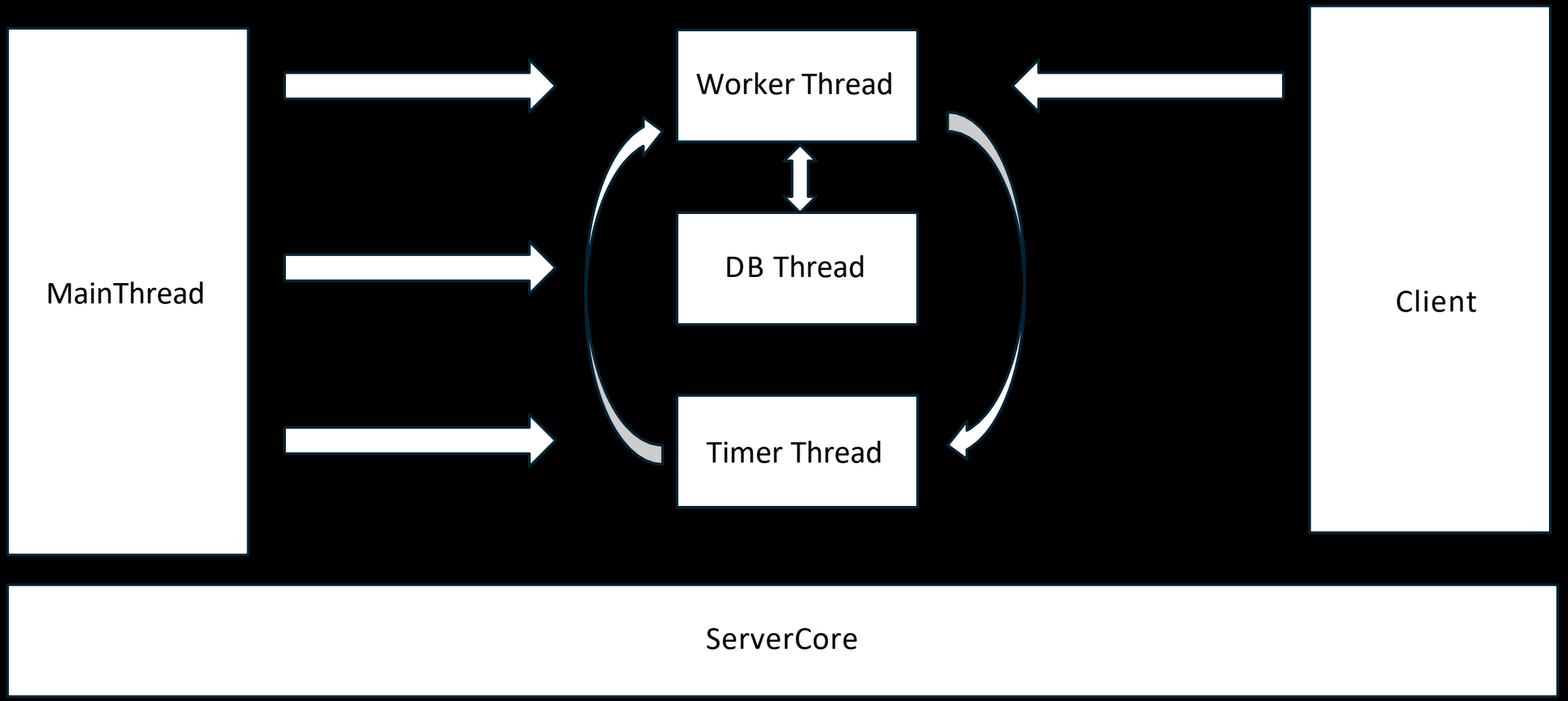
GameServerProgramming TermProject

Develop With IOCP Socket Model

1. 서버 설계도


MainThread
Worker Thread
DB Thread
Timer Thread
Client
ServerCore

ServerCore

게임서버 제작 전 효율적인 서버관리를 위한 라이브러리

DBconnection Pool : DB연결은 상당한 시간이 소요될 수 있음. 매번 연결을 시도하여 DB에 접근하는 것은 DB Thread에 심한 부하를 초래 할 수 있음.

Memory Pool : 메모리를 직접 할당 및 해제하는 것은 커널레벨의 개입이 존재하므로 오버헤드가 발생함. 또한, 메모리가 임의의 공간에 하당되기 때문에 추후 단편화 문제가 발생 할 수 있음.

R/W lock : 표준 뮤텍스의 경우 재귀적인 락 불가능 및 아주 가끔 상호베타적인 특성이 필요한 락의 경우 뮤텍스는 성능이 하락할 수 있음.

DeadLock ProFiler : 그래프의 개념을 이용하여 락의 사이클이 존재하는지 판단

Allocator : 메모리를 할당 및 해제할 때 사용할 구체적인 정책

Global Data

게임서버에서 사용되는 글로벌 객체들을 생성해주는 클래스이다.

```
extern class DBThread*  GDBThread;
extern class Sector* GSector;
extern class WorkerThread* GWorkerThread;
extern class TimerThread* GTimerThread;
extern class NPC* GNPC;
```

```
DBThread*       GDBThread = nullptr;
Sector*         GSector = nullptr;
WorkerThread*   GWorkerThread = nullptr;
TimerThread*    GTimerThread = nullptr;
NPC*            GNPC = nullptr;
class ServerGlobal
{
public:
    ServerGlobal()
    {
        GSector = new Sector();
        GDBThread = new DBThread();
        GWorkerThread = new WorkerThread();
        GTimerThread = new TimerThread();
        GNPC = new NPC();
    }
    ~ServerGlobal()
    {
        delete GSector;
        delete GDBThread;
        delete GWorkerThread;
        delete GTimerThread;
        delete GNPC;
    }
}GServerGlobal;
```

Main Thread

메인 스레드는 각종 초기화 작업을 담당하는 스레드다.

메인스레드에서 socket관련 초기작업을 해주고 NPC및PLAYER의 생성 DB Thread Timer Thread를 생성한다.

```
void NPC::InitNPC()
{
    for (int32 i = MAX_USER; i < MAX_USER + MAX_NPC; ++i) {

        GClients[i] = MakeShared<GameSession>();
        while (true) {

            GClients[i]->_x = rand() % W_WIDTH;
            GClients[i]->_y = rand() % W_HEIGHT;
            if (!isCollision(GClients[i]->_x, GClients[i]->_y)) {

                const auto& client = GClients[i];
                GSector->AddPlayerInSector(i, GSector->GetMySecto
                break;
```

```
template<typename Type, typename... Args>
Type* xnew(Args&&... args)
{
    Type* memory = static_cast<Type*>(PoolAllocator::Alloc(sizeof(Type)));
    new(memory)Type(forward<Args>(args)...); // placement new
    return memory;
}

template<typename Type>
void xdelete(Type* obj)
{
    obj->~Type();
    PoolAllocator::Release(obj);
}

template<typename Type>
shared_ptr<Type> MakeShared()
{
    return shared_ptr<Type>{ xnew<Type>(), xdelete<Type> };
}
```

NPC와 PLAYER모두 MakeShared를 통해 객체를 생성하는데, ServerCore에 있는 Memory Pool을 이용하여 할당한다.

DB Thread

DB 스레드는 Client의 로그인,정보저장, 정보추가에 대한 작업을 진행한다.
이때 ServerCore 제작한 DB Connection Pool을 이용하여 이미 ODBC에 연결되어 있는 객체를 꺼내온 뒤 작업.

```
DBConnection* connetedDB = GDBConnectionPool->Pop();

switch (ev.event) {
case DB_EVENT_TYPE::EV_LOGIN_PLAYER: {
    if (connetedDB->IsPlayerRegistered(ev.player_info._name)) {

        OVER_EXP* ov = xnew<OVER_EXP>();
        ov->_type = IO_TYPE::IO_GET_PLAYER_INFO;
        ov->_playerInfo = connetedDB->ExtractPlayerInfo(ev.player_info._name);
        ::PostQueuedCompletionStatus(gHandle, 1, ev.player_id, &ov->_over);
    }
    else {
        OVER_EXP* ov = xnew<OVER_EXP>();
        ov->_type = IO_TYPE::IO_ADD_PLAYER_INFO;
        ov->_playerInfo = ev.player_info;
        ::PostQueuedCompletionStatus(gHandle, 1, ev.player_id, &ov->_over);
    }
    GDBConnectionPool->Push(connetedDB);
    break;
}
```

이때 DB Thread는 멀티스레드 환경이므로 메서드 내부에 R/W Lock을 이용하여 접근한다.
작업이 끝난 후 PQCS 를 통해 Worker Thread로 결과를 통지한다.

Timer Thread

Timer 스레드는 NPC AI,체력회복,리스폰을 담당한다. Timer Thread를 통해 WorkerThread의 부하를 분산시킨다.

```
case TIMER_EVENT_TYPE::EV_HEAL: {
    OVER_EXP* ov = xnew<OVER_EXP>();
    ov->_type = IO_TYPE::IO_HEAL;
    ::PostQueuedCompletionStatus(gHandle, 1, ev.player_id, &ov->_over);
    break;
}
case TIMER_EVENT_TYPE::EV_PLAYER_RESPAWN: {
    OVER_EXP* ov = xnew<OVER_EXP>();
    ov->_type = IO_TYPE::IO_PLAYER_RESPAWN;
    ::PostQueuedCompletionStatus(gHandle, 1, ev.player_id, &ov->_over);
    break;
}
case TIMER_EVENT_TYPE::EV_AGGRO_MOVE: {
    OVER_EXP* ov = xnew<OVER_EXP>();
    ov->_type = IO_TYPE::IO_NPC_AGGRO_MOVE;
    ov->_aiTargetId = ev.aiTargetId;
    ::PostQueuedCompletionStatus(gHandle, 1, ev.player_id, &ov->_over);
    break;
}
```

PQCS를 통해 통지된 결과는 Worker Thread에서 처리한다.

Worker Thread

Worker Thread에선 GQCS함수를 통해 다양한 일감들을 처리한다.

```
void WorkerThread::DoWork()
{
    while (true) {
        DWORD numOfBytes;
        ULONG_PTR key;;
        WSAOVERLAPPED* over = nullptr;

        BOOL ret = ::GetQueuedCompletionStatus(gHandle, &numOfBytes, &key, &over, INFINITE);
        OVER_EXP* exOver = reinterpret_cast<OVER_EXP*>(over);

        if (FALSE == ret) {
            if (exOver->_type == IO_TYPE::IO_ACCEPT) std::cout << "Accept Error" << endl;
            else {
                std::cout << "GQCS Error on CLient[" << key << "]\n";
                Disconnect(static_cast<int>(key));
                if (exOver->_type == IO_TYPE::IO_SEND) xdelete(exOver);
                continue;
            }
        }

        if ((0 == numOfBytes) && ((exOver->_type == IO_TYPE::IO_RECV) || (exOver->_type == IO
            Disconnect(static_cast<int>(key));
            if (exOver->_type == IO_TYPE::IO_SEND) xdelete(exOver);
            continue;
        }

        switch (exOver->_type) {
        case IO_TYPE::IO_ACCEPT: { ... }
        case IO_TYPE::IO_RECV: { ... }
        case IO_TYPE::IO_SEND: { ... }
        case IO_TYPE::IO_GET_PLAYER_INFO: { ... }
        case IO_TYPE::IO_ADD_PLAYER_INFO: { ... }
        case IO_TYPE::IO_NPC_RANDOM_MOVE: { ... }
        case IO_TYPE::IO_NPC_RESPAWN: { ... }
        case IO_TYPE::IO_PLAYER_RESPAWN: { ... }
        case IO_TYPE::IO_NPC_AGGRO_MOVE: { ... }
        case IO_TYPE::IO_HEAL: { ... }
        }
    }
}
```

GQCS함수에서 반환된 Overlapped구조체의 정보는 한번에 한 스레드만 접근 할 수 있다. 이는 동기화 문제를 줄여준다.

```
case IO_TYPE::IO_ACCEPT: {
    uint32 clientId = GetNewClientId();
    if (clientId != -1) {

        GClients[clientId]->_state = SOCKET_STATE::ST_ALLOC;
        GClients[clientId]->_id = clientId;
        GClients[clientId]->_socket = GClientSocket;
        GClients[clientId]->_maxHp = PLAYER_MAX_HP;
        GClients[clientId]->_hp = PLAYER_MAX_HP;
        GClients[clientId]->_offensive = PLAYER_OFFENSIVE;
        GClients[clientId]->_die.store(false);

        ::CreateIoCompletionPort(reinterpret_cast<HANDLE>(GClientSocket), gHandle, clientId, 0);

        GClients[clientId]->RegisteredRecv();
        GClientSocket = SocketManager::CreateSocket();
    }
    else {
        std::cout << "MAX user exceeded\n";
    }
    ZeroMemory(&GOverExp._over, sizeof(GOverExp._over));
    DWORD bytesReceived = 0;
    SocketManager::AcceptEx(gListenSocket, GClientSocket, GOverExp._sendBuf, 0, sizeof(SOCKADDR_IN) + 16
    break;
}
```

최적화 전 state변경시 lock_guard를 이용했는데, 사실 여기엔 lock을 걸 필요가 없었다. 클라이언트가 서버에 연결하면 처음으로 만나는 코드이며, 다른스레드와 충돌이 일어나지 않는다.

Worker Thread

DB Table -> DB Thread -> PQCS -> GQCS 를 통해 실행되며,DB작업의 분리를 통한 병렬성을 향상시켰다.

```
ALTER PROCEDURE [dbo].[ExtractPlayerInfo]
    @Nickname NVARCHAR(20)
AS
BEGIN
    SET NOCOUNT ON;

    -- 플레이어의 x와 y 좌표를 가져오는 쿼리
    SELECT X, Y
    FROM player_data
    WHERE NICKNAME = @Nickname;
END
```

```
ALTER PROCEDURE [dbo].[IsPlayerRegistered]
    @Nickname nvarchar(20)
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @IsRegistered BIT;

    IF EXISTS (SELECT 1 FROM player_data WHERE NICKNAME = @Nickname)
    BEGIN
        SET @IsRegistered = 1;
    END
    ELSE
    BEGIN
        SET @IsRegistered = 0;
    END
    SELECT @IsRegistered AS IsRegistered;
END
```

```
case DB_EVENT_TYPE::EV_LOGIN_PLAYER: {
    if (connetedDB->IsPlayerRegistered(ev.player_info._name)) {

        OVER_EXP* ov = xnew<OVER_EXP>();
        ov->_type = IO_TYPE::IO_GET_PLAYER_INFO;
        ov->_playerInfo = connetedDB->ExtractPlayerInfo(ev.player_info._name);
        ::PostQueuedCompletionStatus(gHandle, 1, ev.player_id, &ov->_over);
    }
    else {
        OVER_EXP* ov = xnew<OVER_EXP>();
        ov->_type = IO_TYPE::IO_ADD_PLAYER_INFO;
        ov->_playerInfo = ev.player_info;
        ::PostQueuedCompletionStatus(gHandle, 1, ev.player_id, &ov->_over);
    }
    GDBConnectionPool->Push(connetedDB);
    break;
}
```

```
case IO_TYPE::IO_GET_PLAYER_INFO: {
    strcpy_s(GClients[key]->_name, exOver->_playerInfo._name.c_str());
    GClients[key]->_x = exOver->_playerInfo._x;
    GClients[key]->_y = exOver->_playerInfo._y;
    GClients[key]->_sectorX = GSector->GetMySector_X(GClients[key]->_x);
    GClients[key]->_sectorY = GSector->GetMySector_Y(GClients[key]->_y);
    GSector->AddPlayerInSector(key, GSector->GetMySector_X(GClients[key]->_x), G
    GClients[key]->_state = SOCKET_STATE::ST_INGAME;

    GClients[key]->SendLoginSuccessPacket();
    GClients[key]->Heal();

    const short sectorX = GClients[key]->_sectorX;
    const short sectorY = GClients[key]->_sectorY;

    for (int16 dy = -1; dy <= 1; ++dy) {
        for (int16 dx = -1; dx <= 1; ++dx) {
            int16 sectorY = GClients[key]->_sectorY + dy;
            int16 sectorX = GClients[key]->_sectorX + dx;
            if (sectorY < 0 || sectorY >= W_WIDTH / SECTOR_RANGE ||
                sectorX < 0 || sectorX >= W_HEIGHT / SECTOR_RANGE) {
                continue;
            }

            unordered_set<uint32> currentSector;
            {
                lock_guard<mutex> ll(GSector->sectorLocks[sectorY][sectorX]);
                currentSector = GSector->sectors[sectorY][sectorX];
            }

            for (const auto& id : currentSector) {
                auto& client = GClients[id];
                if (SOCKET_STATE::ST_INGAME != client->_state) continue;
                if (client->_id == key) continue;
                if (!CanSee(key, id)) continue;
                if (IsPc(client->_id)) client->SendAddPlayerPacket(key);
                else WakeUpNpc(client->_id, key);
                GClients[key]->SendAddPlayerPacket(client->_id);
            }
        }
    }

    xdelete(exOver);
    break;
```

IOCP Socket Model에서는 커스텀 Overlapped 구조체의 할당 및 해제가 빈번하게 발생하므로 ServerCore의 MemoryPool과 Allocator를 통해 최적화를 진행하였다.

Sector

인게임의 NPC+PLAYER를 모두 순회하는 것은 많은 부하를 불러일으키므로 섹터를 추가하여 인접한 객체들의 대해서만 동기화를 진행한다.

```
class Sector
{
public:
    std::array<std::array<std::unordered_set<uint32>, W_WIDTH / SECTOR_RANGE>, W_HEIGHT / SECTOR_RANGE> sectors;
    std::array<std::array<mutex, W_WIDTH / SECTOR_RANGE>, W_HEIGHT / SECTOR_RANGE> sectorLocks;

public:
    Sector() {};
    ~Sector() {};

    short   GetMySector_X(short x);
    short   GetMySector_Y(short y);

    void    AddPlayerInSector(uint32 player_id, short sector_x, short sector_y);
    void    RemovePlayerInSector(uint32 player_id, short sector_x, short sector_y);
    bool    UpdatePlayerInSector(uint32 player_id, short new_sector_x, short new_sector_y, short old_sector_x, short old_sector_y);
};
```

이 때의 섹터는 다수의 스레드에서 동시에 접근하여 쓰기 작업을 할 수 있으므로 각 섹터마다 mutex를 할당하여 DATA RACE를 예방한다.

NPC AI

NPC AI는 플레이어 섹터 순회 후 시야 안에 있는 NPC에 대해서 실행된다. 이때 NPC의 타입에 따라 다르게 로직을 적용시킨다.

```
void WorkerThread::WakeUpNpc(uint32 npcId, uint32 wakerId)
{
    if (GClients[npcId]->_die.load()) return;
    if (GClients[npcId]->_attack.load()) return;
    if (GClients[npcId]->_active.load()) return;

    bool expected = false;
    bool desired = true;

    if (!atomic_compare_exchange_strong(&GClients[npcId]->_active, &expected, desired)) return;

    switch (GClients[npcId]->_type) {
    case MONSTER_TYPE::PASSIVE: {

        TIMER_EVENT randomMoveEvent{ npcId,chrono::system_clock::now() + 1s,TIMER_EVENT_TYPE::EV_RANOM_MOVE,0 };
        GTimerJobQueue.push(randomMoveEvent);
        break;
    }
    case MONSTER_TYPE::AGGRO: {

        TIMER_EVENT aggroMoveEvent{ npcId,chrono::system_clock::now() + 1s,TIMER_EVENT_TYPE::EV_AGGRO_MOVE,wakerId };
        GTimerJobQueue.push(aggroMoveEvent);
        break;
    }
    }

}
```

PASSIVE NPC에 대해선 랜덤이동을, AGGRO NPC에 대해선 Astar알고리즘을 통해 플레이어를 추적한다.

개선내용

개선 전에는 WRITE_LOCK을 이중for문 외부에 걸어 lock의 범위가 매우 컸으며, Sector lock도 존재하지 않았고, oldSector를 참조로 받고 있었다. 병렬성이 낮고, 데이터가 보호되지 않으며, 이미 삭제된 값이나 잘못된 값에 접근할 가능성이 있었다.

```
{
	WRITE_LOCK;
	for (int16 dy = -1; dy <= 1; ++dy) {
		for (int16 dx = -1; dx <= 1; ++dx) {
			int16 sectorY = npc._sectorY + dy;
			int16 sectorX = npc._sectorX + dx;
			if (sectorY < 0 || sectorY >= W_WIDTH / SECTOR_RANGE ||
				sectorX < 0 || sectorX >= W_HEIGHT / SECTOR_RANGE) {
				continue;
			}
			const auto& oldSector = GSector->sectors[sectorY][sectorX];
			for (const auto& id : oldSector) {

				const auto& object = GClients[id];
				if (object->_state != ST_INGAME) continue;
				if (true == IsNPC(object->_id)) continue;
				if (!CanSee(object->_id, npcId))continue;
				oldList.insert(object->_id);
			}
```

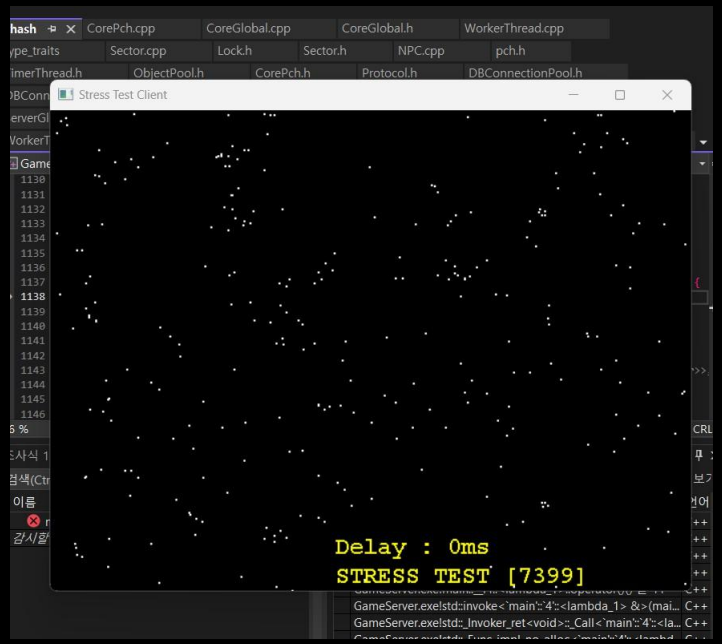
개선 전

```
for (int16 dy = -1; dy <= 1; ++dy) {
	for (int16 dx = -1; dx <= 1; ++dx) {
		int16 sectorY = npc._sectorY + dy;
		int16 sectorX = npc._sectorX + dx;
		if (sectorY < 0 || sectorY >= W_WIDTH / SECTOR_RANGE ||
			sectorX < 0 || sectorX >= W_HEIGHT / SECTOR_RANGE) {
			continue;
		}

		unordered_set<uint32> oldSector;

		{
			lock_guard<mutex> ll(GSector->sectorLocks[sectorY][sectorX]);
			oldSector = GSector->sectors[sectorY][sectorX];
		}

		for (const auto& id : oldSector) {

			const auto& object = GClients[id];
			if (object->_state != ST_INGAME) continue;
			if (true == IsNPC(object->_id)) continue;
			if (!CanSee(object->_id, npcId))continue;
			oldList.insert(object->_id);
		}
	}
```

개선 후

Astar 추가전 동접

개선 전에는 약 7천쯤에서 섹터 순회시 잘못된 메모리를 참조하여 프로그램이 죽었었지만, 개선 후에는 딜레이가 0ms로 지속되면서 1만6천쯤 소켓큐가 비어있다는 창이 뜨고 더 이상 새로운 플레이어를 받지 못하게 됨.

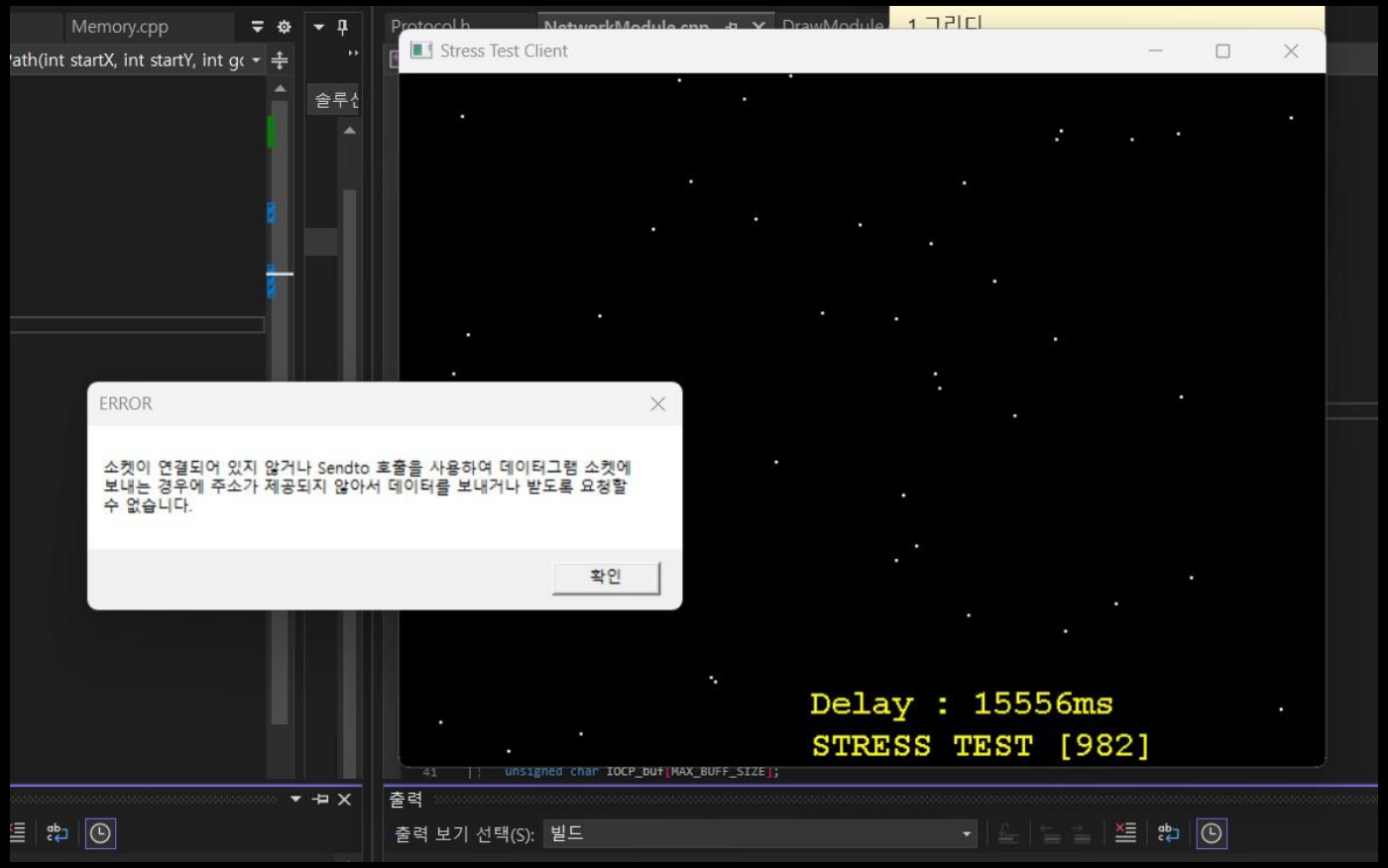


개선 전

개선 후

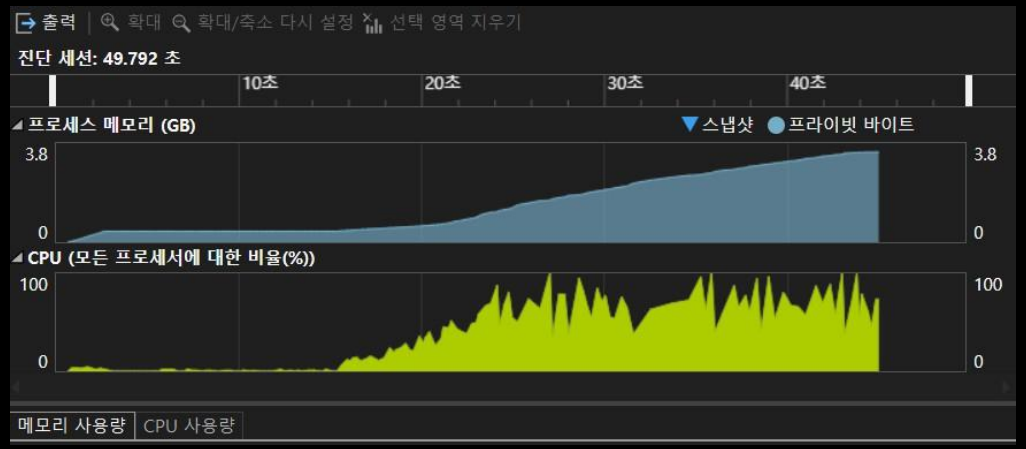
Astar 추가후 동접

기존의 코드에서 AGGRO NPC 5만마리에 대해서 Astar 알고리즘을 통해 경로를 탐색하는 로직만 추가했는데 동접이 매우 떨어짐.



원인 분석

FindPath메서드가 원인 경로를 생성하는 과정이 CPU에게 엄청난 부담.




```
23(0.01%)    int dx[] = { 0,0,-1,1 };
 4(0.00%)    int dy[] = { -1,1,0,0 };

237(0.07%)   for (int i = 0; i < 4; ++i) {
 94(0.03%)       int nextX = current._x + dx[i];
 30(0.01%)       int nextY = current._y + dy[i];
 27(0.01%)       if (nextX < 0 || nextX >= W_WIDTH || nextY < 0 || nextY >= W_HEIGHT) continue;
5092(1.60%)      if (isCollision(nextX, nextY)) continue;

54792(17.25%)    int newCost = gScores[{current._x, current._y}] + 1;
92897(29.24%)    if (!gScores.count({ nextX, nextY }) || newCost < gScores[{nextX, nextY}]) {
37937(11.94%)        gScores[{nextX, nextY}] = newCost;
   32(0.01%)         int priority = newCost + Heuristic(nextX, nextY, goalX, goalY);
  409(0.13%)         openSet.push({ nextX, nextY, priority, newCost, Heuristic(nextX, nextY, goalX, goalY) });
112580(35.44%)       cameFrom[{nextX, nextY}] = { current._x, current._y };
                 }
             }
```

실행 부하 과다 경로

함수 이름	총 CPU [단위, %]	셀프 CPU[단위, %]
GameServer (PID: 3776)	317697(100.00%)	0(0.00%)
[시스템 코드] ntdll.dll!0x00007ffbd892af38	317688(100.00%)	7(0.00%)
std::thread::_Invoke<std::tuple<`ThreadManager::Launch'::`2'::<lambda_1> >,0>	315713(99.38%)	0(0.00%)
std::_Func_impl_no_alloc<`main'::`4'::<lambda_1>,void>::_Do_call	314695(99.06%)	0(0.00%)
std::_Invoker_ret<void>::_Call<`main'::`4'::<lambda_1> &>	314695(99.06%)	0(0.00%)
std::invoke<`main'::`4'::<lambda_1> &>	314695(99.06%)	0(0.00%)
`main'::`4'::<lambda_1>::operator()	314695(99.06%)	0(0.00%)
WorkerThread::DoWork	314695(99.06%)	57(0.02%)
FindPath	307246(96.71%)	223133(70.23%)

최적화 시도

이전에는 매번마다 경로를 탐색했지만, 현재는 멤버로 경로변수를 추가하여, 경로를 저장한 뒤, 경로가 있다면, 그 변수에서 pop_back()메서드를 통해 탐색을 줄임.

```
AStar astar;
vector<NODE> AStarPath = astar.FindPath(GClients[key]->_x, GClients[key]->_y, GClients[playerId]->_x, GClients[playerId]->

if (!AStarPath.empty()) {
        NODE nextNode = AStarPath.back();
        short nextX = nextNode._x;
        short nextY = nextNode._y;

        GNPC->NPCAStarMove(key, nextX, nextY);

        if (CanAttack(key, playerId)) {
                moveNPC->_active.store(false);

                TIMER_EVENT ev{ key,chrono::system_clock::now() ,TIMER_EVENT_TYPE::EV_NPC_ATTACK_TO_PLAYER,playerId };
                GTimerJobQueue.push(ev);
        }
        else {
                if (CanSee(key, playerId)) {
                        TIMER_EVENT ev{ key,chrono::system_clock::now() + 1s, TIMER_EVENT_TYPE::EV_AGGRO_MOVE,playerId };
                        GTimerJobQueue.push(ev);
                }
        }

}
else {
        GClients[key]->_active.store(false);
}
xdelete(exOver);
```

개선 전

```
case IO_TYPE::IO_NPC_AGGRO_MOVE: {
    auto& moveNPC = GClients[key];
    short nextX = -1, nextY = -1;
    if (moveNPC->_astarPath.empty()) {
        moveNPC->_astarPath = FindPath(GClients[key]->_x, GClients[key]->_y, GClients[exOver->_aiTargetId]->_x, GClie
    }

    vector<NODE> path = moveNPC->_astarPath;

    if (!path.empty()) {
        nextX = path.back()._x;
        nextY = path.back()._y;
        moveNPC->_astarPath.pop_back();
    }
    else {
        moveNPC->_active.store(false);
        xdelete(exOver);
        break;
    }

    GNPC->NPCAStarMove(key, nextX, nextY);

    if (CanAttack(key, exOver->_aiTargetId)) {
        moveNPC->_active.store(false);

        TIMER_EVENT ev{ key,chrono::system_clock::now() ,TIMER_EVENT_TYPE::EV_NPC_ATTACK_TO_PLAYER,exOver->_aiTarget
        GTimerJobQueue.push(ev);
    }
    else {
        if (CanSee(key, exOver->_aiTargetId)) {
            TIMER_EVENT ev{ key,chrono::system_clock::now() + 1s, TIMER_EVENT_TYPE::EV_AGGRO_MOVE,exOver->_aiTargetI
            GTimerJobQueue.push(ev);
        }
        else {
            moveNPC->_active.store(false);
        }
    }
```

개선 후

최적화 시도

이전에는 충돌체크를 매번마다 반복문을 순회하여 했지만, 현재는 서버초기단계에서 장애물 정보를 글로벌 컨테이너에 저장하여 바로 충돌정보를 확인할 수 있게 함.

```
bool isCollision(short x_pos, short y_pos)
{
	for (int i = 0; i < SCREEN_WDITH; ++i) {
		for (int j = 0; j < SCREEN_HEIGHT; ++j) {
			int tile_x = i + x_pos;
			int tile_y = j + y_pos;
			if ((tile_x < 0) || (tile_y < 0)) continue;
			if (0 == (tile_x / 3 + tile_y / 3) % 3) {
				return false;
			}
			else if (1 == (tile_x / 3 + tile_y / 3) % 3)
			{
				return false;
			}
			else {  //Àå¾Ö¹° Ãæµ¹ Ã³¸® ÇØ¼ßÇÔ
				if (0 == (tile_x / 2 + tile_y / 2) % 3) {
					return true;
				}
				else if (1 == (tile_x / 2 + tile_y / 2) % 3) {
					return false;
				}
				else {
					return false;
				}
			}
		}
	}
}
```

개선 전

```
array<array<bool, W_HEIGHT>, W_WIDTH> GCollision;

constexpr int SCREEN_WDITH = 16;
constexpr int SCREEN_HEIGHT = 16;

bool checkCollision(short x_pos, short y_pos)
{
    if (x_pos < 0 || y_pos < 0 || x_pos >= W_HEIGHT || y_pos >= W_WIDTH) return false;
    if ((x_pos / 3 + y_pos / 3) % 3 == 0) return false;
    else if ((x_pos / 3 + y_pos / 3) % 3 == 1) return false;
    else {
        if ((x_pos / 2 + y_pos / 2) % 3 == 0) return true;
        else if ((x_pos / 2 + y_pos / 2) % 3 == 1) return false;
        else return false;
    }
}

bool isCollision(short x_pos, short y_pos)
{
    return GCollision[y_pos][x_pos];
}

void InitCollisionTile()
{
    for (int i = 0; i < W_WIDTH; ++i) {
        for (int j = 0; j < W_HEIGHT; ++j) {
            if (checkCollision(i, j) == true)
                GCollision[i][j] = true;
            else
                GCollision[i][j] = false;
        }
    }

    cout << "Init Collision Tile" << endl;
}
```

개선 후

최적화 후 동접

최적화를 진행했음에도 불구하고, 전혀 개선되지 않은 결과가 나옴.

생각의 전환을 시도 -> 다른스레드의 작업으로 인해 엄청나게 먼 거리를 탐색 할 가능성이 있지 않을까?

```
case IO_TYPE::IO_PLAYER_RESPAWN: {

    auto& respawnPlayer = GClients[key];
    while (true) {
        respawnPlayer->_x = rand() % W_WIDTH;
        respawnPlayer->_y = rand() % W_HEIGHT;
        if (!isCollision(respawnPlayer->_x, respawnPlayer->_y)) {
            GSector->AddPlayerInSector(respawnPlayer->_id, GSector->GetMySector_X(respawnPlayer->_x), GSector->GetMySector_Y(respawnPlayer->_y));
            respawnPlayer->_sectorX = GSector->GetMySector_X(respawnPlayer->_x);
            respawnPlayer->_sectorY = GSector->GetMySector_X(respawnPlayer->_y);
            break;
        }
    }
    respawnPlayer->_die.store(false);
    respawnPlayer->_hp = PLAYER_MAX_HP;
    GClients[key]->_state = SOCKET_STATE::ST_INGAME;

    for (int16 dy = -1; dy <= 1; ++dy) {
        for (int16 dx = -1; dx <= 1; ++dx) {
            int16 sectorY = respawnPlayer->_sectorY + dy;
            int16 sectorX = respawnPlayer->_sectorX + dx;
            if (sectorY < 0 || sectorY >= W_WIDTH / SECTOR_RANGE ||
                sectorX < 0 || sectorX >= W_HEIGHT / SECTOR_RANGE) {
                continue;
            }

            unordered_set<uint32> currentSector;

            {
                lock_guard<mutex> ll(GSector->sectorLocks[sectorY][sectorX]);
                currentSector = GSector->sectors[sectorY][sectorX];
            }

            for (const auto& id : currentSector) {
                if (GClients[id]->_state != SOCKET_STATE::ST_INGAME) continue;
                if (!CanSee(respawnPlayer->_id, id)) continue;
                if (IsPc(id)) GClients[id]->SendRespawnPlayerPacket(respawnPlayer->_id);
                else GWorkerThread->WakeUpNpc(id, respawnPlayer->_id);
                respawnPlayer->SendAddPlayerPacket(id);
            }
        }
    }

    respawnPlayer->Heal();

    xdelete(exOver);
```

여기는 플레이어가 사망한 후 부활할 때 실행되는 코드인데 만약 여기서 ST_INGAME으로의 변경이 좌표설정보다 먼저 된다면?
➔ 플레이어는 사망 시 -1,-1 위치로 초기화 해놓음
그렇다면 찰나의 순간 NPC의 좌표와 -1,-1의 경로가 탐색될 수 있는 여지가 있음.

그렇다면 탐색길어지고 메모리도 많이 잡아먹게 되고, 오랜탐색으로 병목현상이 발생할 수 도 있게 됨.

최적화 시도

Atomic_thread_fence를 추가하여 Session의 기본정보의 초기화를 확실하게 보장한 뒤 ST_IGNAME을 하여 다른스레드에서 정확한 값을 읽도록 수정.

```
auto& respawnPlayer = GClients[key];
while (true) {
    respawnPlayer->_x = rand() % W_WIDTH;
    respawnPlayer->_y = rand() % W_HEIGHT;
    if (!isCollision(respawnPlayer->_x, respawnPlayer->_y)) {
        GSector->AddPlayerInSector(respawnPlayer->_id, GSector->GetMySector_X(respawnPlayer->_x), GSector->GetMySector_Y(respawnPlayer->_y));
        respawnPlayer->_sectorX = GSector->GetMySector_X(respawnPlayer->_x);
        respawnPlayer->_sectorY = GSector->GetMySector_X(respawnPlayer->_y);
        break;
    }
}
respawnPlayer->_die.store(false);
respawnPlayer->_hp = PLAYER_MAX_HP;

atomic_thread_fence(memory_order_release);

GClients[key]->_state = SOCKET_STATE::ST_INGAME;
```

이렇게 했음에도 여전히 많은 부하가 발생. 이것이 원인이 아님.
CPU성능의 문제일 수도 있을 것 같음.

2024-11-23 최적화 시도

Astar 코드를 전반적으로 수정

1.Map자료구조 -> unordered_map으로 변경(이에 따른 해시함수 정의)
➔ 기존 map의 경우 정렬을 유지하면 삽입이 일어나지만 이 코드에서는 정렬이 불필요
(정렬 오버헤드 제거)
2.탐색의 시작점과 목표점이 같다면 빈 벡터를 바로 반환
-> 같은 위치에 있다면 바로 반환하여 최적화
3.경로 완성하고 벡터에 경로를 저장할 때 복사가 아닌 참조로 변경
->불필요한 메모리 복사 방지

GameSession클래스에 멤버변수 추가
1. 플레이어가 추적당하는 NPC의 아이디를 저장하는 멤버변수 추가
->하나의 몬스터만 NPC를 추적하도록 하기 위함

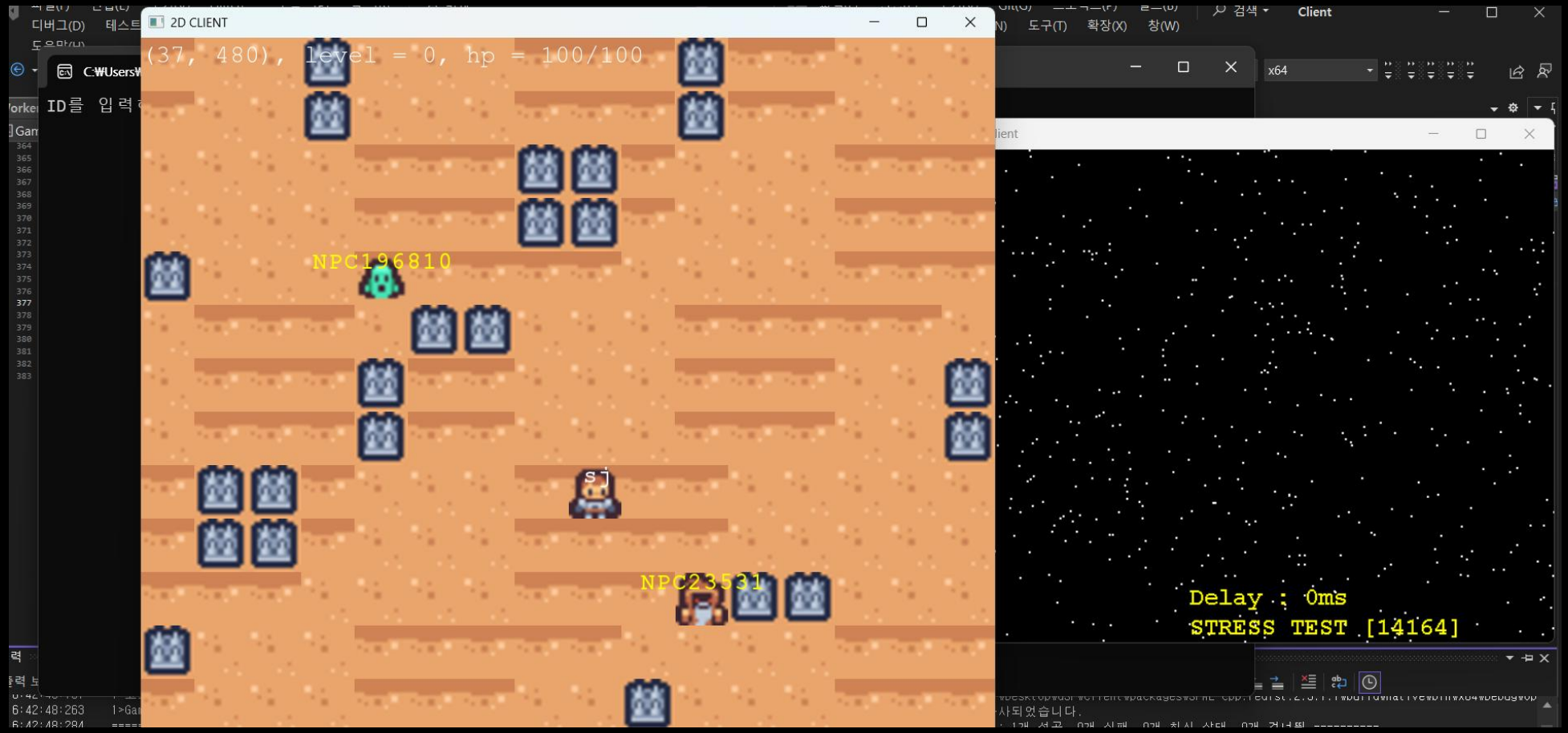
2024-11-23 최적화 시도

WorkerThread 코드 수정

1. 플레이어 입장,추가,부활 하는 상황에서 atomic_thread_fence를 제거하고 확실히 초기화 된 sessio만 접근 가능하도록 lock_guard를 추가.

2.GQCS를 통해 통지된 정보로 접근되는 IO_NPC_RANDOM_MOVE부분 불필요한 코드 제거
->잘못된 조건문 사용중이었음
3.플레이어가 사망 or NPC가 사망한 상황일 때 새로 추가한 멤버변수의 값을 -1로 초기화
->사망한 상태에서 잘못된 플레이어를 추적하는 것을 방지

2024-11-23 최적화 시도 후 동접



이전보다 확실히 개선이 이루어짐