto iter = userMap.begin(); iter != userMapEnd; ++iter) {
        auto* p_user = iter->second;
        if (p_user->isLogin) {
            SendPacket(p_user->sessionID, p_packet);
        }
    }
    userMapLock.ReleaseSharedLock();

    PacketBuffer::Free(p_packet);
    return;
}
```

스레드 구조도의 ④에 해당하는 작업입니다. 모니터링 세션들에게 실시간 모니터링 정보를 브로드 캐스트합니다.

4. DB Write

```
void MonitoringLanServer::DBWriteFunc(){
    for (;;) {
        WaitForSingleObject(dbEvent, INFINITE);
        if (dbStop) break;
        for (;;) {
            if (dbQ.GetUseCount() < 1) break;
            MonitorData* p_logData;
            dbQ.Dequeue(&p_logData);

            // DB Write
            p_dbConnector->Query(
                "INSERT INTO logdb.monitorlog (serverno, type, avr, min, max) VALUES (%d, %d, %d, %d, %d)",
                p_logData->serverNo, p_logData->dataType, p_logData->sum / p_logData->count,
                p_logData->min, p_logData->max);

            monitorDataPool.Free(p_logData);
        }
    }
}
```

스레드 구조도의 ⑥에 해당하는 작업입니다. 'en_PACKET_SS_MONITOR_DATA_UPDATE' 패킷의 처리 과정에서 해당 스레드로 DB작업이 큐잉됩니다. '서버 번호, 모니터링 항목의 타입, 평균값, 최소값, 최대값'을 DB에 저장합니다. (저장되는 정보는 1초 주기로 수신받는 모니터링 데이터를 모니터링서버에서 5분간 모은 통계입니다.)

4. 락프리 자료구조

락프리 자료구조의 동작 방식은 다른 스레드에게 영향을 주지 않는 작업을 수행한 뒤, 작업 수행 전 자료구조 상태와 비교하여, 상태가 일치한다면 자료구조에 변형이 없었던 것이므로 작업을 적용시키는 방식의 스레드 동기화 기법을 사용하는 자료구조입니다.

락프리 스택

네트워크 라이브러리에서 사용하기 위해 구현한 락프리 스택 자료구조입니다.

(github : https://github.com/dkdldjswkd/cpp_Project/blob/main/00%20lib_jy/LFStack.h)

LFStack.h // LFStack::Push()

```
template <typename T>
void LFStack<T>::Push(T data) {
    // Create Node
    Node* pushNode = nodePool.Alloc();
    pushNode->Set();
    pushNode->data = data;

    for (;;) {
        DWORD64 copyTopStamp = topStamp;

        // top에 이어줌
        pushNode->next = (Node*)(copyTopStamp & useBitMask);

        // Stamp 추출 및 newTopStamp 생성
        DWORD64 nextStamp = (copyTopStamp + stampCount) & stampMask;
        DWORD64 newTopStamp = nextStamp | (DWORD64)pushNode;

        // 스택에 변화가 있었다면 다시시도
        if (copyTopStamp != InterlockedCompareExchange64((LONG64*)&topStamp, (LONG64)newTopStamp, (LONG64)copyTopStamp))
            continue;

        InterlockedIncrement((LONG*)&nodeCount);
        return;
    }
}
```

LFStack.h // LFStack::Pop()

```
template <typename T>
bool LFStack<T>::Pop(T* dst) {
    for (;;) {
        DWORD64 copyTopStamp = topStamp;
        Node* topClean = (Node*)(copyTopStamp & useBitMask);

        // Not empty!!
        if (topClean) {
            // Stamp 추출 및 newTopStamp 생성
            DWORD64 nextStamp = (copyTopStamp + stampCount) & stampMask;
            DWORD64 newTopStamp = nextStamp | (DWORD64)topClean->next;

            // 스택에 변화가 있었다면 다시시도
            if (copyTopStamp != InterlockedCompareExchange64((LONG64*)&topStamp, (LONG64)newTopStamp, (LONG64)copyTopStamp))
                continue;

            InterlockedDecrement((LONG*)&nodeCount);
            *dst = topClean->data;
            nodePool.Free(topClean);
            return true;
        }

        // empty!!
        else {
            return false;
        }
    }
}
```

락프리 스택 구현 중 발생한 문제와 원인 특정

(구현단계의 코드이기 때문에 위의 완성코드와 다릅니다.)

멀티스레드에서 발생한 문제를 단순 암산으로 파악하기는 매우 어렵기 때문에 각각의 스레드 코드 진행 상황을 파악하기 위해 코드의 진행마다 로그를 삽입했습니다. 코드 진행 후 최대한 빠르게 로그가 심어져야 하기에 메모리에 로그를 남기는 방법을 사용했습니다. 물론 작업의 진행순서와 100% 동일하게 로그가 심어지지 않을 수 있지만, 로그에 의존하여 원인파악을 했습니다. 논리적으로 불가능 하다고 생각하는 곳곳에 CRASH 코드를 심고, 프로세스의 덤프에 남은 메모리 로그를 분석하는 방법을 사용했습니다.

1. 디커밋 메모리 접근 (아래는 미완성 코드로, 구현 중 문제 발생 코드입니다.)

문제 발생 코드 // LFStack::Pop()

```
void LockFreeStack::Pop(BYTE flag) {
    char* wp;

    for (;;) {
        Node* copy_top = top;
        if (copy_top == nullptr) return;
        LOG(5);

        Node* new_top = copy_top->p_next;
        LOG(6);

        // 스택에 변화가 없었다면,
        if (copy_top == (Node*)InterlockedCompareExchange64((LONG64*)&top, (LONG64)new_top, (LONG64)copy_top)) {
            InterlockedDecrement((LONG*)&node_count);

            if (node_count < 0) { printf("팝 크래시 \n"); CRASH(); }
            delete copy_top;
            LOG(7);
            break;
        }
        else {
            LOG(8);
        }
    }
}
```

예외가 처리되지 않음

처리되지 않은 예외가 throw됨: 읽기 액세스 위반입니다.
copy_top이(가) 0x13D6AE7E010였습니다.

세부 정보 복사 | Live Share 세션을 시작합니다.

예외 설정

1-2. '디커밋 메모리 접근 원인' 특정 근거

메모리 로그	
<div>메모리 1</div> <div>주소: 0x0000013D6C9B2610</div> <div>0x0000013D6C9B2610 ① c5 10 e0 e7 6a 3d 01 00</div> <div>0x0000013D6C9B2618 e7 00 00 00 00 00 00 00</div> <div>0x0000013D6C9B2620 a5 10 e0 e7 6a 3d 01 00</div> <div>0x0000013D6C9B2628 d1 10 e0 e7 6a 3d 01 00</div> <div>0x0000013D6C9B2630 ② b5 10 e0 e7 6a 3d 01 00</div> <div>0x0000013D6C9B2638 a6 00 00 00 00 00 00 00</div> <div>0x0000013D6C9B2640 c6 00 00 00 00 00 00 00</div> <div>0x0000013D6C9B2648 e5 10 e0 e7 6a 3d 01 00</div> <div>0x0000013D6C9B2650 d2 00 00 00 00 00 00 00</div> <div>0x0000013D6C9B2658 e6 00 00 00 00 00 00 00</div> <div>0x0000013D6C9B2660 a8 00 00 00 00 00 00 00</div> <div>0x0000013D6C9B2668 d4 00 00 00 00 00 00 00</div> <div>0x0000013D6C9B2670 a5 90 3d e7 6a 3d 01 00</div> <div>0x0000013D6C9B2678 e8 00 00 00 00 00 00 00</div> <div>0x0000013D6C9B2680 ③ c7 00 00 00 00 00 00 00</div> <div>0x0000013D6C9B2688 d1 90 3d e7 6a 3d 01 00</div> <div>0x0000013D6C9B2690 a6 00 00 00 00 00 00 00</div>	<div>1. ① c5 10 e0 e7 6a 3d 01 00 가장 낮은 주소 1 Byte는 스레드 ID와 코드 진행을 표시 했습니다. c5는 '스레드 ID(c), 5번 코드 진행'을 뜻합니다. 그 후 메모리에는 주소 또는 값을 로깅했습니다. 이 경우 copy_top == 0x013d6ae7e010을 뜻하는 로그입니다.</div> <div>2. ② b5 10 e0 e7 6a 3d 01 00 b 스레드에서 5번 코드 진행, copy_top에 c 스레드와 같은 top을 읽어온 것으로 보입니다.</div> <div>3. ③ c7 00 00 00 00 00 00 00 // * POP c 스레드에서 스택에 변화가 없다고 판단하여 pop 작업을 진행했습니다. top 주소를 최신화 하고 copy_top을 delete 했습니다.</div> <div>4. b 스레드의 copy_top은 c 스레드에서 delete 했으므로, 디커밋 메모리 접근 예외가 발생되었습니다.</div>

2. ABA 이슈 발생 (아래는 미완성 코드로, 구현 중 문제 발생 코드입니다.)

문제 발생 코드 // LFStack::Push() (Pop()은 위의 코드와 동일)

```
void LFStack::Push(BYTE flag) {
    int v = rand() % 1000; 경과 시간 1ms 이하
    char* wp;
    Node* insert_node = new Node(v);
    LOG_(0);

    for (;;) {
        Node* copy_top = top;
        LOG_(1);

        insert_node->p_next = copy_top;
        LOG_(2);

        // 스택에 변화가 없었다면,
        if (copy_top == (Node*)InterlockedCompareExchange64((LONG64*)&top, (LONG64)insert_node, (LONG64)copy_top)) {
            InterlockedIncrement((LONG*)&node_count);
            LOG_(3);
            if (node_count > TEST_LOOP * THREAD_NUM) { printf("푸시 크래시\n"); CRASH(); }
            break;
        }
        else {
            LOG_(4);
        }
    }
}
```

2-2. 'ABA 이슈' 특정 근거

메모리 로그

메모리 1

주소: 0x0000023BB91A2A98

0x0000023BB91A2A98	b5 50 3c 6b f0 3b 02 00
0x0000023BB91A2AA0	b6 00 00 00 00 00 00 00
0x0000023BB91A2AA8	c5 50 3c 6b f0 3b 02 00
0x0000023BB91A2AB0	c6 00 00 00 00 00 00 00
0x0000023BB91A2AB8	c7 50 3c 6b f0 3b 02 00
0x0000023BB91A2AC0	c5 30 3b 6b f0 3b 02 00
0x0000023BB91A2AC8	c6 00 00 00 00 00 00 00
0x0000023BB91A2AD0	c7 30 3b 6b f0 3b 02 00
0x0000023BB91A2AD8	c5 50 40 6b f0 3b 02 00
0x0000023BB91A2AE0	c6 00 00 00 00 00 00 00
0x0000023BB91A2AE8	c7 50 40 6b f0 3b 02 00

메모리 1

주소: 0x0000023BBE275000

0x0000023BBE275000	c3 00 00 00 00 00 00 00
0x0000023BBE275008	a0 50 3c 6b f0 3b 02 00
0x0000023BBE275010	c0 20 88 6a f0 3b 02 00
0x0000023BBE275018	c1 c0 83 6a f0 3b 02 00
0x0000023BBE275020	c2 00 00 00 00 00 00 00
0x0000023BBE275028	c3 00 00 00 00 00 00 00
0x0000023BBE275030	c0 c0 82 6a f0 3b 02 00
0x0000023BBE275038	c1 20 88 6a f0 3b 02 00
0x0000023BBE275040	c2 00 00 00 00 00 00 00
0x0000023BBE275048	c3 00 00 00 00 00 00 00
0x0000023BBE275050	c0 80 84 6a f0 3b 02 00
0x0000023BBE275058	c1 c0 82 6a f0 3b 02 00

메모리 1

주소: 0x0000023BBE2753A0

0x0000023BBE2753A0	c2 00 00 00 00 00 00 00
0x0000023BBE2753A8	c3 00 00 00 00 00 00 00
0x0000023BBE2753B0	c0 e0 82 6a f0 3b 02 00
0x0000023BBE2753B8	c1 80 85 6a f0 3b 02 00
0x0000023BBE2753C0	c2 00 00 00 00 00 00 00
0x0000023BBE2753C8	c3 00 00 00 00 00 00 00
0x0000023BBE2753D0	a1 e0 82 6a f0 3b 02 00
0x0000023BBE2753D8	a2 00 00 00 00 00 00 00
0x0000023BBE2753E0	a3 00 00 00 00 00 00 00
0x0000023BBE2753E8	b7 50 3c 6b f0 3b 02 00
0x0000023BBE2753F0	b5 30 3b 6b f0 3b 02 00
0x0000023BBE2753F8	b6 00 00 00 00 00 00 00
0x0000023BBE275400	b7 30 3b 6b f0 3b 02 00

- ① b5 50 3c 6b f0 3b 02 00
b 스레드의 copy_top으로 top을 복사했습니다.
(b 스레드 copy_top == 0x6B3C50)
- ② c5 50 3c 6b f0 3b 02 00
c 스레드의 copy_top으로 top을 복사했습니다.
(c 스레드 copy_top == 0x6B3C50)
- ③ c7 50 3c 6b f0 3b 02 00 // * POP
c 스레드에서 스택에 변화가 없다고 판단하여, POP 진행
top 주소를 최신화 하고 copy_top을 delete 했습니다.
(delete copy_top == 0x6B3C50)
- 다른 노드들의 PUSH, POP 이 진행되었습니다.
- ④ a0 50 3c 6b f0 3b 02 00 // * PUSH
a 스레드에서 노드를 할당받고 PUSH 되었습니다.
이때 할당 노드는 '스레드 b'에서 copy_top으로 바라보는 '0x6B3C50'
- ⑤ b7 50 3c 6b f0 3b 02 00
b 스레드에서 스택에 변화가 없다고 판단하여, POP 진행
* 스레드에서 스택에 변화가 없다고 오인함!! ABA 이슈

※ 위의 두 문제의 해결방법은 이어지는 락프리 메모리풀에서 설명하겠습니다.

락프리 메모리 풀

네트워크 라이브러리에서 사용하기 위해 구현한 락프리 메모리 풀입니다. 락프리 메모리풀을 사용하여 락프리 스택 구현 중 발생한 메모리 디커밋 문제를 해결했으며, 락프리 스택 구조로 구현했습니다.

(github : https://github.com/dkldjswkd/cpp_Project/blob/main/00%20lib_jy/LFObjectPool.h)

LFObjectPool.h // Alloc(), Stack의 Pop에 해당

```
template<typename T>
T* LFObjectPool<T>::Alloc() {
    for (;;) {
        DWORD64 copyTopStamp = topStamp;
        Node* topClean = (Node*)(copyTopStamp & useBitMask);

        // Not empty!!
        if (topClean) {
            // Stamp 추출 및 newTopStamp 생성
            DWORD64 nextStamp = (copyTopStamp + stampCount) & stampMask;
            DWORD64 newTopStamp = nextStamp | (DWORD64)topClean->next;

            // 스택에 변화가 있었다면 다시시도
            if (copyTopStamp != InterlockedCompareExchange64((LONG64*)&topStamp, (LONG64)newTopStamp, (LONG64)copyTopStamp))
                continue;

            if (useCtor) {
                new (&topClean->object) T;
            }

            InterlockedIncrement((LONG*)&useCount);
            return &topClean->object;
        }

        // empty!!
        else {
            InterlockedIncrement((LONG*)&capacity);
            InterlockedIncrement((LONG*)&useCount);

            // Node 중 Object 포인터 ret
            return &(new Node((ULONG_PTR)this))->object;
        }
    }
}
```

LFObjectPool.h // Free(), Stack의 Push에 해당

```
template<typename T>
void LFObjectPool<T>::Free(T* p_object) {
    // 오브젝트 노드로 변환
    Node* pushNode = (Node*)((char*)p_object - objectOffset);

    if (integrity != pushNode->integrity)
        throw std::exception("ERROR_INTEGRITY");
    if (memGuard != pushNode->overGuard)
        throw std::exception("ERROR_INVALID_OVER");
    if (memGuard != pushNode->underGuard)
        throw std::exception("ERROR_INVALID_UNDER");

    if (useCtor) {
        p_object->~T();
    }

    for (;;) {
        DWORD64 copyTopStamp = topStamp;

        // top에 이어줌
        pushNode->next = (Node*)(copyTopStamp & useBitMask);

        // Stamp 추출 및 newTopStamp 생성
        DWORD64 nextStamp = (copyTopStamp + stampCount) & stampMask;
        DWORD64 newTopStamp = nextStamp | (DWORD64)pushNode;

        // 스택에 변화가 있었다면 다시시도
        if (copyTopStamp != InterlockedCompareExchange64((LONG64*)&topStamp, (LONG64)newTopStamp, (LONG64)copyTopStamp))
            continue;

        InterlockedDecrement((LONG*)&useCount);
        return;
    }
}
```

락프리 메모리 풀 설명

1. 락프리 구조 사용 및 락프리 스택 메모리 접근 예외 해결

락프리 구조로 구현되었기 때문에 락프리 자료구조 내부에서 노드할당을 위한 노드 풀로 사용할 수 있습니다. 해당 락프리 메모리풀을 사용함으로써, 락프리 스택 구현 중 발생한 디커밋 메모리 접근 예외를 해결했습니다. Alloc()은 락프리 스택의 POP()에 해당하는 동작을 하며, Free()는 락프리 스택의 PUSH()에 해당하는 동작을 합니다.

2. useCtor Flag 사용

락프리 메모리 풀 객체 생성 시 해당 플래그를 사용하여, 사용자가 객체 생성자, 소멸자 호출 여부를 결정할 수 있게했습니다. 해당 플래그가 true 라면 Alloc() 시 생성자를 호출하며, Free() 시 소멸자를 호출합니다. 해당 플래그가 false인 경우 생성자, 소멸자 호출이 생략되지만 Alloc(), Free() 시 성능이 향상됩니다.

3. integrity, under/over flow 체크

메모리 풀의 안정성을 높이기 위해 노드 반환(Free) 시 integrity, under/over flow 감지 기능을 구현했습니다. integrity 값은 메모리 풀 객체 주소값을 사용했으며, 반환 노드의 integrity 검사를 진행했을 때 false를 반환하게 된다면 이는 해당 메모리 풀에서 할당해준 노드가 아님을 뜻합니다.

락프리 메모리 풀 구현 중 나타난 문제

락프리 메모리풀은 락프리 스택 구조로 구현 되었기 때문에, 위에서 기재한 ‘락프리 스택 구현 중 발생한 문제’ 중 아직 ABA 문제가 남아있음을 인지했으며, 문제 해결을 위해 Stamp값을 도입했습니다. 락프리 스택에서 ABA 문제상황을 메모리 로그로 추적한 시나리오의 다음과 같았습니다.

1. top(A)을 copy_top에 복사하고 new_top = copy_top->next 의 작업을 진행
 2. 다른 스레드에서 A 노드 반환 및 재할당 (A 노드 POP 후 PUSH 됨)
 3. copy_top(A) == top(A) 이므로, top을 new_top으로 수정 (* ABA 이슈 발생)

메모리 로그의 시나리오를 추적해본 결과, ‘copy_top == top의 조건 만으로는 스택의 변화를 판단할 수 없다’의 결과에 도달할 수 있었습니다. 때문에 ‘스택의 변화를 top 노드의 주소 일치’만으로 판단하는 것이 아닌, 스택에 변화가 가해지는 PUSH, POP 작업 마다 증가하는 Stamp를 사용하여 Stamp값이 일치하는지 판단하는 방법으로 스택의 변화를 판단하는 방법을 사용했습니다.

락프리 큐

네트워크 라이브러리에서 사용하기 위해 구현한 락프리 큐입니다.

(github : https://github.com/dkldjswkd/cpp_Project/blob/main/00%20lib_jy/LFQueue.h)

LFQueue.h // LFQueue::Enqueue()

```
template<typename T>
void LFQueue<T>::Enqueue(T data) {
    Node* enqueueNode = nodePool.Alloc();
    enqueueNode->data = data;

    for (;;) {
        DWORD64 copyTailStamp = tailStamp;
        Node* tailClean = (Node*)(copyTailStamp & useBitMask);
        Node* tailNext = tailClean->next;

        if (tailNext == nullptr) {
            // Enqueue 시도
            if (InterlockedCompareExchange64((LONG64*)&tailClean->next, (LONG64)enqueueNode, NULL) == NULL) {
                InterlockedIncrement((LONG*)&nodeCount);

                // tail 이동
                DWORD64 newTailStamp = ((copyTailStamp + stampCount) & stampMask) | (DWORD64)enqueueNode;
                LONG64 ret = InterlockedCompareExchange64((LONG64*)&tailStamp, (LONG64)newTailStamp, (LONG64)copyTailStamp);
                return;
            }
        }
        else {
            // tail 이동
            DWORD64 newTailStamp = ((copyTailStamp + stampCount) & stampMask) | (DWORD64)tailNext;
            InterlockedCompareExchange64((LONG64*)&tailStamp, (LONG64)newTailStamp, (LONG64)copyTailStamp);
        }
    }
}
```

LFQueue.h // LFQueue::Dequeue()

```
template<typename T>
bool LFQueue<T>::Dequeue(T* data) {
    if (InterlockedDecrement((LONG*)&nodeCount) < 0) {
        InterlockedIncrement((LONG*)&nodeCount);
        return false;
    }

    for (;;) {
        DWORD64 copyHeadStamp = headStamp;
        DWORD64 copyTailStamp = tailStamp;

        //-----
        // head, tail 역전 방지
        //-----
        if (copyHeadStamp == copyTailStamp) {
            Node* tailClean = (Node*)(copyTailStamp & useBitMask);
            Node* tailNext = tailClean->next;

            // ABA or 큐에 Eq 반영 안됨 headNext
            if (tailNext == nullptr)
                continue;

            // tail 밀기
            DWORD64 newTailStamp = ((copyTailStamp + stampCount) & stampMask) | (DWORD64)tailNext;
            auto ret = InterlockedCompareExchange64((LONG64*)&tailStamp, (LONG64)newTailStamp, (LONG64)copyTailStamp);
        }

        //-----
        // Dequeue
        //-----
        Node* headClean = (Node*)(copyHeadStamp & useBitMask);
        Node* headNext = headClean->next;

        // ABA
        if (headNext == nullptr)
            continue;

        T dqData = headNext->data;

        // Dequeue 시도
        DWORD64 newHeadStamp = ((copyHeadStamp + stampCount) & stampMask) | (DWORD64)headNext;
        if (InterlockedCompareExchange64((LONG64*)&headStamp, (LONG64)newHeadStamp, (LONG64)copyHeadStamp) == (DWORD64)copyHeadStamp) {
            *data = dqData;
            nodePool.Free(headClean);
            return true;
        }
    }
}
```

락프리 큐 구현 중 나타난 문제

(구현단계의 코드이기 때문에 위의 완성 코드와 다릅니다.)

Enqueue 삽입이 성공했지만, Tail 이동 실패 (아래는 미완성 코드로, 구현 중 문제 발생 코드입니다.)

문제 발생 코드 // LFQueue::Enqueue()

```
template<typename T>
void LFQueue<T>::Enqueue(T data, BYTE thread_id) {
    char* wp;

    Node* insert_node = node_pool.Alloc();
    insert_node->data = data;
    insert_node->next = NULL;
    LOG(0, &insert_node);

    for (;;) {
        Node* copy_tail = tail;
        LOG(1, &copy_tail);
        Node* copy_next = copy_tail->next; // tail 전진을 위한 주소
        LOG(2, &copy_next);

        // Enqueue 성공
        if (InterlockedCompareExchange64((LONG64*)&copy_tail->next, (LONG64)insert_node, NULL) == NULL) {
            LOG(3, &insert_node);

            auto tmp = InterlockedIncrement((LONG*)&size);
            if (MAX_NODE < tmp) CRASH();

            // tail 뒤로 밀기
            LONG64 ret = InterlockedCompareExchange64((LONG64*)&tail, (LONG64)insert_node, (LONG64)copy_tail);
            if (ret != (LONG64)copy_tail) {
                // 밀기 실패 Tail 이동 실패!
                LOG(4);
                CRASH();
            }

            return;
        }
        // Enqueue 실패
        else {
            LOG(5);
        }
    }
}
```

문제 발생 코드 // LFQueue::Dequeue()

```
template<typename T>
bool LFQueue<T>::Dequeue(T* data, BYTE thread_id) {
    char* wp;
    (매개 변수) BYTE thread_id
    온라인 검색

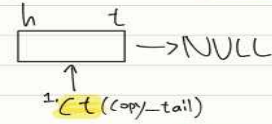
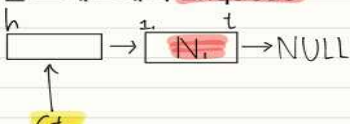
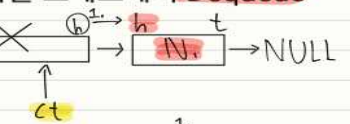
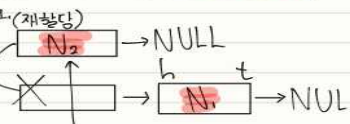
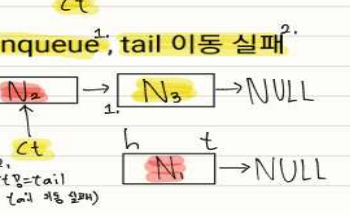
    for (;;) {
        Node* copy_head = head;
        LOG(6, &copy_head);

        Node* copy_next = copy_head->next;
        LOG(7, &copy_next);

        // Check Empty
        if (copy_next == nullptr) {
            LOG(8);
            return false;
        }

        // 디큐 성공
        if (InterlockedCompareExchange64((LONG64*)&head, (LONG64)copy_next, (LONG64)copy_head) == (LONG64)copy_head) {
            *data = copy_next->data;
            node_pool.Free(copy_head);
            auto tmp = InterlockedDecrement((LONG*)&size);

            // 로그
            LOG(9, &copy_next);
            if (tmp < 0) CRASH();
            return true;
        }
        // 디큐 실패
        else {
            LOG(10);
        }
    }
}
```

문제 상황 메모리 로그	문제 발생 상황 요약
메모리 1 주소: 0x0000020CB84ABFE0 0x0000020CB84ABFE0 1.a0 80 50 b3 b6 0c 02 00 0x0000020CB84ABFE8 2.a1 a0 b3 b4 b6 0c 02 00 0x0000020CB84ABFF0 3.b0 90 b2 b4 b6 0c 02 00 0x0000020CB84ABFF8 a2 00 00 00 00 00 00 00 0x0000020CB84AC000 4.b1 a0 b3 b4 b6 0c 02 00 0x0000020CB84AC008 5,6.a3 80 50 b3 b6 0c 02 00 0x0000020CB84AC010 7.a6 60 3b b4 b6 0c 02 00 0x0000020CB84AC018 a7 80 b9 b3 b6 0c 02 00 0x0000020CB84AC020 8.a9 80 b9 b3 b6 0c 02 00 0x0000020CB84AC028 9.a6 80 b9 b3 b6 0c 02 00 0x0000020CB84AC030 a7 a0 b3 b4 b6 0c 02 00 0x0000020CB84AC038 10.a9 a0 b3 b4 b6 0c 02 00 0x0000020CB84AC040 11.a6 a0 b3 b4 b6 0c 02 00 0x0000020CB84AC048 a7 80 50 b3 b6 0c 02 00 0x0000020CB84AC050 12.a9 80 50 b3 b6 0c 02 00 0x0000020CB84AC058 b2 80 50 b3 b6 0c 02 00 0x0000020CB84AC060 13.a0 a0 b3 b4 b6 0c 02 00 0x0000020CB84AC068 14.b3 90 b2 b4 b6 0c 02 00 0x0000020CB84AC070 a1 80 50 b3 b6 0c 02 00 0x0000020CB84AC078 15.b4 00 00 00 00 00 00 00 0x0000020CB84AC080 a2 00 00 00 00 00 00 00 0x0000020CB84AC088 a3 a0 b3 b4 b6 0c 02 00	문제 발생 상황 요약 1. Enqueue 하기위해 ^{1.} tail을 복사  2. 다른 스레드에서 ^{1.} Enqueue  3. 다른 스레드에서 ^{1.} Dequeue  4. 다른 스레드에서 ^{1.} 노드 생성 (Enqueue 준비단계)  5. ^{1.} Enqueue, ^{2.} tail 이동 실패 

문제 상황 재연

메모리 로그를 분석하여 Enqueue에 성공했지만, Tail 이동에 실패한 이유를 추적해보았습니다.

1. ⑪ a6 a0 b3 b4 b6 0c 02 00
a7 80 50 b3 b6 0c 02 00

B 스레드, Enqueue 예정 노드
0xB4B290 → NULL

Head

0xB4B3A0

Copy_tail(b)

Tail

0xB35080

Copy_next(a)

0xB4B3A0 → 0xB35080 → NULL

a 스레드의 Dequeue()
Dequeue CAS 연산을 위해, 'Head, Head->next'를 'copy_head, copy_next'에 복사했습니다.

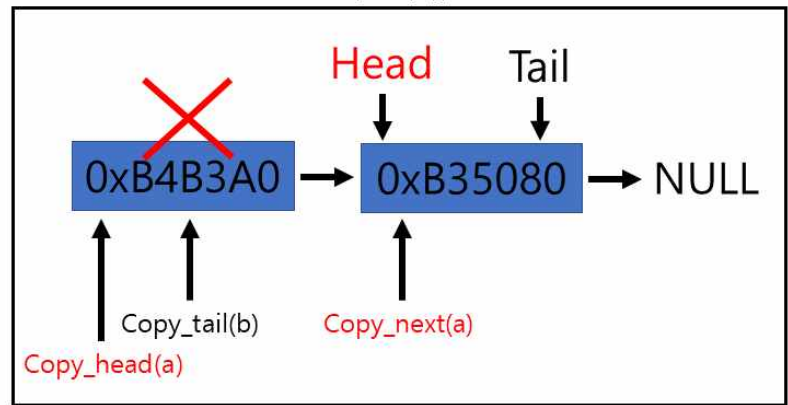
- 35 -

2. ⑫ a9 80 50 b3 b6 0c 02 00

B 스레드, Enqueue 예정 노드



락프리 큐



a 스레드의 Dequeue()

Dequeue CAS(head, copy_next, copy_head) 성공, Head를 copy_next로 변경에 성공했습니다.

3. ⑬ a0 a0 b3 b4 b6 0c 02 00

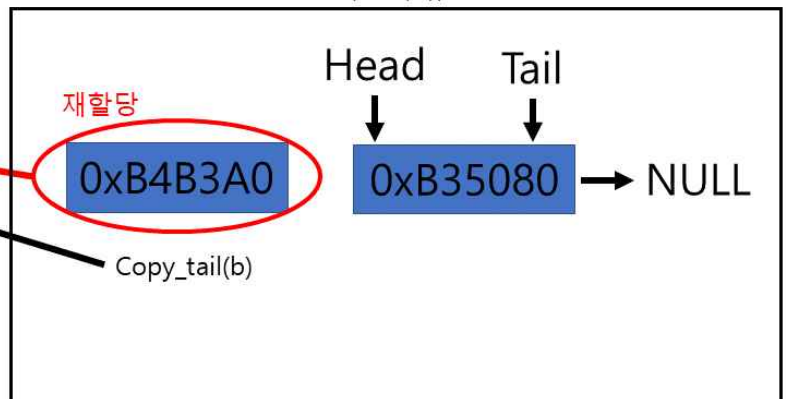
A 스레드, Enqueue 예정 노드



B 스레드, Enqueue 예정 노드



락프리 큐

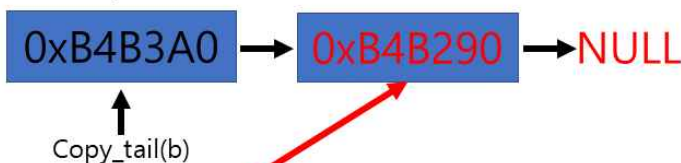


a 스레드의 Enqueue()

Enqueue 하기위해 Pool에서 노드를 할당했습니다. 이때 직전 Pool에 반환한 노드가 할당되었습니다. 이 노드를 b 스레드의 copy_tail이 바라보고 있습니다.

3. ⑭ b3 90 b2 b4 b6 0c 02 00

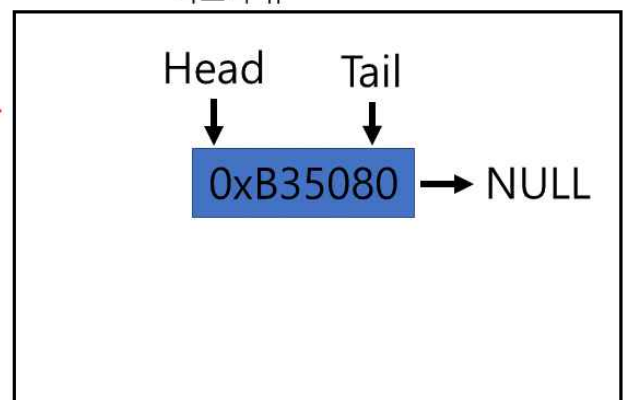
A 스레드, Enqueue 예정 노드



B 스레드, Enqueue 예정 노드



락프리 큐



b 스레드의 Enqueue()

b 스레드의 copy_tail이 a 스레드에서 Enqueue를 위해 할당받는 노드를 가르키고있으며, 해당 노드의 next가 nullptr 이므로 Enqueue CAS(copy_tail->next, 0xB4B290, nullptr) 성공, a 스레드의 Enqueue 예정 노드에 연결되었습니다.

Tail에 연결되지 않았으므로, Enqueue 후 Tail을 이동시키는 CAS 연산에 실패했습니다.

분석 결과

1. 스레드 B의 Enqueue에서 '0xB4B3A0' 노드를 copyTail로 바라보고 있었습니다.
 2. 스레드 A의 Dequeue로 스레드 B가 바라보는 '0xB4B3A0'가 풀에 반환되었습니다.
 3. 스레드 A의 Enqueue에서 노드를 할당했는데, '0xB4B3A0' 노드를 재할당 받았습니다.
 4. 스레드 A의 노드 할당으로 인해, 스레드 B에서 바라보던 '0xB4B3A0'노드의 next는 NULL이 되었습니다.
 5. 스레드 B는 노드 Enqueue 조건에 충족되었다고 판단하여 스레드 A의 노드에 Enqueue 했습니다.
- 위 과정을 살펴본 결과 예정노드에 Enqueue 하는 상황이 벌어져, Tail의 이동에 실패하는 것을 알 수 있었습니다.

문제 해결

위의 분석결과로 인해, 락프리 큐는 전용 Pool을 사용해야 한다는 결과를 얻을 수 있었습니다. tail 노드에 연결되지 않고, 다른 스레드에서 할당한 노드에 연결이 되었지만, 연결된 노드는 Enqueue를 위해 할당된 노드로, Enqueue가 예정된 노드입니다. 그렇기에 큐 자료구조에 즉시 반영되지는 못했지만, Enqueue 예정 노드에 연결이 되었으므로 노드의 반영을 기대할 수 있습니다. 하지만, 락프리 스택 또는 다른 락프리 큐와 같은 Pool을 사용한다면 다른 락프리 큐, 스택에서 사용중인 노드에 우리의 락프리 큐 노드가 달라 붙어 버리게 됩니다. 이로인해 락프리 큐에서 사용하는 노드는 전용 Pool에서 할당 받아야한다는 결과를 도출할 수 있었습니다. 이는 1. 락프리 큐 노드 유실문제, 2. 다른 메모리에 접근하여 write 해버리는 문제, 두 개의 치명적인 문제가 발생할 수 있습니다.

락프리 큐 구현 중 나타난 문제와 원인 특정 2

(구현단계의 코드이기 때문에 위의 완성코드와 다릅니다.)

Tail이 nullptr인 경우 (아래는 미완성 코드로, 구현 중 문제 발생 코드입니다.)

문제 발생 코드 // LFQueue::Dequeue()

```
template<typename T>
bool LFQueue<T>::Dequeue(T* data, BYTE thread_id) {
    char* wp;

    int dq_size = InterlockedDecrement((LONG*)&size);
    // 내가 빼서 마이너스가 되버린다면
    if (dq_size < 0) {
        // 증가 시키고 반환
        InterlockedIncrement((LONG*)&size);
        return false;
    }

    for (int i=0; i++) {
        // 내가 빼서 가야하는 노드 숫자
        wp = memLogger.Log(thread_id + 6, &dq_size);

        DWORD64 copy_head_ABA = head_ABA;
        Node* copy_head = (Node*)(copy_head_ABA & mask);
        wp = memLogger.Log(thread_id + 7, &copy_head);

        Node* copy_head_next = copy_head->next;
        wp = memLogger.Log(thread_id + 8, &copy_head_next);

        //-----
        // head, tail 역전 방지
        //-----

        DWORD64 copy_tail_ABA = tail_ABA;
        if (copy_head_ABA == copy_tail_ABA) {
            Node* copy_tail = (Node*)(copy_tail_ABA & mask);
            wp = memLogger.Log(thread_id + 9, &copy_tail);

            Node* copy_tail_next = copy_tail->next;
            wp = memLogger.Log(thread_id + 10, &copy_tail_next);

            DWORD64 tail_aba_count = (copy_tail_ABA + UNUSED_COUNT) & mask_stamp;
            DWORD64 next_tail_ABA = (tail_aba_count | (DWORD64)copy_tail_next);
            auto ret = InterlockedCompareExchange64((LONG64*)&tail_ABA, (LONG64)next_tail_ABA, (LONG64)copy_tail_ABA);

            // tail 밀기 성공
            if (ret == copy_tail_ABA) {Tail 이동 성공
                wp = memLogger.Log(thread_id + 11, &copy_tail_next);
                // 근데 tail이 nullptr로 밀림
                if (copy_tail_next == nullptr)
                    CRASH(); ✗ Tail이 nullptr !
            }

            if (i != 0 && i % 10000 == 0) printf("dq : %d, thread_id : %x\n", i, thread_id);
            continue;
        }

        //-----
        // Dequeue
        //-----

        // aba count 추출 및 new_ABA 생성
        DWORD64 haed_aba_count = (copy_head_ABA + UNUSED_COUNT) & mask_stamp;
        Node* next_head_ABA = (Node*)(haed_aba_count | (DWORD64)copy_head_next);

        // Dequeue 시도
        if (InterlockedCompareExchange64((LONG64*)&head_ABA, (LONG64)next_head_ABA, (LONG64)copy_head_ABA)
            == (DWORD64)copy_head_ABA)
        {
            wp = memLogger.Log(thread_id + 12, &copy_head_next);

            if (copy_head_next == nullptr)
                CRASH();

            *data = copy_head_next->data;
            node_pool.Free(copy_head);
            return true;
        }
    }
}
```

문제 상황 메모리 로그	문제 발생 상황 요약
메모리 1 주소: 0x00000165F001BA48 0x00000165F001BA48 1.b0 80 56 3a ee 65 01 00 0x00000165F001BA50 2.a3 70 dc 3a ee 65 01 00 0x00000165F001BA58 3.b1 50 4e 3a ee 65 01 00 0x00000165F001BA60 a3 01 00 00 00 6c b3 00 0x00000165F001BA68 4.a4 70 dc 3a ee 65 01 00 0x00000165F001BA70 5.a6 00 00 00 00 65 01 00 0x00000165F001BA78 a7 50 4e 3a ee 65 01 00 0x00000165F001BA80 a8 70 dc 3a ee 65 01 00 0x00000165F001BA88 6.ac 70 dc 3a ee 65 01 00 0x00000165F001BA90 7.a0 50 4e 3a ee 65 01 00 0x00000165F001BA98 a1 70 dc 3a ee 65 01 00 0x00000165F001BAA0 8.b2 00 00 00 00 00 00 00 0x00000165F001BAA8 9.b3 80 56 3a ee 65 01 00 0x00000165F001BAB0 b3 01 00 00 00 6c b3 00 0x00000165F001BAB8 b6 00 00 00 00 65 01 00 0x00000165F001BAC0 10.b7 70 dc 3a ee 65 01 00 0x00000165F001BAC8 b8 00 00 00 00 00 00 00 0x00000165F001BAD0 11.b9 70 dc 3a ee 65 01 00 0x00000165F001BAD8 ba 00 00 00 00 00 00 00 0x00000165F001BAE0 12.bb 00 00 00 00 00 00 00 0x00000165F001BAE8 a2 00 00 00 00 00 00 00 0x00000165F001BAF0 a3 50 4e 3a ee 65 01 00 0x00000165F001BAF8 a3 01 00 00 00 6c b3 00	<p>① tail 복사¹, Enqueue, Dequeue⁴</p> <p>② 노드 생성¹</p> <p>③ Enqueue¹, tail 이동 실패²</p> <p>④ Dequeue 실패 (tail 옮기 실패¹, 큐 노드 없음)</p>

문제 상황 재연 2

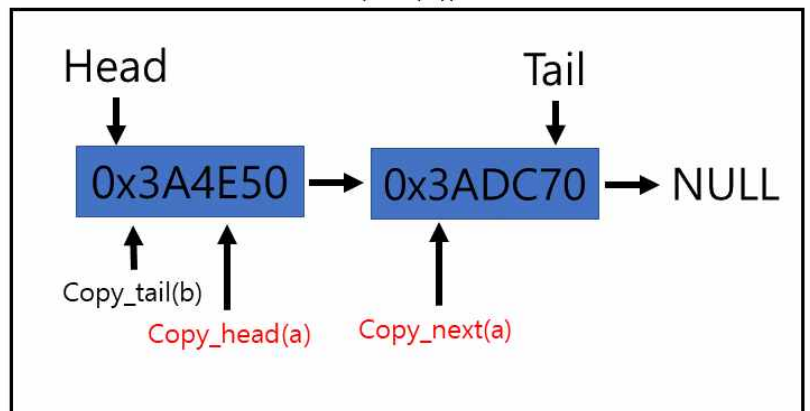
메모리 로그를 분석하여 'Tail == nullptr'인 이유를 추적해보았습니다.

- ⑤ a7 50 4e 3a ee 65 01 00
a8 70 dc 3a ee 65 01 00

B 스레드, Enqueue 예정 노드

0x3A5680 → NULL

락프리 큐



a 스레드의 Dequeue()

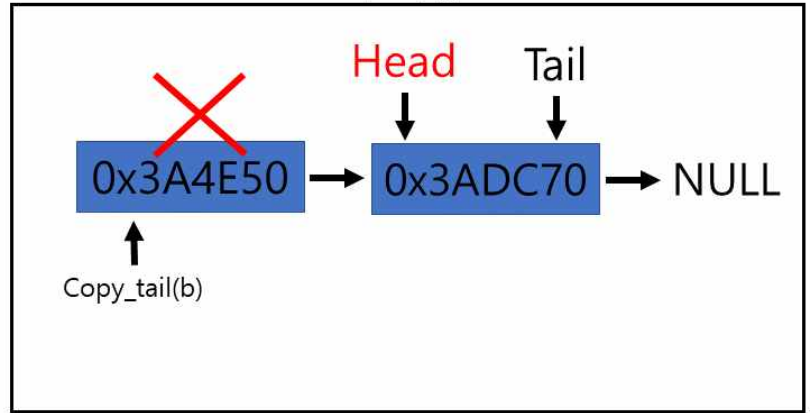
Dequeue CAS 연산을 위해, 'Head, Head->next'를 'copy_head, copy_next'에 복사했습니다.

2. ⑥ ac 70 3a ee 65 01 00

B 스레드, Enqueue 예정 노드

0x3A5680 → NULL

락프리 큐



a 스레드의 Dequeue()

Dequeue CAS(head, copy_next, copy_head) 성공, Head를 copy_next로 변경했습니다.

3. ⑦ a0 50 4e 3a ee 65 01 00

a1 70 dc 3a ee 65 01 00

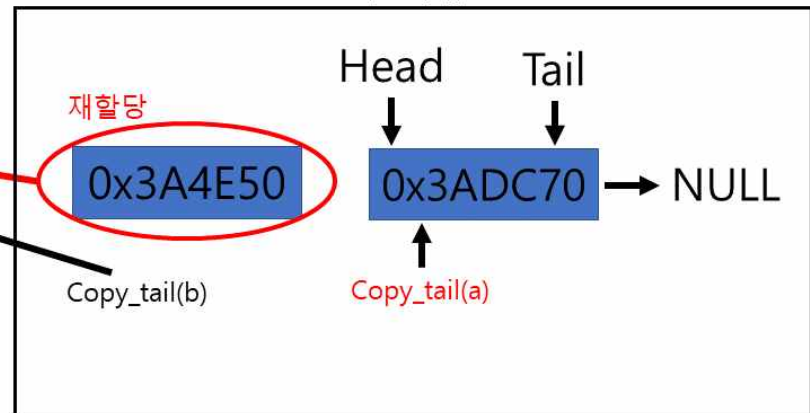
A 스레드, Enqueue 예정 노드

0x3A4E50 → NULL

B 스레드, Enqueue 예정 노드

0x3A5680 → NULL

락프리 큐



a 스레드의 Enqueue()

Enqueue 하기위해 Pool에서 노드를 할당했는데, 직전 Pool에 반환한 노드가 할당되었습니다. 이 노드를 b 스레드의 copy_tail이 바라보게 되었습니다.

4. ⑨ b3 80 56 3a ee 65 01 00

A 스레드, Enqueue 예정 노드

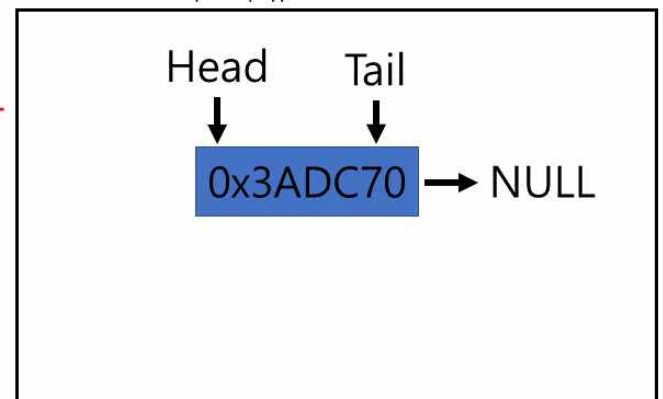
0x3A4E50 → 0x3A5680 → NULL

↑
Copy_tail(b)

0x3A5680

B 스레드, Enqueue 예정 노드

락프리 큐



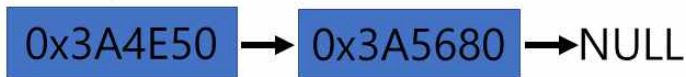
b 스레드의 Enqueue()

b 스레드의 copy_tail이 a 스레드에서 Enqueue를 위해 할당받는 노드를 가르키고 있습니다. 해당 노드의 next가 nullptr 이므로 Enqueue CAS(copy_tail->next, 0xB4B290, nullptr) 성공, a 스레드의 Enqueue 예정 노드에 연결되었습니다.

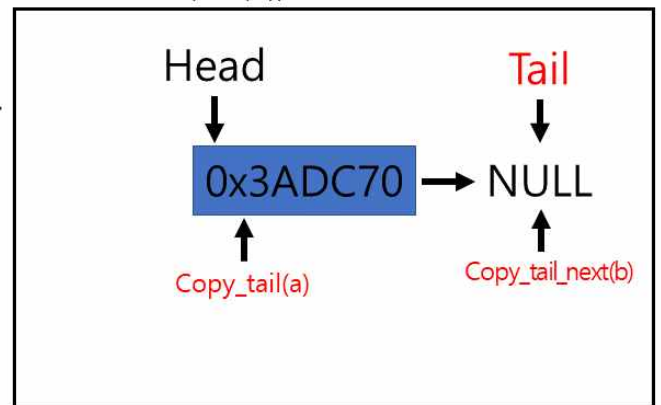
Tail에 연결되지 않았으므로, Enqueue 후 Tail을 이동시키는 CAS 연산에 실패하게됩니다.

5. ⑪ b9 70 dc 3a ee 65 01 00
 ba 00 00 00 00 00 00 00
 ⑫ bb 00 00 00 00 00 00 00

A 스레드, Enqueue 예정 노드



락프리 큐



b 스레드의 Dequeue()

‘Tail, Tail->next’를 ‘copy_tail, copy_tail_next’에 복사했습니다.

(‘Head == Tail’ 이므로 Dequeue 코드에서 Head를 이동하게되면 Head, Tail 역전상황이 발생합니다. 이를 방지하기 위해 Dequeue 코드에서는 ‘Head == Tail’시 Tail을 이동시키게 됩니다.

위는 ‘Head == Tail’의 상황이었으므로, Tail을 이동한 과정입니다.)

Tail 이동으로 인해 Tail이 nullptr이 되었습니다.

분석 결과 2

위의 과정을 살펴본 결과 ‘분석결과 1’에서 분석한 상황이 이번에도 동일하게 발생한 것을 확인할 수 있었습니다. 즉, Enqueue가 큐에 반영되지 않고, 다른 스레드의 노드에 반영됨으로써 Dequeue 작업 시 Tail을 이동시킬 다음 노드가 없기 때문에 Tail이 nullptr이 되버린 것을 알 수 있었습니다.

락프리 큐 문제 해결 2

Enqueue 시 락프리 큐 자료구조에 반영이 즉시 되지 않을 수 있으므로, 위 문제를 해결하기 위해 ‘Head == Tail’의 상황에서 Tail 이동 시 next가 nullptr인지 체크하고 nullptr이 아닐때만 Tail을 이동하는 방법으로 문제를 해결했습니다.

(tail의 next가 nullptr의 경우는 위 상황 뿐아니라 tail 노드 재할당의 경우도 존재합니다.)

new 보다 13배 빠른 메모리풀

기존 메모리 풀의 성능을 향상시키기 위해 객체 메모리 블록 단위가 아닌, 단위를 청크로하는 개선된 메모리풀을 구현했습니다.

(github : https://github.com/dkdldjswkd/cpp_Project/blob/main/00%20lib_jy/LFObjectPoolTLS.h)

LFObjectPoolTLS.h

```
private:
    LFObjectPool<Chunk> chunkPool;
    const int tlsIndex;
    const bool useCtor;
    int count = 0;

public:
    int GetChunkCapacity() {
        return chunkPool.GetCapacityCount();
    }

    int GetChunkUseCount() {
        return chunkPool.GetUseCount();
    }

    int GetUseCount() {
#ifdef USE_COUNT
        return count;
#else
        return GetChunkUseCount() * CHUNCK_SIZE;
#endif
    }

    T* Alloc() {
#ifdef USE_COUNT
        InterlockedIncrement((LONG*)&count);
#endif

        Chunk* p_chunk = (Chunk*)TlsGetValue(tlsIndex);
        if (nullptr == p_chunk) {
            p_chunk = chunkPool.Alloc();
            p_chunk->Set(&chunkPool, useCtor, tlsIndex);
            TlsSetValue(tlsIndex, p_chunk);
        }

        return p_chunk->Alloc();
    }

    void Free(T* p_object) {
#ifdef USE_COUNT
        InterlockedDecrement((LONG*)&count);
#endif

        ChunkData* p_chunkData = (ChunkData*)((char*)p_object - sizeof(Chunk*));
        p_chunkData->Free();
    }
};
```

사용자가 외부에서 사용하는 방법은 위에서 소개한 락프리 메모리풀과 동일합니다.

다만, 내부작동에서 차이가 있는데 기존 락프리 메모리풀은 Alloc() 시 객체가 담기는 메모리 블록을 POP 하여, 메모리 블록에서 객체 시작주소를 반환해주었고 Free() 시 메모리 블록을 다시 PUSH 했습니다.

즉, 할당 및 반환의 단위가 객체하나를 담는 메모리 블록이었는데, 지금 소개하는 개선된 락프리 메모리풀은 내부적으로 할당 및 반환의 단위를 청크단위(ex. 객체 500개)로 하는 청크 Pool을 사용합니다.

LFObjectPoolTLS::Chunk

```
static struct Chunk {
public:
    Chunk() {}
    ~Chunk() {}

private:
    LFObjectPool<Chunk>* p_chunkPool;
    bool useCtor;
    int tlsIndex;

private:
    ChunkData chunkDataArray[CHUNCK_SIZE];
    int allocIndex;
    alignas(64) int freeIndex;

public:
    void Set(LFObjectPool<Chunk>* p_chunkPool, bool useCtor, int tlsIndex) {
        this->p_chunkPool = p_chunkPool;
        this->useCtor = useCtor;
        this->tlsIndex = tlsIndex;

        allocIndex = 0;
        freeIndex = 0;
        for (int i = 0; i < CHUNCK_SIZE; i++) {
            chunkDataArray[i].p_myChunk = this;
        }
    }

    T* Alloc() {
        T* p_object = &(chunkDataArray[allocIndex++].object);
        if (useCtor) new (p_object) T;

        if (CHUNCK_SIZE == allocIndex) {
            Chunk* p_chunk = p_chunkPool->Alloc();
            p_chunk->Set(p_chunkPool, useCtor, tlsIndex);
            TlsSetValue(tlsIndex, (LPVOID)p_chunk);
        }
        return p_object;
    }

    void Free(T* p_object) {
        if (useCtor) p_object->~T();

        if (CHUNCK_SIZE == InterlockedIncrement((DWORD*)&freeIndex)) {
            p_chunkPool->Free(this);
        }
    }
};
```

위는 개선된 락프리 메모리풀에서 할당 단위로 사용하는 Chunk입니다.

개선버전에서는 Pool 내부에 존재하는 chunkPool에서 Chunk를 할당 받아, Chunk에서 객체를 사용자에게 반환하는 방식을 사용합니다.

성능 향상 기대 이유

락프리 자료구조는 Push(Enqueue), Pop(Dequeue) 경합 시 작업을 다시 수행해야 하므로 매우 큰 오버헤드이며, Pool은 락프리 스택으로 구현되었기 때문에 Alloc(), Free() 경합 시 위 문제를 갖고있습니다. 때문에 Alloc, Free 경합을 최소화 한다면 성능을 향상 시킬 수 있기 때문에 내부적으로 Chunk Pool을 두는 방식을 채택했습니다. 만약 객체 500개를 담는 Chunk를 사용한다면, 스레드는 자신의 Chunk에서 500개의 객체를 전부 할당해준 후 Chunk Pool에서의 Alloc을 하게됩니다. 이로 인해 Alloc의 경합확률은 '1/500'으로 감소하게 되어, 성능향상을 기대할 수 있습니다.

new/delete vs 개선 메모리 풀 성능 비교

성능 비교 코드, 결과	
<pre>#define OBJECT_SIZE 1024 #define LOOP_COUNT 3 #define ALLOC_COUNT 1000000 #define THREAD_COUNT 6 struct Object { char v[OBJECT_SIZE]; }; void HeapPerformanceTest() { Object ** objectArr = (Object**)malloc(sizeof(Object*) * ALLOC_COUNT); auto startTime = timeGetTime(); for(int i=0; i< LOOP_COUNT; i++){ for (int j = 0; j < ALLOC_COUNT; j++) { objectArr[j] = new Object; } for (int j = 0; j < ALLOC_COUNT; j++) { delete objectArr[j]; } } auto profileTime = timeGetTime() - startTime; free(objectArr); printf("profile Time : %d \n", profileTime); }</pre>	<div>New/Delete</div> <div>profile Time : 20541 profile Time : 20559 profile Time : 20601 profile Time : 20661 profile Time : 20673 profile Time : 20697</div> <div>TLSPool</div> <div>profile Time : 1582 profile Time : 1583 profile Time : 1591 profile Time : 1612 profile Time : 1645 profile Time : 1751</div>
(Pool 성능 측정 코드는 'new -> Pool::Alloc()', 'delete -> Free()' 부분만 다르며, 다른부분은 동일합니다.)	

성능 측정 결과, 개선된 메모리 풀이 new/delete보다 최대 13배 더 빠른 것으로 측정되었습니다. new/delete 보다 더 좋은 성능을 보일 수 있었던 이유는 다음과 같이 생각합니다.

1. Heap의 경우 일정 단위 이상의 메모리 반환 시 시스템에게 페이지 단위로 메모리를 반환하지만, Pool의 경우 메모리를 유저메모리 영역에서 관리하므로 메모리 할당 오버헤드가 감소합니다.
2. 멀티 스레드 환경에서 동일 Heap에 메모리 할당을 요청하므로 동기화로 인한 성능저하가 발생하지만, 개선된 Pool의 경우 다른 스레드의 간섭 없이 자신 스레드의 Chunck에서 메모리를 반환함으로, 동기화로 인한 성능저하가 없습니다.

5. 서버 개발을 위해 구현한 클래스

네트워크 라이브러리 및 사용자 구현 서버를 구현을 위해 구현하여 사용한 클래스들을 소개합니다.

DBConnector

MySQL C Connector를 사용하여 DB와 연결하는 클래스입니다.

(github : https://github.com/dkdldjswkd/cpp_Project/tree/main/01%20Network/DBConnector)

```
class DBConnector {
public:
    DBConnector(const char* dbAddr, int port, const char* loginID, const char* password,
                const char* schema, unsigned short loggingTime = INFINITE);
    ~DBConnector();

private:
    // mysql
    MYSQL conn;
    MYSQL* connection = nullptr;

    // opt
    int loggingTime;

private:
    void Log(const char* query, unsigned long queryTime);
    void ConnectDB(const char* dbAddr, int port, const char* loginID, const char* password,
                  const char* schema, unsigned short loggingTime = INFINITE);

public:
    MYSQL_RES* Query(const char* queryFormat, ...);
    MYSQL_RES* Query(const char* queryFormat, va_list args);
};
```

```
MYSQL_RES* DBConnector::Query(const char* queryFormat, ...) {
    char query[MAX_QUERY];

    va_list var_list;
    va_start(var_list, queryFormat);
    StringCchVPrintfA(query, MAX_QUERY, queryFormat, var_list);
    va_end(var_list);

    auto start = timeGetTime();
    if (0 != mysql_query(connection, (char*)query)) {
        throw DBException(query, mysql_errno(connection), mysql_error(connection));
    }
    auto durTime = timeGetTime() - start;
    if (loggingTime < durTime) {
        Log(query, durTime);
    }

    return mysql_store_result(connection);
}
```

쿼리 요청 시간을 측정하여 지정된 loggingTime을 초과하는 경우, 추후 분석을 위해 File Log를 남깁니다.

DBConnectorTLS

외부에서의 사용 방법은 DBConnector와 동일 합니다. DBConnectorTLS는 멀티 스레드 환경에서 각 스레드마다 독립적인 DB 연결을 사용하기 위해 구현했습니다.

(github : https://github.com/dkdljdjswkd/cpp_Project/tree/main/01%20Network/DBConnector)

```
class DBConnectorTLS {
public:
    DBConnectorTLS(const char* dbAddr, int port, const char* loginID, const char* password,
                   const char* schema, unsigned short loggingTime);
    ~DBConnectorTLS();

private:
    // tls
    const int tlsIndex;
    int useIndex = -1;
    DBConnector* connectorArr[MAX_CONNECTOR];

    // DBConnector 생성자 전달 인자
    char dbAddr[20];
    int port;
    char loginID[100];
    char password[100];
    char schema[100];
    unsigned short loggingTime = INFINITE;

private:
    DBConnector* Get();

public:
    MYSQL_RES* Query(const char* queryFormat, ...);
};
```

```
DBConnector* DBConnectorTLS::Get() {
    // TLS의 DBConnector를 가져옵니다
    DBConnector* p = (DBConnector*)TlsGetValue(tlsIndex);
    // TLS에 DBConnector 객체가 없는 경우입니다. (* 스레드에서 Get()이 최초 호출된 경우입니다.)
    if (nullptr == p) {
        // DBConnector를 생성하여, TLS에 셋팅합니다. (이 후 Get()호출 시 이때 생성한 DBConnector를 사용하게 됩니다.)
        p = new DBConnector(dbAddr, port, loginID, password, schema, loggingTime);
        TlsSetValue(tlsIndex, (LPVOID)p);
        connectorArr[InterlockedIncrement((LONG*)&useIndex)] = p;
    }

    return p;
}

MYSQL_RES* DBConnectorTLS::Query(const char* queryFormat, ...) {
    va_list args;
    va_start(args, queryFormat);
    MYSQL_RES* result = Get()->Query(queryFormat, args);
    va_end(args);
    return result;
}
```

DBConnectorTLS는 각 스레드가 자신만의 DBConnector 객체를 TLS에 보관합니다. DBConnectorTLS::Query() 함수를 호출하면 Get() 함수로 해당 스레드의 DBConnector 객체를 가져와서 DBConnector::Query() 함수를 실행합니다. DBConnectorTLS는 로그인 서버 구현 시 Worker Thread에서 AccountDB 조회를 위해 사용했습니다.

Profiler

코드 실행 시간을 측정 및 서버 퍼포먼스 개선을 위해 구현한 클래스입니다.

(github : https://github.com/dkdlldjswkd/cpp_Project/blob/main/00%20lib_jy/Profiler.h)

```
class Profiler {
private:
    Profiler();
public:
    ~Profiler();

public:
    static Profiler& getInst() {
        static Profiler instance;
        return instance;
    }

private:
    static struct ProfileData {
        // 중략
    };

private:
    const DWORD tlsIndex;
    int profileIndex = -1;
    ProfileData profileData[MAX_THREAD_NUM][PROFILE_DATA_NUM];

private:
    ProfileData* Get();

public:
    void ProfileBegin(const char* name);
    void ProfileEnd(const char* name);
    void ProfileFileOut();
    void ProfileReset();
};
```

```
Profiler::ProfileData* Profiler::Get() {
    ProfileData* profileArr = (ProfileData*)TlsGetValue(tlsIndex);
    if (profileArr == nullptr) {
        profileArr = profileData[InterlockedIncrement((LONG*)&profileIndex)];
        TlsSetValue(tlsIndex, (LPVOID)profileArr);
    }
    return profileArr;
}

void Profiler::ProfileEnd(const char* name) {
    ProfileData* profileArr = Get();

    for (int i = 0; i < PROFILE_DATA_NUM; i++) {
        // ProfileData 검색
        if (false == profileArr[i].useFlag) return;
        if (strcmp(profileArr[i].name, name) != 0)
            continue;

        // Profile Reset, 데이터 반영 x
        if (true == profileArr[i].resetFlag) {
            profileArr[i].Init();
            profileArr[i].resetFlag = false;
            return;
        }

        // profile time이 최대/최소 값이라면, 반영 x
        DWORD profileTime = timeGetTime() - profileArr[i].startTime;
        if (profileArr[i].VaildateData(profileTime)) return;

        // profile time 반영 o
        profileArr[i].totalTime += profileTime;
        profileArr[i].totalCall++;
    }
}
```

멀티 스레드 환경에서 사용 하기 위해 TLS를 사용했습니다. Profiler는 각 스레드가 자신이 기록하는 profileData를 TLS에 보관합니다. profile(Begin/End) 시 profileData를 Get() 함수로 가져와서 측정결과를 업데이트 합니다.

Parser

서버 시작 시 Parser 클래스로 System File을 읽어, 서버를 셋팅하는데 사용했습니다.

(github : https://github.com/dkldjswkd/cpp_Project/blob/main/00%20lib_jy/Parser.h)

```
struct Parser {
public:
    Parser();
    ~Parser();

private:
    char* fileBegin = nullptr;
    char* fileCur = nullptr;
    char* fileEnd = nullptr;

private:
    bool FindChar();
    bool FindSpace();
    bool FindWord(const char* word);
    bool NextChar();
    bool GetValueInt(int* value);
    bool GetValueStr(char* value);

public:
    int LoadFile(const char* file); // return file len
    bool GetValue(const char* section, const char* key, int* value);
    bool GetValue(const char* section, const char* key, char* value);
};
```

실사용 예

```
ServerConfig.ini
1 // NET_TYPE
2 // 0 : LAN
3 // 1 : NET
4
5 ChattingServer_Single // NET Server
6 {
7     // IP, PORT
8     IP = "0.0.0.0"
9     PORT = 12001
10
11     // 라이브러리 프로토콜
12     NET_TYPE = 1
13     PROTOCOL_CODE = 119 // 0x77
14     PRIVATE_KEY = 50 // 0x32
15
16     // 라이브러리 옵션
17     MAX_SESSION = 20000
18     NAGLE = 1
19     MAX_WORKER = 4
20     ACTIVE_WORKER = 3
21     // 타임아웃
22     TIME_OUT_FLAG = 0
23     TIME_OUT = 20000
24     TIME_OUT_CYCLE = 10000
25
26     // ChattingServer_Single
27     // ...
28 }
```

```
NetServer::NetServer(const char* systemFile, const char* server) {
    int serverPort;
    int nagle;

    // Read SystemFile
    parser.LoadFile(systemFile);
    parser.GetValue(server, "PROTOCOL_CODE", (int*)&protocolCode);
    parser.GetValue(server, "PRIVATE_KEY", (int*)&privateKey);
    parser.GetValue(server, "NET_TYPE", (int*)&netType);
    parser.GetValue(server, "PORT", (int*)&serverPort);
    parser.GetValue(server, "MAX_SESSION", (int*)&maxSession);
    parser.GetValue(server, "NAGLE", (int*)&nagle);
    parser.GetValue(server, "TIME_OUT_FLAG", (int*)&timeoutFlag);
    parser.GetValue(server, "TIME_OUT", (int*)&timeout);
    parser.GetValue(server, "TIME_OUT_CYCLE", (int*)&timeoutCycle);
    parser.GetValue(server, "MAX_WORKER", (int*)&maxWorker);
    parser.GetValue(server, "ACTIVE_WORKER", (int*)&activeWorker);
}
```