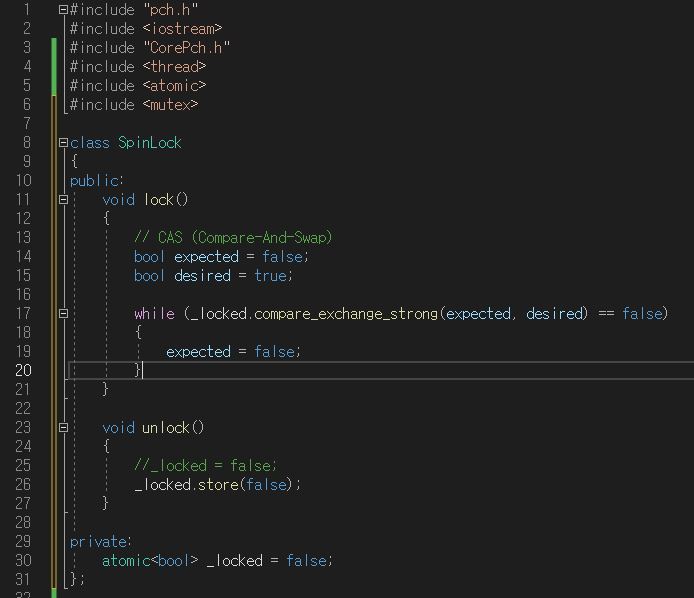
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **주차** | 4주차 | **기간** | 2023.7.17~ 2023.7.23 | **지도교수** | (서명) |
| 이번주 한일 요약 | - 게임 서버 공부  (멀티스레드), (메모리 관리) | | | | |

<상세 수행내용>

[멀티스레드]

**<SpinLock>**

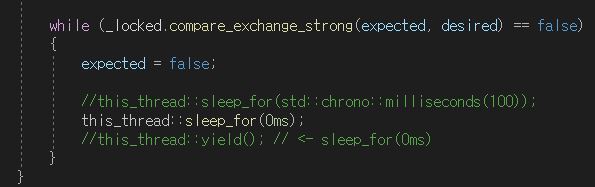
****

무작정 대기하기.

장점: 빠른 응답성을 제공.

단점: CPU를 계속 사용하여 자원을 낭비할 수 있고, 효율성 문제가 발생할 수 있다.

**<Sleep>**

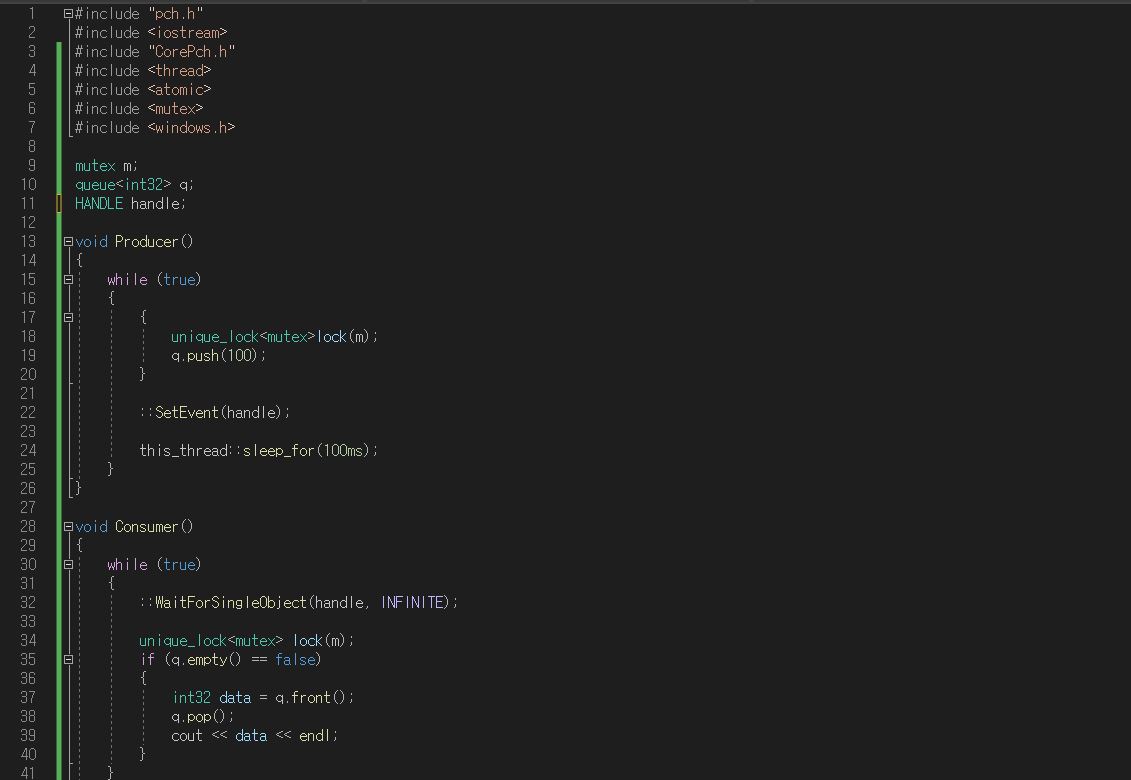
****

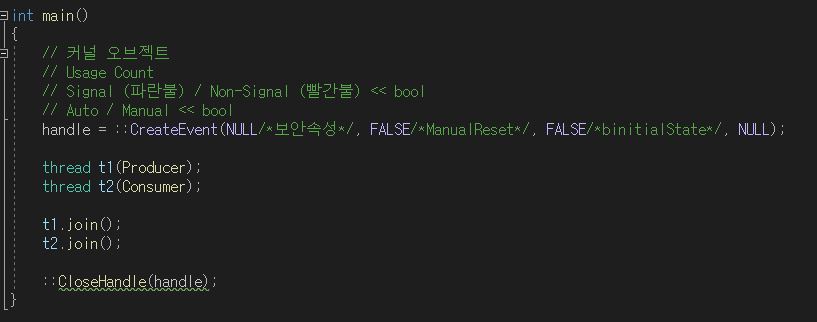
스레드를 일시적으로 정지시켜 다른 스레드에게 실행 기회를 양보하는데 사용

장점: 대기하는 동안 CPU사용을 해제하여 시스템 전체적인 자원 활용도를 높일 수 있다.

단점: 응답성이 떨어짐.

**<Event>**

****

****

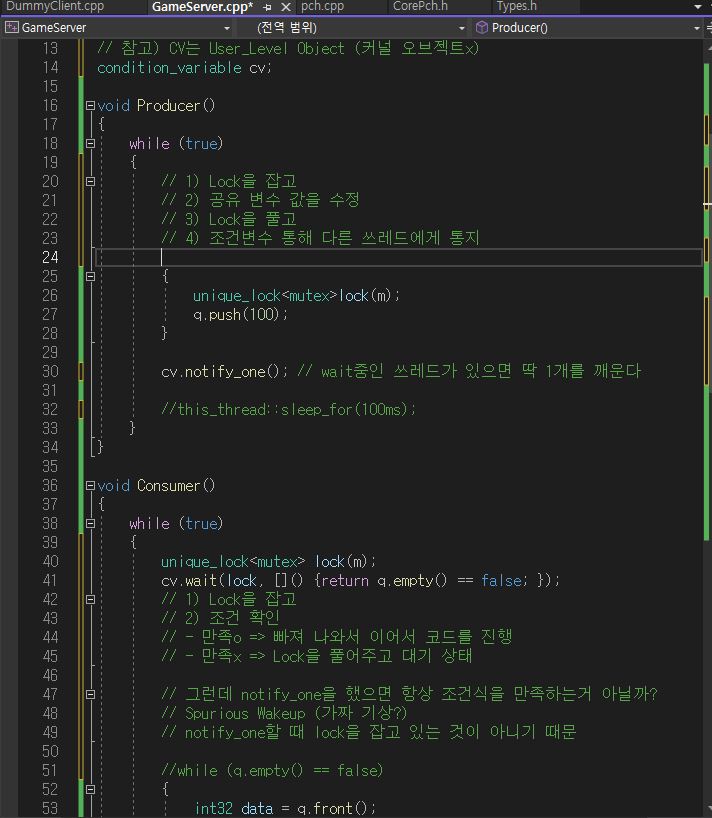
특정 조건이 충족될 때까지 기다리는 역할

장점: 블록킹하여 CPU를 해제하므로 시스템 전체적인 자원 활용도를 높일 수 있고, 유연성을 제공한다.

단점: 추가적인 시그널링 메커니즘이 필요하며, 교착 상태가 발생할 수 있다.

블록킹(Blocking): 스레드가 어떤 작업을 수행 중일 때, 해당 작업이 완료될 때까지 스레드의 실행이 멈추는 상태. 특정 작업이 완료되기를 기다리는 동안 다른 작업을 수행할 수 없는 상태.

**<Condition Variable>**

****

특정 조건을 만족할 때까지 기다리는 기법.

장점: Sleep과 비교하여 효율적인 대기가 가능하며, 조건의 변화에 따라 스레드를 깨우는 기능을 제공.

단점: 사용 시 적절한 동기화 메커니즘이 필요하며, 복잡성이 높을 수 있다.

Event는 스레드가 특정 이벤트의 발생을 기다리고 블록킹하는데 주로 사용되며,

Condition Variable은 특정 조건의 충족 여부를 기다리고 블록킹하지 않고 스레드를 깨우는 데 주로 사용

**<Future>**

비동기적으로 실행되는 작업의 결과를 나타내는데 사용되는 동기화 기법

장점: 비동기 작업의 결과를 효율적으로 처리할 수 있고, 작업 완료 여부를 확인하는 기능을 제공.

단점: 작업이 완료될 때까지 블록킹되며, 작업이 복잡해질 경우 관리가 어려울 수 있다.

복습

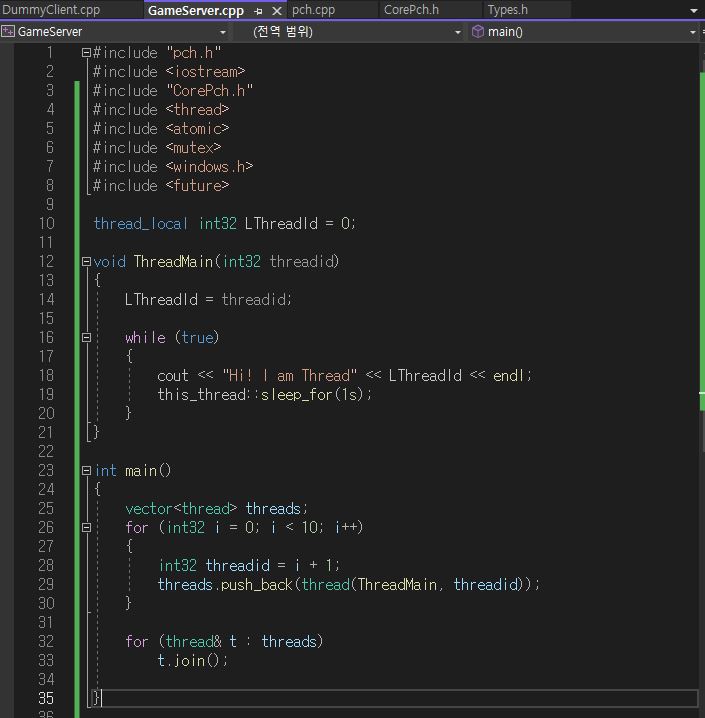
여러 스레드가 동일한 메모리에 동시 접근. 그 중 하나는 write 연산.

결과는? Race Condition(경합 조건)이 일어난다.

Undefined Behavior (정의되지 않은 행동)

* Lock (mutex)를 이용한 상호 배타적(Mutual Exclusive) 접근
* Atomic (원자적) 연산을 이용

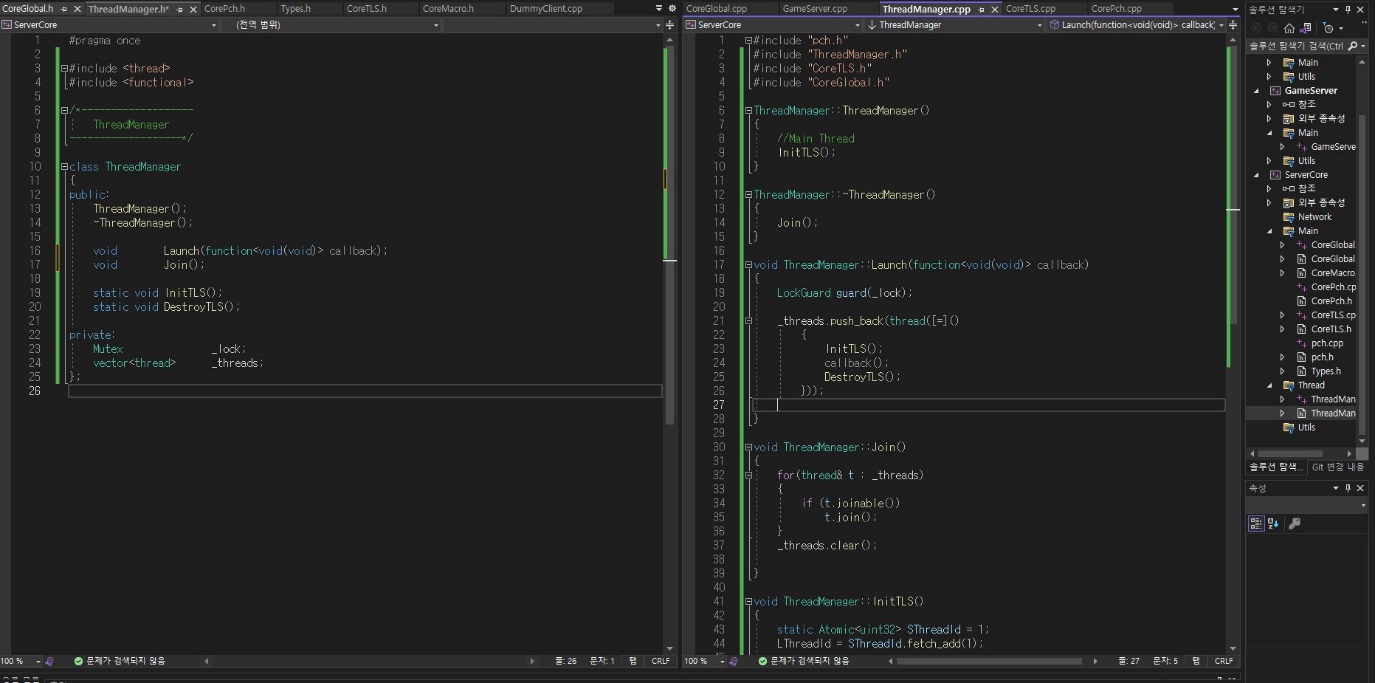
**<Thread Local Storage (TLS)>**

****

멀티스레드 환경에서 각각의 스레드마다 독립적인 저장소를 제공하는 메모리 공간

스레드 사이의 데이터 간섭 문제를 해결할 수 있음.

**<ThreadManager>**

****

텍스트, 디스플레이, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

멀티스레드 환경에서 스레드를 관리하고 조정하는 역할을 수행하는 클래스 또는 구성 요소

[메모리 관리]

**<Reference Counting>**

객체에 대한 참조 횟수를 관리하여 사용 중인 객체를 추적하고, 더 이상 참조되지 않을 때 자동으로 메모리를 해제

1. 객체가 생성될 때 참조 카운트를 1로 초기화.
2. 객체를 참조하는 새로운 포인터가 생길 때마다 카운트를 증가시킴.
3. 객체를 가리키는 포인터가 없어질 때마다 카운트를 감소시킴.
4. 카운트가 0이 되면 해당 객체를 자동으로 삭제하고 메모리를 해제.

**<스마트 포인터>**

스마트 포인터는 C++에서 제공되는 자동 메모리 관리 클래스로, 동적으로 할당된 메모리를 자동으로 해제하여 메모리 누수를 방지

std::unique\_ptr: 단일 소유권(Unique Ownership)을 가지는 스마트 포인터로, 하나의 스마트 포인터만이 특정 객체를 소유할 수 있다.

std::shared\_ptr: 공유 소유권(Shared Ownership)을 가지는 스마트 포인터로, 여러 개의 스마트 포인터가 같은 객체를 공유하여 소유할 수 있다.

std::weak\_ptr: 약한 참조(Weak Reference)를 제공하는 스마트 포인터로, shared\_ptr와 함께 사용되어 순환 참조 문제를 해결하는데 사용된다.

순환 참조는 두 개 이상의 객체가 서로를 참조하여 메모리 해제가 올바르게 이루어지지 않는 상황을 말한다. 객체 간 서로를 std::shared\_ptr로 참조할 때는 std::weak\_ptr로 관찰하면서, 순환 참조를 끊을 수 있다. std::weak\_ptr는 객체를 소유하지 않고, std::shared\_ptr가 관리하는 객체를 관찰하기만 함.

**<Allocator>**

Allocator는 컨테이너의 메모리 관리를 커스터마이징하고, 특정 요구 사항에 맞게 최적화하는 데 사용

1. 컨테이너와 메모리 할당을 분리하여 유연성을 높임.
2. 컨테이너의 메모리 할당 및 해제 정책을 사용자가 제어할 수 있음.
3. 특정 메모리 할당 전략을 구현하여 메모리 사용량을 최적화할 수 있음.
4. 특수한 메모리 할당 요구 사항에 대응할 수 있음.

|  |  |  |  |
| --- | --- | --- | --- |
| **문제점 정리** |  | | |
| **해결방안** |  | | |
| **다음주차** | 5주차 | **다음기간** | 2023.7.24 ~ 2023.7.30 |
| **다음주 할일** | - 게임 서버 공부(네트워크 프로그래밍)  - 메모리 관리 복습  - 언리얼 소스코드 분석(Actor) | | |
| **지도 교수**  **Comment** |  | | |