|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | **작성자** | **2014132002 구태균** | **팀명** | - |
| **주차** | **20 주차** | **기간** | **2020.05.25 ~ 2020.05.31** | **지도교수** | **정 내 훈** (서명) |
| **이번주 한일** | * [Why do we need atomic\_shared\_ptr?](https://www.justsoftwaresolutions.co.uk/threading/why-do-we-need-atomic_shared_ptr.html) 정리 * Lock-free shared\_ptr 구현 정리 | | | | |

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

기존의 shared\_ptr을 멀티쓰레드에서 사용하면, reference count는 정상적으로 작동하지만 인스턴스가 정상적이지 않다. 따라서, 인스턴스의 안정성도 보장해야 한다. 방법으로 mutex사용으로 한번에 하나의 쓰레드의 접근만 허용하는 방법과 여러 쓰레드에서 접근을 허용하는 두가지 방법있는데, 주제에 맞게 두번째 방법으로 생각해보자.

탐색의 경우, 불변리스트라면 여러 쓰레드에서 동시에 탐색할 수 있지만 불변리스트는 관심밖의 이야기.

void traverse(…){

Node\* p = head.get();

while(p){

f(p->data);

P = p->next;

}

}

첫번째 문제는 delete.

void pop\_front(){  
 Node\* p = head.get();

if(p){

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

}

}

A->B->C인 리스트에서, 쓰레드 X, Y가 동시에 A를 얻었고 한 쓰레드가 A를 delete할 때, 다른 쓰레드는 A에 대한 dangling pointer를 가진다. 그래서 리스트의 노드를 shared\_ptr로 선언하자.

두번째 문제는 동시 삭제.

void pop\_front(){  
 std::shared\_ptr<Node> p;

while(true){

p = head.get();

if(p && !head.compare\_exchanfe\_strong(p. p->next) return;

}

}

pop\_front()를 여러 쓰레드에서 호출하면, 동일한 head를 얻는다. delete는 shared\_ptr가 관리하지만, 인스턴스는 보장하지 않는다. 따라서 compare\_exchange\_strong 를 이용하자.

세번째 문제는 head 자체.

Class Mylist{

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

};

여러 쓰레드가 동시에 head를 이용하는 것 자체에서 경쟁이 일어난다. 그래서 내부적으로 mutex를 사용하는 std::experimental::atomic\_shared\_ptr을 사용하자. 장점으로는 사용하는 사람이 lock()&unlock()을 명시적으로 사용하지 않아도 되고 정확성이 향상되지만, mutex사용으로 성능이 떨어지고 병렬성이 감소된다.

ABA문제

Shaerd\_ptr을 사용하면서 추가된 이점은 ABA문제를 따지지 않아도 된다는 점이다. 한 쓰레드가 head의 node에 대한 포인터를 갖게 되면, 해당 쓰레드가 해당 포인터를 destroy하거나 재할당 할때까지는 이 node가 delete되지 않는다. 즉, 새로운 노드가 동일한 주소에 할당될 수 없고, head가 값 p와 동일하다면 실제로 동일한 node이다.

Lock-free

Lock-free atomic\_shared\_ptr의 이점은 분명해야한다.

1. Mutex를 사용하지 않는다.
2. 동시성을 보장한다. (특히, 여러 동시 읽기는 방해없이 진행될 수 있어야한다.)

단점으로는

1. 코드가 더 복잡하다.
2. 많은 작업을 하므로, 충돌이 없는경우에도 더 느려진다.
3. 유지 관리와 정확성은 구현자에게 달려있다.

|  |  |  |  |
| --- | --- | --- | --- |
| **문제점 정리** |  | **해결 방안** |  |
| **다음 주차** | **21 주차** | **다음 기간** | **2020.06.01 ~ 2020.06.07** |
| **다음주 할 일** |  | | |
| **지도교수**  **Comment** |  | | |