**[Why do we need atomic\_shared\_ptr?](https://www.justsoftwaresolutions.co.uk/threading/why-do-we-need-atomic_shared_ptr.html)**

**Friday, 21 August 2015**

다가오는 [**Concurrency TS**](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4501.html)에 제공되는 새로운 클래스 template중 하나는 atomic\_weak\_ptr과 함께 [**atomic\_shared\_ptr**](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4501.html#atomic)이다. 짐작한 것처럼, std::shared\_ptr와 std::weak\_ptr는 std::atomic<T\*>와 동일하지만, 이것들이 왜 필요할까?

std::shared\_ptr는 이미 reference count를 동기화하고 있지 않은가?

**std::shaerd\_ptr와 multiple threads**

std::shared\_ptr은 각 쓰레드에 shared\_ptr의 자체 사본이 있는 경우에 여러 쓰레드에서 훌륭하게 작동한다. 이 경우 reference count의 변경은 분명하게 동기화되고, 공유 데이터로 하는 작업이 정확하게 동기화되면 모든 것이 작동한다.

class MyClass;

void thread\_func(std::shared\_ptr<MyClass> sp){

sp->do\_stuff();

std::shared\_ptr<MyClass> sp2=sp;

do\_stuff\_with(sp2);

}

int main(){

std::shared\_ptr<MyClass> sp(new MyClass);

std::thread thread1(thread\_func,sp);

std::thread thread2(thread\_func,sp);

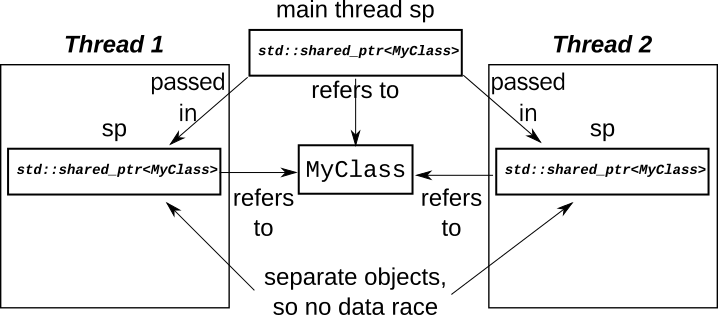
thread2.join();

thread1.join();

}

이 예제에서는, 여러 쓰레드에서 동일한 인스턴스에서 MyClass::do\_stuff()와 do\_stuff\_with()의 동시호출에 대한 안전을 보장해야 하지만, reference count는 정상적으로 처리된다.

**Figure 1 : 분할된 shared\_ptr 인스턴스**



쓰레드 사이에서 단일std::shared\_ptr을 공유하려할 때, 문제가 발생한다.

**쓰레드 사이에서 std::shared\_ptr 인스턴스 공유**

간단한 예제에서 쓰레드 사이에서 공유된 std::shared\_ptr인스턴스 제공할 수 있지만, 대신에 좀 더 흥미로운 것을 살펴보고, 왜 필요한지 더 잘 느낄 수 있다.

각 노드가 다음 노드에 대한 포인터를 가지는, 간단한 단일 linked list를 생각해보자.

class MyList{

struct Node{

MyClass data;

std::unique\_ptr<Node> next;

};

std::unique\_ptr<Node> head;

// constructors etc. omitted.

};

만약 2개의 쓰레드가 이것에 접근한다면, 우리는 선택할 수 있다.

1. 전체 object를 하나의 mutex로 감싸, 오직 하나의 쓰레드만 list에 접근한다. 아니면,
2. 동시 접근을 시도하고 허용할 수 있다.

이 글을 위해서, 우리는 동시 접근을 허용한다고 가정한다.

간단하게 시작하자. 우리는 List를 탐색하고 싶다. 탐색함수 작성은 쉽다.

class MyList{

void traverse(std::function<void(MyClass)> f){

Node\* p=head.get();

while(p){

f(p->data);

p=p->next;

}

}

};

List가 불변이라면, 괜찮지만, 불변list는 관심없다! 우리는 list의 front에서 항목을 제거하고 싶다. 이를 위해 우리가 지원해기 위해서 어떤 변화가 필요한가?

**List의 front에서 제거**

만약 모든 것이 단일 쓰레드인 경우, 항목을 제거하는 것이 쉽다.

class MyList{

void pop\_front(){

Node\* p=head.get();

if(p){

head=std::move(p->next);

}

}

};

목록이 비어있지 않다면, 새로운 head는 이전 head의 다음 pointer이다. 하지만, 이 list에 접근하는 쓰레드가 여러 개이므로, 매우 간단하지 않다.

3개의 항목인 list가 있다고 가정하자.

A->B->C

하나의 쓰레드가 List를 탐색하고, 다른 쓰레드가 첫 항목을 제거하는 경우, 경쟁 가능성이 있다.

1. 쓰레드 X가 list의 head pointer를 읽고, A에 대한 pointer를 얻었다.
2. 쓰레드 Y가 list에서 A를 제거하고 delete했다.
3. 쓰레드 X가 node의 데이터에 접근하려 하지만, node A는 delete되었으므로, dangling pointer와 정의되지 않은 행동을 가진다.

어떻게 고칠까?

가장 먼저 변경할 것은 모든 std::unique\_ptr을 std::shared\_ptr로 만들고, 탐색함수가 원시pointer를 사용하지않고 std::shared\_ptr을 갖도록 하는것이다. 이렇게 하면, node A는 탐색쓰레드가 여전히 reference를 갖기 때문에, 즉시 delete되지 않는다.

class MyList{

struct Node{

MyClass data;

std::shared\_ptr<Node> next;

};

std::shared\_ptr<Node> head;

void traverse(std::function<void(MyClass)> f){

std::shared\_ptr<Node> p=head;

while(p){

f(p->data);

p=p->next;

}

}

void pop\_front(){

std::shared\_ptr<Node> p=head;

if(p){

head=std::move(p->next);

}

}

// constructors etc. omitted.

};

불행하게도 오직 저런 경쟁상태(1번쨰)만 고쳤다. 안좋은 두번째 경쟁상태가 있다.

**두번째 경쟁 상태**

두번째 경쟁 조건은 head 그 자체다. List를 탐색하기위해, 쓰레드 X는 head를 읽어야 한다. 그동안 쓰레드 Y가 head를 update했다. 이것은 Data-race의 정의이므로, 정의되지 않은 동작이다.

우리는 그것을 고치기 위해 무언가를 해야한다.

Head를 보호하기 위해 mutex를 사용할 수 있다. 이것은 pointer가 읽히거나 변경될때의 짧은 시간동안 유지되므로, 전체 list mutex보다 세밀하다. 하지만, std::experimental::atomic\_shared\_ptr을 대신에 사용할 수 있기 때문에, 필요하지 않다.

구현은 atomic\_shared\_ptr와 함께 내부적으로 mutex를 사용할 수 있는데, 이 경우 성능이나 병렬성을 얻지 못하지만, 코드 유지 부하의 감소를 얻는다. 우리는 명시적인 mutex 데이터 member를 가질 필요가 없고, head에 대한 모든 접근마다 올바른 lock&unlock을 기억할 필요가 없다. 대신에, 한 라인의 변경으로 모든 것을 구현할 수 있다.

class MyList{

std::experimental::atomic\_shared\_ptr<Node> head;

};

이제, head에서의 읽기는 head에서의 저장과 경쟁하지 않는다. Atomic\_shared\_ptr의 구현은 문제없이 새로운 값을 읽거나 이전의 값을 얻고록 보장하고, reference count가 올바르게 update되도록 한다.

불핼하게도, 이 코드는 여전히 버그가 있다. 2개의 쓰레드가 동시에 한 node를 제거하려하면 어떻게 될까?

**경쟁 3: 이중 제거**

현재, pop\_front는 수정중인 쓰레드라고 가정하고, 경쟁 상태에 대해 가능성을 열어둔다. 2개의 쓰레드가 동시에 pop\_front를 호출하면, 다음 시나리오를 얻을 수 있다.

1. 쓰레드 X가 head를 읽고, node A에 대한 pointer를 얻는다.
2. 쓰레드 Y가 head를 읽고, node A에 대한 pointer를 얻는다.
3. 쓰레드 X가 head를 head->next로 옮기고, 이는 node B다.
4. 쓰레드 Y가 head를 head->next로 옮기고, 이는 node B다.

따라서, 두 쓰레드가 pop\_front를 호출하지만, 하나의 node만 제거된다. 이거는 버그다.

여기서 수정은 atomic 변수를 사용한 코드를 작성한 프로그래머의 중요한, 어디에나 있는 compare\_exchange\_strong 함수를 이용하는 것이다.

class MyList{

void pop\_front(){

std::shared\_ptr<Node> p=head;

while(p && !head.compare\_exchange\_strong(p,p->next));

}

};

Head를 읽은 후 head가 변경된 경우, compare\_exchange\_strong에 대한 호출은 false를 반환하고, p를 다시 읽는다. 그런 다음 루프를 반복하며, p가 여전히 null인지 확인한다.

이는 두번의 pop\_front를 호출이 서로, 혹은 탐색쓰레드와 함께 문제없이 두개의 node(두개의 노드인 경우)를 제거하는 것을 보장한다.

숙련된 lock-free프로그래머는 지금 당장 “ABA 문제는 어떤가?”라고 생각할지 모른다. 고맙게도, 우리는 걱정할 필요가 없다.

**ABA 문제가 없나?**

맞다. Pop\_front는 ABA문제로 고통받지 않는다. 심지어 새로운 값을 추가하는 함수가 있다고 가정해도, 이전값과 동일한 head의 새로운 값을 얻을수 없다. 이것은 std::shared\_ptr을 사용하는 추가된 이점이다 : 한 쓰레드가 pointer를 갖는동안 이전의 노드는 살아서 유지된다. 따라서, 쓰레드 X는 head를 읽고, node A에 대한 pointer를 읽는다. 쓰레드 X가 해당 포인터를 destroy하거나 재할당 할 때까지는 이 node는 살아서 유지된다. 즉 새로운 노드가 동일한 주소에 할당될수 없고, head가 값 p와 같으면 실제로 동일한 node이고, 동일한 주소를 공유하는 사칭이 아니다.

**Lock-freedom**

앞서 구현에서 mutex를 사용해 atomic\_shared\_ptr에 동기화를 제공할 수 있다고 언급했다. 또 lock-free로 만들어 관리할 수 있다. 이것은 일반 모든 c++ atomic type에 대해 is\_lock\_free() 멤버 함수를 사용해서 테스트 할 수 있다.

Lock-free atomic\_shared\_ptr을 제공하는 이점은 분명해야한다 : mutex 사용을 피하면서, atomic\_shared\_ptr 연산의 동시성에 대해 더 큰 범위가 있다. 특히, 여러 동시 읽기는 방해없이 진행될 수 있다.

단점은 lock-free 코드에 대한 경험이 있는 사람들에게도 분명하다. 코드가 더 복잡하고, 더 많은 작업을 하므로, 충돌이 없는 경우 더 느려질 수 있다. Lock-free 코드의 다른 큰 단점(유지 관리 및 정확성)은 구현자에게 보내진다(책임이 있다).

확장성의 이점이 복잡성을 능가한다고 생각해, Just :: Thread v2.2는 lock-free atomic\_shared\_ptr 구현과 함께 제공된다.