lock

목표

recursive shared mutex를 만들고 안전하게 트랜잭션을 처리할 수 있는 모드를 갖는다.

배경에서 설명한 바와 같이 upgrade 시의 데드락 문제로 인해 boost의 upgrade_lock 과 같은 안전한 구현을 사용할 수 없기 때문에 downgrade는 안전하고 upgrade는 안전하지 않은 구현을 선택했다.

이를 보완하기위해 완전하게 안전한 모드를 추가했다.

배경

proactor를 사용하는 asio 기반의 서버 처리 구조는 다른 게임에서 그 경쟁력이 입증되었다.

post를 통한 타이머와 이벤트 콜백 처리와 함께 여러 코어를 사용하는 단일 장비에서 분산 처리를 배제하고 처리하는 구조로 처리 코드가 단순하면서 동시성을 손쉽게 높일 수 있다는 장점이 가장 크다.

단지, shared state multithreading 구조이기 때문에 thread-safe 하게 만들고 맵 내 엔티티 관리와 같이 중요한 처리 부분의 동시성을 올리면서 동시에 thread-safe 하게 만드는 것이 함께 중요하고 어렵다.

게임들에서 read에 대해 락을 걸지 않고 처리하는 경우가 많은데 이는 캐시를 많이 쓰는 CPU 구조에서 잠재적인 문제들이 다수 발생하게 하는 원인이 될 뿐만 아니라 프로그래밍 할 때 `확신 없이`, `검증 불가능한 상태`로 코딩하게 만든다.

[1] read도 thread-safe 해야 한다.

읽기에 락을 걸면 여러 쓰레드 간에 락 경쟁 (lock contention)이 높아질 가능성이 생기므로 read / write 락을 쓰거나 spinlock을 사용해야 성능 문제가 발생하지 않도록 할 수 있다. spinlock은 CPU 사용량을 전체적으로 많이 올리 수 있기 때문에 전체에 적용하기는 어렵다.

[2] reader / writer 락이 필요하다.

read / write 락의 표준 구현은 shard_mutex이며 windows에서는 Slim Reader Writer Lock으로 구현되어 있고 recursive_mutex, mutex 모두 이를 사용하고 있다.

[3] std::shared_mutex가 적절한 reader / writer 락이다.

한 클래스의 모든 public 함수들이 이제 적절한 락을 사용해야 하므로 한 멤버함수 내에서 다른 멤버 함수를 호출할 때 락을 여러 번 잡게 되므로 recursive해야 한다.
std::shared_mutex의 기본 구현은 recursive 하지 않으므로 그렇게 만들어야 한다.

[4] recurisve 한 std::shared_mutex가 필요하다.

또한 같은 오브젝트의 락에 대해 읽기 중에 쓰기 함수 호출, 쓰기 중에 읽기 함수 호출이 있을 수 있고 생각보다 많으므로 reader --> writer, writer --> reader 간의 락 전환이 중간에 필요하다. 앞의 것을 upgrade, 뒤의 것을 downgrade라고 하자.

[5] upgrade, downgrade가 필요한 std::shared_mutex가 필요하다.

이들 정보를 유지하는 효율적인 방법은 TLS (thread local storage)를 사용하는 것이다. 유지해야 할 정보를 매우 작게 하고 한 메모리 위치에 둬서 쓰레드 캐시에 모두 정보가 로딩되도록 하여 빠르게 처리 가능하게 하면서 [1]~[5]를 만족하는 구현을 찾을 수 있다.

upgrade는 아래 upgrade 데드락에서 설명한 바와 같이 필연적으로
데드락이 발생하고 게임 코드에서 쓰기 모드에서 읽기 함수를 호출할 때
자주 생길 수 있는 상황이므로 unlock_shared()를 사용하는 구현이 필요하다.
그렇게 하면 이전에 읽은 값의 유효성을 보장하지 못 하는 상황이 발생하므로
안전한 모드를 제공해야 한다.

[6] writer 락을 유지하면서 재진입 가능한 락 모드가 필요하다.

설계와 구현

락 모드

xlock :

eXclusive Lock으로 downgrade 가능하다.

xlock_keep :

eXclusive Lock을 유지(keep)하는 모드로 xlock 상태를 유지하면서
재진입 가능하다.

slock :

Shared Lock으로 xlock으로 upgrade 가능하다.

lock_thread_tracer

cache coherence:

매우 작은 정보를 갖는 캐시에 들어갈 수 있는 크기의 배열을 사용하여
락 상태를 추적한다. cache coherence는 성능에 중요하므로 배열을 작게
유지한다. max_lock_depth는 현재 구현과 게임의 용례에서 볼 때 6개를
넘기는 경우는 거의 없을 것으로 보인다.

locked / called:

- locked :
  - when the lock is in lock state either in shared or unique
- called :
  - lock is acquired by explicitly calling lock or lock_shared
  - this is not changed after acquring the lock
  - used to lock again when exit a lock

검증

성질 (properties)

xlock, slock로 안전한 코드를 작성할 수 있고, 한 쓰레드 내에서 재진입이 가능하므로
매우 편하게 사용할 수 있으며, downgrade에서 읽기 유효성이 보장되므로 대체로
이렇게만 사용해도 된다.

완전히 안전한 트랜잭션을 구성하기 위해서는 xlock_keep를 사용할 수 있고,
xlock_keep를 쓰더라도 xlock, slock으로 보호한 코드는 그대로 재진입 가능하며
xlock을 재사용해서 처리한다.

- xlock, xlock_keep, slock 모두 안전한 읽기와 쓰기를 보장한다.

- downgrade간 경쟁은 이전에 읽은 값의 유효성이 보장된다.
- downgrade와 lock 간의 경쟁은 이전에 읽은 값의 유효성이 보장된다.
- downgrade와 upgrade 간의 경쟁은 이전에 읽은 값의 유효성이 보장되지 않는다.
- 이는 unlock을 사용하는 모드 변경에서 논리적으로 해결 가능하지 않다.
- upgrade와 새로운 xlock 간의 경쟁에서 이전 값의 유효성이 보장된다.
- upgrade 간의 경쟁에서 이전 값의 유효성은 하나의 쓰레드에 대해서만 보장된다.
- xlock_keep는 읽기와 쓰기 동작을 포함하여 atomicity를 보장한다.

성능

- std::mutex 비교
  - 2 readers and 2 writers
  - 4 core 장비
  - 1천만 루프
  - lockable:
    - 1456ms, 1172ms, 1160ms
  - mutex:
    - 1167ms, 1199ms, 1214ms

느리지 않고 reader 동시성은 올라가므로 괜찮다.

성질의 증명 (not formal proof)

- xlock, xlock_keep, slock 모두 안전한 읽기와 쓰기를 보장한다.

> 해당 락이 걸린 시점에서 xlock은 하나의 쓰레드만 쓰기를 하고
> slock은 변경이 불가한 상태에서 읽는 것을 보장한다.
> 이는 shared_mutex에서 보장한다.

- downgrade간 경쟁은 이전에 읽은 값의 유효성이 보장된다.

> downgrade() 함수에서 latch를 걸고 진행하고, downgrade는
> 읽기만 한다는 뜻이므로 쓰기를 하는 쓰레드가 없다면
> 안전하다. lock_shared()로 lock()을 추가로 하는 쓰레드가
> 없기 때문에 안전하다.

- downgrade와 lock 간의 경쟁은 이전에 읽은 값의 유효성이 보장된다.

> lock() 함수에서 latch를 체크하고 latch가 걸려 있으면 unlock을 하기 때문에
> lock()을 걸 수가 없고 이는 쓰기가 안 된다는 뜻이므로 읽기와 쓰기간 atomicity가 보장된다.

- downgrade와 upgrade 간의 경쟁은 이전에 읽은 값의 유효성이 보장되지 않는다.

> downgrade()에서 unlock_shared()를 하면 upgrade()에서 다른 쓰레드가
> 쓰기 락을 얻을 수 있으므로 값이 변경된 상태에서 lock_shared()가 될 수 있다.

- 이는 unlock을 사용하는 모드 변경에서 논리적으로 해결 가능하지 않다.

> 이를 막으려면 unlock을 쓰지 않는 구현이 필요하고 그럴 경우 upgrade에서
> 데드락이 발생하므로 선택할 수 없다. (다른 구현 가능성은 없을까?)

- upgrade와 새로운 xlock 간의 경쟁에서 이전 값의 유효성이 보장된다.

> upgrade 중이라면 latch가 걸려 있으므로 다른 쓰레드가 쓰기 락을
> 얻을 수 없다. 따라서, 읽은 값의 유효성이 보장된다.

- upgrade 간의 경쟁에서 이전 값의 유효성은 하나의 쓰레드에 대해서만 보장된다.

> upgrade는 한 쓰레드만 가능하고, 다른 쓰레드는 변경된 값을 나중에 보게 되므로
> 읽은 값의 유효성이 보장되지 않는다.

- xlock_keep는 읽기와 쓰기 동작을 포함하여 atomicity를 보장한다.

> xlock_keep는 항상 쓰기 락을 유지하므로 다른 쓰레드에서 쓸 수 없으므로
> 읽은 값의 유효성이 보장된다.

upgrade 데드락

https://oroboro.com/upgradable-read-write-locks/

Upgradable Write Locks and Deadlock
Using read/write locks with upgrading is tricky. You only want to upgrade a read/write lock in cases where you are sure only one active reader will eventually want to write lock.

Consider the following situation:

Thread 1 acquires a read lock
Thread 2 acquires a read lock
Thread 1 tries to acquire a write lock, and is blocked on thread 2's read lock
Thread 2 tries to acquire a write lock, and is blocked on thread 1's read lock.

위의 설명과 같이 락을 잡고 있는 상태에서 업그레이드를 할 경우 데드락이 발생한다.

lockable / lock_thread_tracer의 구현에서 불가피하게 unlock / unlock_shared를 사용하는 락 업그레이드 / 다운그레이드를 사용했고 이와 같은 이유로 데드락이 발생하지 않는다.

xlock 간의 데드락이 발생하는 부분은 별도로 정리한다.

xlock 데드락

데드락은 피하거나 발생할 경우 해결할 수 있어야 한다. 데드락 탐지 후 처리가 애매하므로 그 전에 최대한 데드락을 피할 수 있는 컨벤션으로 구현하게 해야 한다.

개선

디버깅

- lock_thread_tracer를 lock_tracer에서 볼 수 있게 한다.
  - watch에서 learn::lock_tracer::inst로 조회 가능
- lock_thread_tracer의 상태를 한번에 모아서 출력할 수 있게 한다.
  - to_string() 함수들 추가.
- lock_tracer에서 deadlock을 찾을 수 있게 한다.
  - 효율적인 알고리즘 설계 후 구현 진행

사용

정의

- container와 object

다른 오브젝트를 담는 자료 구조이다. Map 또는 Level과 내부에 포함되는 Sector를 컨테이너로 볼 수도 있다. Sector도 여러 Entity를 담는 container로 볼 수 있다.

흔히 만나는 UserManager, GuildManager와 같은 Manager들도 컨테이너로 볼 수 있다. 다른 관점에서 이들을 처리기로 구현할 수도 있다. 처리기는 처리 논리를 포함하는 클래스이다.

object는 이들 컨테이너에 담기는 대상이다. GuildManager는 Guild를 포함한다.
UserManager는 User를 포함한다. 이들 object는 상태 값을 제공하는 용도로 사용한다.

- entity

  > entity는 내부적으로 slock과 xlock으로 thread-safe하게 한다.

- handler

  > 핸들러는 패킷이나 타이머 호출을 처리하는 함수로 게임 서버 처리의 모든 시작점들이다.
  > 따라서, handler들에서 잡는 락 흐름이 서버의 락 처리 구조가 된다.

- view

  > 게임 서버는 트랜잭션을 처리하는 시스템이 아닌 게임의 현재 상태에 따라 다음 상태로
  > 옮겨가는 시스템이다. 현재 관찰된 값들에 대해 처리하여 다음 상태로 이행한다는 관점에서
  > `관찰된 값들`을 뷰라고 명명한다.

  > view는 게임에서 흔히 쓰는 read uncommitted isolation 수준과 같이 완전한 트랜잭션 처리가
  > 아니더라도 현재 변경된 값들에서 게임 논리가 처리되더라도 거의 일관성(consistency)가 유지
  > 될 수 있다는 점에서 사용할 수 있는 개념이다.

가이드

[1] container는 내부적으로 thread-safe하게 한다.

[2] object는 내부적으로 thread-safe하게 한다.

[3] entity는 내부적으로 xlock/slock으로 thread-safe하게 한다.

[4] handler는 대상 오브젝트에 대한 view들을 얻어서 업데이트 처리를 한다.
즉, 함수 호출로 상태 값들을 얻어서 업데이트 함수들을 호출한다.

[5] 다른 entity에 대한 lock을 풀고 호출하면 데드락을 피할 수 있으며
대체로 안전하다.

[6] concurrent 구조와 같은 더 빠른 구현이 없다면 xlock, slock을 사용한다.

[7] 읽기 / 쓰기간 원자성(트랜잭션)이 필요하면 xlock_keep를 사용한다.