메모리 관리를 위한 lock-free shared\_ptr 구현

요약

(작성)

ABSTRACT

(작성)

1. 서론

온라인 게임의 장르 중 MMORPG(Massively Multiplayer Online Role-Playing Game)는 대규모 다중 사용자 온라인 롤플레잉 게임의 줄임말로 유저가 게임을 꺼도 지속되는 세계관을 가지며, 유저가 세계 안에서 본인이 원하는 플레이를 하는 자유도가 높은 특징을 갖는다. 유저들은 서버에 접속해 동일한 서버에 접속한 다른 여러 유저들과 상호 작용을 하며 플레이한다. MMORPG는 지속적으로 신작이 나오며, 후속작 및 기존 게임의 리뉴얼이 끊임 없이 나올 정도로 많은 유저가 관심을 갖는 장르 중 하나이며, 대표적으로 NCSoft의 리니지가 있다. 이러한 MMORPG의 세계는 많은 유저를 수용할 수 있어야 하며, 랙이 없는 플레이 환경을 제공할 수 있어야한다.

MMORPG에서 서버는 게임의 진행에서 가장 중요한 부분을 차지하는데, 패킷 처리, 컨텐츠 실행객체 관리등의 작업을 수행하며, 유저에게 수월한 플레이를 제공하기 위해 높은 성능과 안정성이 요구된다. 이를 위해 서버는 짧은 시간동안 많은 동작을 처리하여야 하며, 이를위해 대부분의 MMORPG 서버는 C++을 기반으로 한 멀티쓰레드 프로그램으로 구현되지만, 이에 따른 구현의 어려움과 문제점이 생긴다. 잘못된 구현으로 인한 문제점의 대표적인 예로 성능저하와 메모리 관리 문제가 있다. 서버에서 유저로부터 받은 패킷을 처리하는 시간이 길어질수록 성능저하가 심해지며, 동기화를 위한 오버헤드가 클수록 더욱 악화된다. 메모리 관리문제는 메모리 해제 타이밍 문제와 ABA문제가 있다. 메모리 해제 타이밍 문제는 할당된 메모리를 재사용을 위해 해제 할 때 다른 쓰레드들의 참조를 피해야 하는 문제이며, ABA문제는 CAS(compare-and-swap)연산과 관련된 공유객체의 변경을 감지하지 못했을 때 발생한다. 이를 피하기 위해 메모리 재사용을 하지 않는 방법이 있으나, 필요하지 않은 메모리를 계속 점유하고 있는 현상인 메모리 누수를 야기한다.

본 논문에서는 C++11에서 제공하는 메모리 관리 객체인 std::shared\_ptr를 개선해, 서버 프로그래밍에서 사용가능한 lock-free shared\_ptr를 제안한다. 멀티쓰레드에서 ABA문제없이 이용 가능하며 더 나아가 이를 이용해 lock-free 자료구조를 만들 수 있도록 구현하였다.

논문의 구성은 다음과 같다. (작성)

1. 본론

2.1 스마트 포인터 (smart pointer)

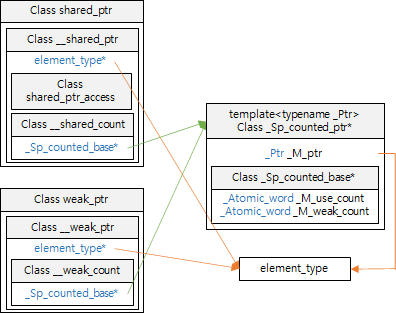
C++을 이용하는 프로그램에서는 new 키워드를 사용해 동적으로 메모리를 할당 받았다면, 반드시 delete 키워드를 사용해 해당 메모리를 해제해야 한다. C++11부터 스마트 포인터를 제공해 사용자의 편리성과 메모리 누수로부터 프로그램의 안전성을 보장한다. 스마트 포인터는 포인터처럼 동작하는 클래스 템플릿으로 더 이상 필요하지 않은 메모리를 자동으로 해제해 준다.

스마트 포인터는 기본 포인터(raw pointer)가 메모리를 가리키도록 new 키워드를 사용해 초기화한 뒤, 해당 기본 포인터를 스마트 포인터에 대입해 사용한다. 스마트 포인터의 수명이 다하면 소멸자는 delete 키워드를 사용해 할당된 메모리를 자동으로 해제하고, 사용자가 따로 메모리를 해제할 필요가 사라진다.

스마트 포인터의 종류는 총 3가지로, 하나의 스마트 포인터만 해당 객체를 소유할 수 있도록 하는 unique\_ptr, 하나의 객체를 참조하는 스마트 포인터의 수(소유자의 수)를 참조하는 shared\_ptr, shared\_ptr 인스턴스가 소유하는 객체에 대한 접근을 제공하지만 소유자의 수에 포함되지 않는 weak\_ptr가 있다. 본 논문에서는 lock-free shared\_ptr을 논하므로 unique\_ptr은 제외한다.

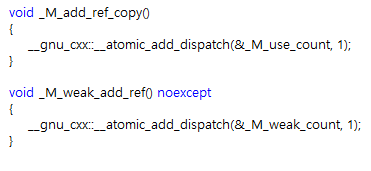
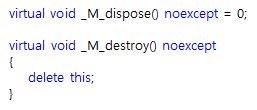
// 너무 자세하게 써 놓았다. 간략하게 설명하도록 [그림1]을 설명할 수 있는 레벨이면 충분, \_gnu\_ 관련된 내용은 다 빼도록

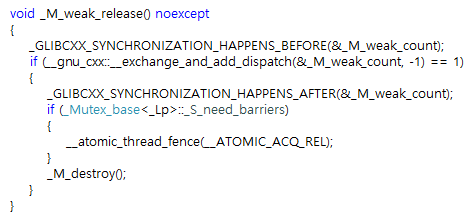
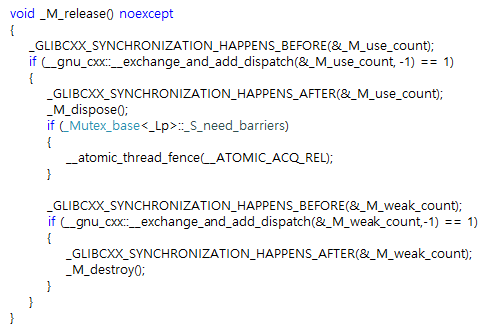
shared\_ptr 구현 코드를 살펴보면, mutex, \_\_gnu\_cxx::\_Lock\_policy, \_\_gnu\_cxx::\_\_default\_lock\_policy, \_\_gnu\_cxx::\_S\_single, \_\_gnu\_cxx::\_S\_mutex가 등장하지만 lock-free와 관련없는 내용이므로 다루지 않는다. 더하여, 기본 포인터(raw pointer) 외에도 deleter와 allocator를 인자로 하는 메소드가 있으며 배열(array)과 다른 타입이름(typename)를 갖는 인수도 존재하나, 본 논문에서는 lock-free shared\_ptr의 확장성을 논하지 않는다.



[그림 1] std::smart\_pointer 클래스 모델

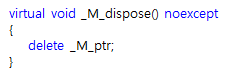
* + 1. class template \_Sp\_counted\_base





\_Sp\_counted\_base는 shared\_ptr에서 관리할 소유자의 수(이하 \_M\_use\_count)와 weak\_ptr을 통해 접근할 접근자의 수(이하 \_M\_weak\_count)를 가지며, \_M\_add\_ref\_copy(), \_M\_release(), \_M\_weal\_add\_copy(), \_M\_weak\_release()함수를 통해 \_M\_use\_count와 \_M\_weak\_count를 원자(이하 atomic)적으로 증가, 감소시켜 스마트 포인터의 수명을 연산한다. release()함수를 통해 \_M\_use\_count가 0이 되어 소유자가 없는 경우 순수 가상함수로 정의된 \_M\_dispose()를 실행하며, \_M\_use\_count뿐만 아니라 \_M\_weak\_count가 0이 되어 접근자가 없는 경우 \_M\_destory()를 통해 자신 메모리(\_Sp\_counted\_base)를 반환한다. \_M\_dispose()또는 \_M\_destory()의 메모리 해제 상황은 스마트 포인터가 가리키고 있는 객체의 마지막 소유자, 접근자가 호출하므로 멀티쓰레드 환경에서 정상적으로 동작한다. 따라서, \_Sp\_counted\_base을 이용하는 shared\_ptr와 weak\_ptr의 수명 연산과 메모리 해제의 동작은 atomic하다고 할 수 있다. *[그림 1]*

* + 1. class template \_Sp\_counted\_ptr



\_Sp\_counted\_ptr는 \_Sp\_counted\_base를 상속받은 클래스 템플릿에 추가로 \_M\_ptr 멤버변수를 갖는다. template<typename \_Ptr>인 \_Sp\_counted\_ptr의 인수로 shared\_ptr(혹은 weak\_ptr)의 객체의 타입이 들어온다. 순수 가상함수로 정의된 \_Sp\_counted\_base의 \_M\_dispose()가 객체(이하 ptr)의 메모리를 반환하도록 구현한다.

* + 1. class template \_\_shared\_count

shared\_ptr 인스턴스의 ptr과 \_Sp\_counted\_base를 갖는 클래스 템플릿 객체로, shared\_ptr의 동작의 구현에 필요한 메소드를 정의한다.

메모리 해제를 위한 \_M\_use\_count뿐만 아니라, deleter와 allocator에 관한 정보로 객체의 생성과 소멸도 관리한다.

* + 1. class template \_\_weak\_count

\_\_shared\_count와 비슷한 구조의 인스턴스의 객체에 대한 클래스 템플릿 객체이다. shared\_ptr는 \_M\_use\_count를 참조해 객체를 관리하는 반면, weak\_ptr는 shared\_ptr의 객체에 대한 접근을 제공하므로 \_M\_use\_count가 아닌 \_M\_weak\_count로 객체를 관리하는 메모리(\_Sp\_counted\_ptr)를 관리한다. 더하여, shared\_ptr가 가리키는 객체에 접근하려는 특성으로 deleter와 allocator에 관한 정보가 요구되지 않는다.

* + 1. class template shared\_ptr
       1. method
    2. class template std::weak\_ptr
  1. lock-free shared\_ptr

// Lock\_Free shared pointer가 필요한 이유, shared\_ptr를 멀티쓸데에서 사용했을 때 생기는 문제

shared\_ptr[2.1.5]은 \_Sp\_counted\_ptr를 통해 ptr을 관리한다. 즉, 하나의 atomic 멤버변수를 이용해 다른 하나의 멤버변수를 관리하는 방식이다.

// Lock Free임을 증명, Correctness를 증명 (증명이 안되면 설명이라도)

* + 1. class template control\_block
    2. class template shared\_ptr
    3. class template weak\_ptr
    4. class template enable\_shared\_from\_this
  1. 벤치마크 프로그램
  2. 벤치마크 프로그램 결과

// owner\_before를 제외한 이유 : control block의 pointer 뿐만 아니라 객체 pointer도 같이 관리해야 해서 구현이 복잡해 짐(성능 저하의 원인)

// swap을 제외한 이유 : 임의의 두개 객체의 동시 lock\_free 실행을 구현해야 하는데 이를 위해서는 Software Transactional Memory를 사용해야 하고 이 방법은 성능 overhead가 너무 크다.

참고문헌

부록