Lock-Free shared\_ptr와 weak\_ptr의 설계 및 구현

요약

C++11부터 제공하는 동적 메모리 관리 객체인 shared\_ptr는 멀티스레드 환경에서 자유롭게 사용할 수 없는 문제가 있다. 이는 공유 shared\_ptr 객체 접근 시 데이터 레이스가 발생하기 때문이며, 이를 해결하기 위해서는 접근 시 atomic\_store()/atomic\_load()같은 std::atomic 템플릿을 반드시 사용해야 한다. 하지만 이러한 방법은 뮤텍스 사용이 강제되어 멀티스레드 환경에서 큰 성능 저하를 일으킨다. 본 논문에서는 별도의 std::atomic 템플릿의 도움 없이 멀티스레드에서 원자적으로 동작하는 Lock-Free shared\_ptr와 weak\_ptr를 제안한다. 제안하는 두 객체는 다른 메모리 관리 기법들과는 달리 프로그램에서 기존 shared\_ptr의 네임스페이스 수정만으로 사용 가능하다. 기존의 atomic 템플릿을 통한 shared\_ptr 사용시보다 높은 성능을 보임을 실험을 통해 확인하였으며, 이는 본 논문에서 제안한 메모리 관리 객체가 Lock-Free 알고리즘으로 구현된 고성능 C++ 소프트웨어들에 적합하다는 것을 나타낸다.

키워드

abstract

keyword

1 서론

최근 멀티코어 프로세서의 발전으로 Lock-Free 알고리즘이 많은 분야에 사용되고 있으며, 고성능 어플리케이션 제작을 위해서 low level 프로그래밍이 가능한 C++을 이용한다. Lock-Free 알고리즘은 non-blocking 알고리즘 중 하나로, 여러 쓰레드에서 동시에 실행될 때 적어도 하나의 쓰레드는 정해진 시간에 반드시 완료되는 알고리즘이다**[1, +ref]**. 하지만 C++는 가비지 컬렉션(garbage collection)을 제공하지 않아 동적 메모리를 사용할 때 사용자가 직접 메모리를 할당(new)/해제(delete)해야 하는 단점을 갖는다. 이로 인해 C++로 구현된 Lock-Free 알고리즘은 메모리 누수(memory leak)와 ABA문제**[1]**가 야기된다. 이러한 문제를 해결하기 위해 Hazard Pointer**[2]**와 EBR(Epoch Based Reuse)**[3]** 등 여러 가지 메모리 관리 시스템이 제안되었다. 참고문헌 **[4]**는 효율적인 EBR인 DEBRA를 소개하며, Hazard pointer와 DEBRA를 이용한 코드를 보여준다. 두 코드를 살펴보면, HP와 DEBRA가 적용 대상 알고리즘과 밀접하게 연동되어 있어 해당 알고리즘에 대한 이해가 낮으면 HP와 DEBRA을 적용하기 어려운 것을 알 수 있다. 따라서 이러한 시스템들은 알고리즘에서 메모리 사용과 해제를 고려해 고유의 API를 적절한 위치에 추가해야 하는 높은 사용 난이도를 단점으로 가지고 있다.

C++11에서는 동적 메모리를 쉽게 관리할 수 있도록 std::shared\_ptr 템플릿을 제공한다. shared\_ptr는 객체를 공유하는 포인터로, 사용 횟수가 0이 되면 공유하는 객체가 해제(delete)되는 계수 포인터이다. 참고문헌 **[5]**는 두 개의 포인터를 갖는 shared\_ptr의 구조를 보여준다. 두 포인터는 각각 공유 객체와 control\_block를 참조하며, 여기서 control\_block은 사용 횟수와 관련된 use count, weak use count와 deleter를 가진 객체를 말한다. shared\_ptr는 싱글스레드에서 정상적으로 동작하지만, 멀티스레드에서 두 포인터가 원자적으로 수정되지 않아 Load/Store연산 실행 시 데이터 레이스가 발생한다. 여기서 원자적인(atomic) 연산은 연산 실행 도중에 중단되지 않는 연산을 뜻하며, 대표적으로 Lock-Free 알고리즘에서 이용하는 CAS(compare-and-swap)가 있다. 이는 명령어의 실행 도중 운영체제에 의해 중지되지 않으며, 연산이 완료되기 전 다른 스레드는 이전의 메모리 값을 이용하기 때문에, 멀티스레드에서 메모리를 일관성 있게 유지시켜준다**[6]**. 따라서 shared\_ptr를 멀티스레드에서 데이터 레이스 없이 사용하기 위해서 C++11 std::atomic 템플릿을 이용해야 한다**[7]**. 이때 사용하는 std::atomic 템플릿의 API는 atomic\_load()/atomic\_store()로, 전역 뮤텍스를 사용해 객체의 load/store에서 발생하는 데이터 레이스를 방지한다**[8]**. 뮤텍스를 획득하지 못한 스레드들은 대기상태로 전환되며, 뮤텍스를 획득한 스레드의 작업이 완료될 때까지 잠들게 된다. 이는 스레드의 수와 관련없이 한 개의 스레드만 공유 메모리에 접근하게 하므로 멀티스레드의 성능을 악화시킨다. 따라서 std::atomic 템플릿을 통한 기존 shared\_ptr를 멀티스레드 프로그램에서 사용하면 성능 저하를 피할 수 없으며, 이는 4장에서 벤치마크프로그램을 통해 확인할 수 있다. 이러한 shared\_ptr의 문제점을 개선한 std::experimental::atomic\_shared\_ptr가 있다. 하지만 이는 C++20부터 제공되기 때문에 현재 멀티스레드에서 C++에서 제공하는 shared\_ptr를 사용하기 위해서는 std::atomic 템플릿을 이용해야 한다.

본 논문에서는 기존 shared\_ptr의 데이터 레이스를 해결하는 Lock-Free shared\_ptr와 weak\_ptr를 제안한다. 두 객체는 std::atomic 템플릿을 사용하지 않아도 멀티스레드에서 원자적인 실행을 보장하며, 싱글스레드 프로그램에 shared\_ptr를 적용시키는 것과 동일한 방법으로 멀티스레드에서 사용할 수 있어 다른 메모리 관리 시스템보다 사용 난이도가 낮다. 그리고 기존 shared\_ptr와의 성능 비교 실험을 통해 Lock-Free shared\_ptr를 이용한 멀티스레드 프로그램의 향상된 성능을 확인하였다.

본 논문의 구성은 다음과 같다. 2장에서 Lock-Free shared\_ptr와 weak\_ptr의 동작에 대해 설명하고 정확성을 논증하며, 3장에서 Lock-Free control\_block 재사용을 통한 ABA문제 해결에 대해 논한다. 4장과 5장에서 실험 방법과 결론 및 향후 계획에 대해 기술하였다.

2 Lock-Free shared\_ptr와 Lock-Free weak\_ptr의 구현

2.1 구조

기존 shared\_ptr와 weak\_ptr에서 데이터 레이스는 두 개의 포인터가 동시에 수정되지 않아 발생한다. 이를 해결하기 위해 Lock-Free shared\_ptr(LFSP)와 weak\_ptr(LFWP)는 다음과 같은 구조를 가진다.

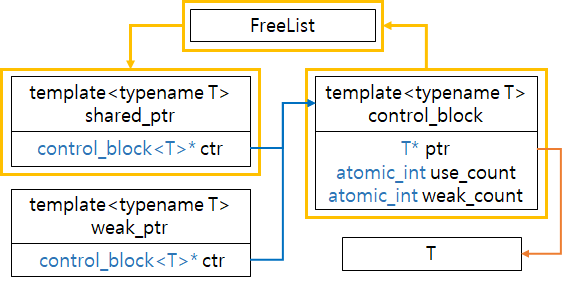
 그림 1

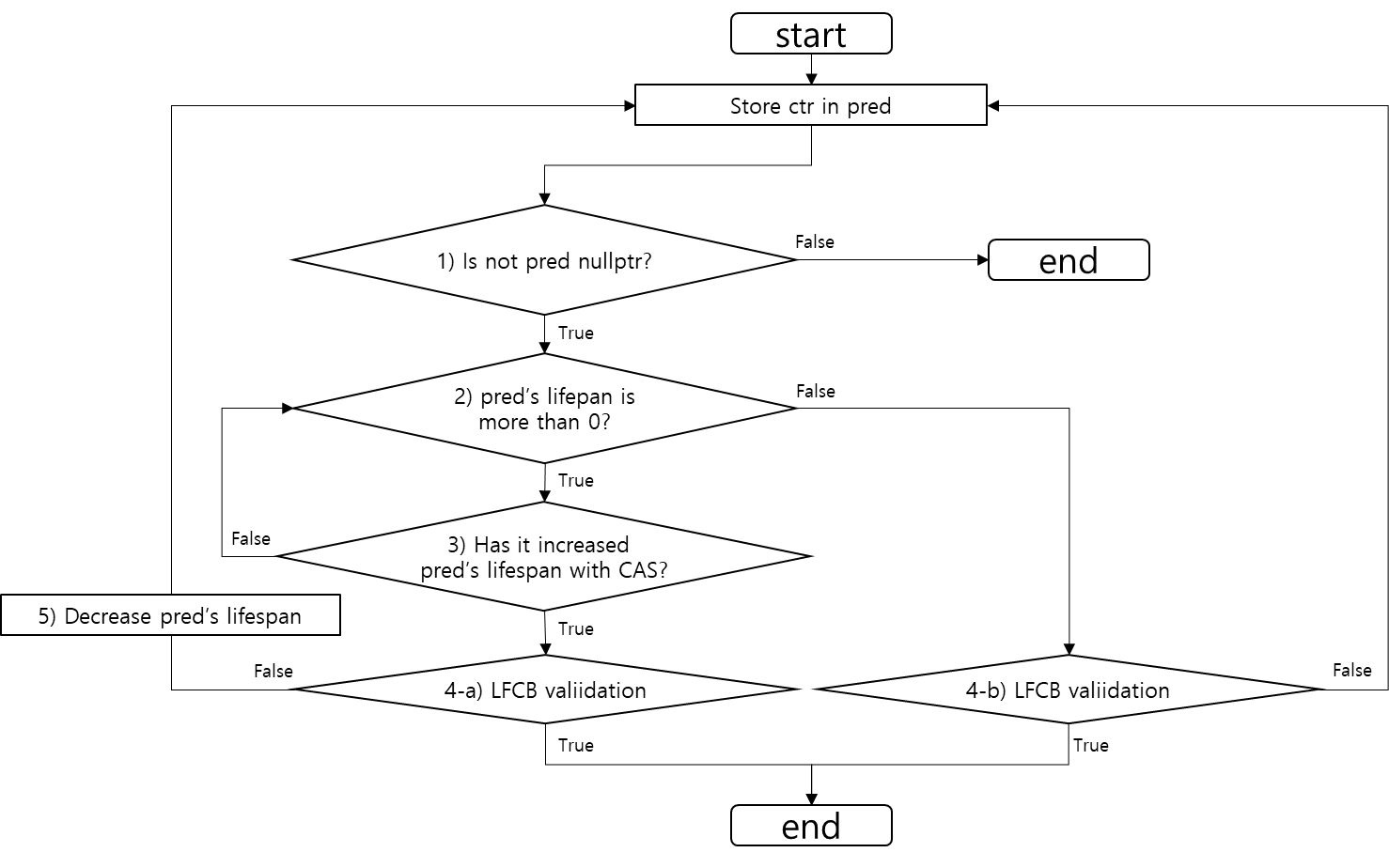
그림 1을 보면 LFSP/LFWP는 Lock-Free control\_block(LFCB)을 참조하는 ctr 포인터만을 가지며, LFCB는 원본 객체 T를 참조하는 ptr 포인터와 참고문헌 **[5]**에서의 use count와 weak use count에 해당하는 use\_count와 weak\_count를 가진다. use\_count와 weak\_count가 0이 되면 공유 객체와 LFCB를 해제(delete)해야 하지만, 다른 쓰레드에서 LFCB를 참조하고 있을 때 해제한다면 ABA문제를 발생시킬 수 있다. 이러한 문제를 해결하기 위해 LFCB를 재사용하였으며, 재사용 시기를 관리하는 자료구조 FreeList는 사용이 끝난 LFCB들을 수거하고, LFSP 생성시 안전하게 재사용한다.

2.2 Lock-Free 구현

LFSP/LFWP의 공유 객체 변경을 위한 Lock-Free 알고리즘은 크게 두가지로 나뉜다. 첫 번째는 공유할 객체의 use\_count나 weak\_count를 증가시키는 addcopy 알고리즘이고, 두 번째는 ctr을 변경하는 ‘=’ operator 알고리즘이다. LFSP와 LFWP의 addcopy 알고리즘과 operator 알고리즘은 사용하는 변수(use\_count와 weak\_count)의 차이만 있으므로, 2.2.1 addcopy 알고리즘과 2.2.2 operator 알고리즘에서는 LFSP에 대해서만 논하였다.

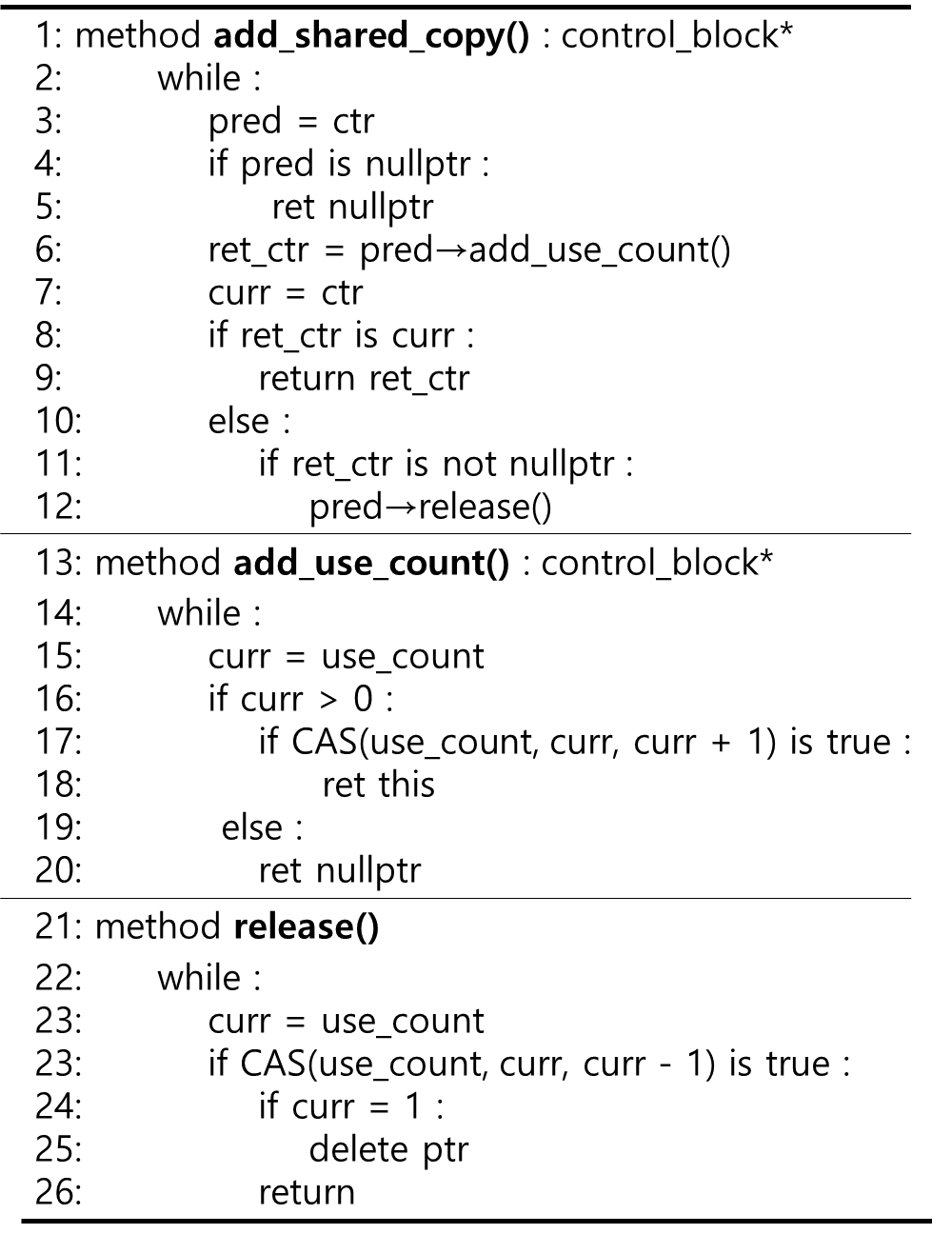
2.2.1 addcopy 알고리즘

addcopy 알고리즘은 LFSP가 공유하는 객체의 use\_count를 미리 증가시켜 LFCB가 재사용되지 않음을 보장한다. 하지만 알고리즘 실행 도중에 LFSP가 다른 스레드에 의해 다른 객체를 공유한다면, 증가시킨 use\_count와 공유 객체가 다를 수 있다. 이를 확인하기 위해 알고리즘 마지막 단계에서 LFCB 유효성을 검사(LFCB validation)한다. 다음 그림 2는 addcopy 알고리즘의 순서도를 보여준다.

그림 2

addcopy 알고리즘은 위와 같이 5개의 과정이 있으며, 그림 2의 1)부터 4)까지의 과정이 만족되어야 LFSP가 공유하는 객체의 lifespan를 안전하게 증가시킬 수 있다. 이해를 돕기 위해 lifespan을 use\_count(uc)라 하였다. 1)은 LFSP가 객체를 공유하는지 확인한다. 만약 ctr이 nullptr로 객체를 공유하지 않다면 알고리즘이 종료되며, 이는 공유하는 객체가 없음을 의미한다. LFSP가 객체를 공유하고 있다면 2)와 3)의 순서와 같이 LFCB의 uc가 0보다 큰 값인지 검사한 뒤 CAS를 이용해 증가시킨다. 이는 다른 스레드로부터 LFCB가 재사용되지 않음을 확인한 뒤 uc를 증가시켜, 1 이상의 uc를 가진 LFCB가 재사용으로부터 안전함을 보장할 수 있기 때문이다. uc를 증가시켰다면 4-a)에서 LFCB 유효성을 검사하며, 여기서 LFCB가 유효하다는 것은 LFSP가 공유하고 있는 객체의 uc를 증가시켰음을 뜻한다. 만약 LFCB가 유효하지 않다면 uc를 감소시켜 초기 상태로 돌아가는 5)를 실행한다. 반대로 LFCB 재사용을 뜻하는 uc가 0인 경우 4-b)에서 LFCB 유효성을 검사한다. 여기서 LFCB가 유효하다면 LFSP가 객체를 공유하지 않으며, uc를 증가시키지 않음을 뜻한다. 따라서 이는 LFCB가 유효하지만 공유하는 객체가 없음을 의미한다. LFCB 유효성 검사와 관련된 내용은 2.2.1.1 LFCB 유효성 검사에서 다룬다.

다음 그림 3은 addcopy 알고리즘을 이용한 대표적인 메소드 LFSP::add\_shared\_copy()의 의사코드다.

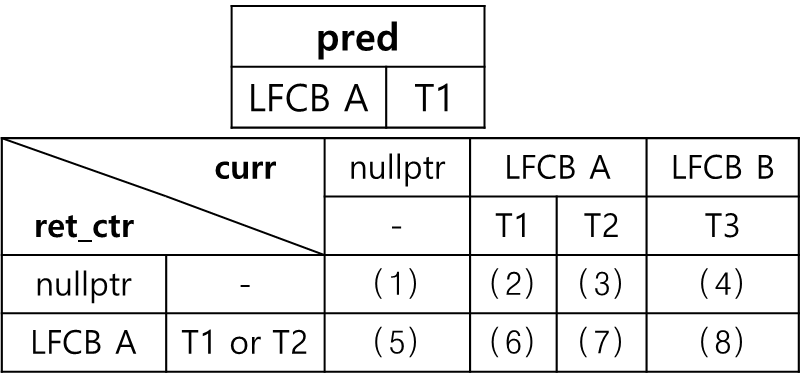
그림 3

add\_shared\_copy()는 uc를 증가시킨 경우 LFCB를 가리키는 curr를 반환하며, uc를 증가시키지 못한 경우 nullptr를 반환한다. 따라서 반환된 값이 nullptr가 아닌 경우 addcopy 알고리즘이 성공했음을 알 수 있다. add\_shared\_copy()의 이해를 돕기 위해 그림 2와 3을 함께 보자. 그림 2의 1)에 해당하는 line 4에서는 ctr이 nullptr라면 nullptr를 반환하며, 이는 LFSP가 객체를 공유하고 있지 않음을 뜻한다. line 6의 add\_use\_count()는 그림 2의 2)와 3)에 해당하는 LFCB의 멤버 메소드이다. line 13~ 20을 살펴보면 add\_use\_count()는 uc가 0보다 큰 경우 CAS를 이용해 uc를 1 증가시킨 후 LFCB 포인터를 반환하며, uc가 0 이하인 경우 nullptr을 반환하는 것을 볼 수 있다. 따라서 반환된 값을 통해 uc 증가 성공과 실패를 확인할 수 있으며, 이는 각각 그림 2의 3)의 True와 2)의 False를 의미한다. line 6에서 ret\_ctr은 add\_shared\_copy()의 반환 값, line 7에서 curr은 add\_shared\_copy() 이후 ctr을 참조하는 지역 포인터이며, 두 포인터는 line 8에서 LFCB 유효성 검사에 이용된다. LFCB 유효성 검사는 ret\_ctr과 curr을 비교하며, 동일한 경우 curr을 반환한다. 여기서 add\_use\_count()가 실패한 ret\_ctr과 LFSP가 객체를 공유하지 않는 curr은 nullptr이므로 nullptr을 반환한다. 따라서 add\_shared\_copy()의 성공과 실패는 반환된 값을 확인해야 알 수 있다. 만약 LFCB 유효성 검사가 실패하면 release()를 호출한다. release()는 그림 2 5)에 해당하는 LFCB의 멤버 메소드로, uc를 1 감소시키고 uc가 0인 경우 객체의 메모리를 해제한다(line 21~ 26). 다음으로 LFCB 유효성 검사를 통해 addcopy 알고리즘의 정확성을 논증하겠다.

2.2.1.1 LFCB 유효성 검사

이번 절에서는 그림 3에서의 LFCB 유효성 검사에서 발생할 수 있는 모든 상황을 가정하며, 이를 통해 LFSP가 공유하는 객체의 uc를 증가시키는 addcopy 알고리즘의 정확성을 논증한다. 그림 3에서 LFCB가 유효하다는 것은 ret\_ctr과 curr이 동일한 경우로, LFSP가 공유하는 객체의 uc를 증가시켰음을 보장한다. 다음은 그림 3 line 8에서의 LFCB 유효성 검사에 대한 가정이다.

그림 3 line 3에서 pred는 LFCB A를 참조하며, T1 객체를 공유한다고 가정하자. line 6에서 ret\_ctr은 LFCB A와 nullptr를 참조할 수 있으며, line 7에서 curr은 다른 스레드에 의해 ctr이 수정되지 않은 경우 LFCB A를 참조한다. 하지만 여기서 중요한 것은 LFCB A가 T1 객체를 가리키는 것을 알 수 없다는 점이다. curr이 ctr의 값을 저장하기 전 다른 스레드에 의해 LFCB A가 재사용되어 T2 객체를 가리킨 뒤, ctr이 이를 참조할 수 있기 때문이다. 따라서 curr은 T1이나 T2를 가리키는 LFCB A를 참조할 수 있으며, 다른 스레드에 의해 수정되어 nullptr와 T3 객체를 가리키는 LFCB B를 참조할 수 있다. 위의 모든 가정은 그림 4와 같이 표현할 수 있다.

 그림 4

설명을 위해 그림 2와 4를 함께 보자. 그림 4에서 ret\_ctr이 nullptr인 경우 uc 증가 실패를 의미한다. 따라서 LFSP가 공유하는 객체가 없는 상황 (1)은 LFCB가 유효하며, 상황 (2), (3)과 (4)는 LFSP가 객체를 공유하고 있으므로 LFCB가 유효하지 않다. 반대로 ret\_ctr가 LFCB A인 경우 uc가 증가했음을 의미한다. 그러므로 curr이 여전히 LFCB A를 가리키는 상황 (6)은 LFCB가 유효하며, nullptr과 LFCB B를 가리키는 상황 (5)와 (8)은 LFCB가 유효하지 않다. 이제 curr이 재사용된 LFCB A를 가리키는 상황 (7)에 대해 논해보자. pred는 그림 2 1)이 성공했으므로 LFCB A를 가리킨다. 따라서 상황 (7)의 LFCB A 재사용은 1)과 4) 사이에서 발생했음을 알 수 있다. 다음은 LFCB A가 재사용된 시점에 대한 가정이다.

가정 1) LFCB A가 그림 2 1)과 3) 사이에서 재사용 되었다고 가정해보자. 3)에서 LFCB A는 T2 객체를 가리킨 이후 uc가 증가했다. 따라서 3)의 True에서 ret\_ctr(LFCB A)는 T2 객체를 가리키며, curr(LFCB A)도 T2 객체를 가리키고 있으므로 LFCB가 유효하다. 만약 1)과 3) 사이에서 LFCB A의 uc가 0이 되었다면, 3)과 2)가 실패하므로 상황 (3)과 동일한 결과를 가진다.

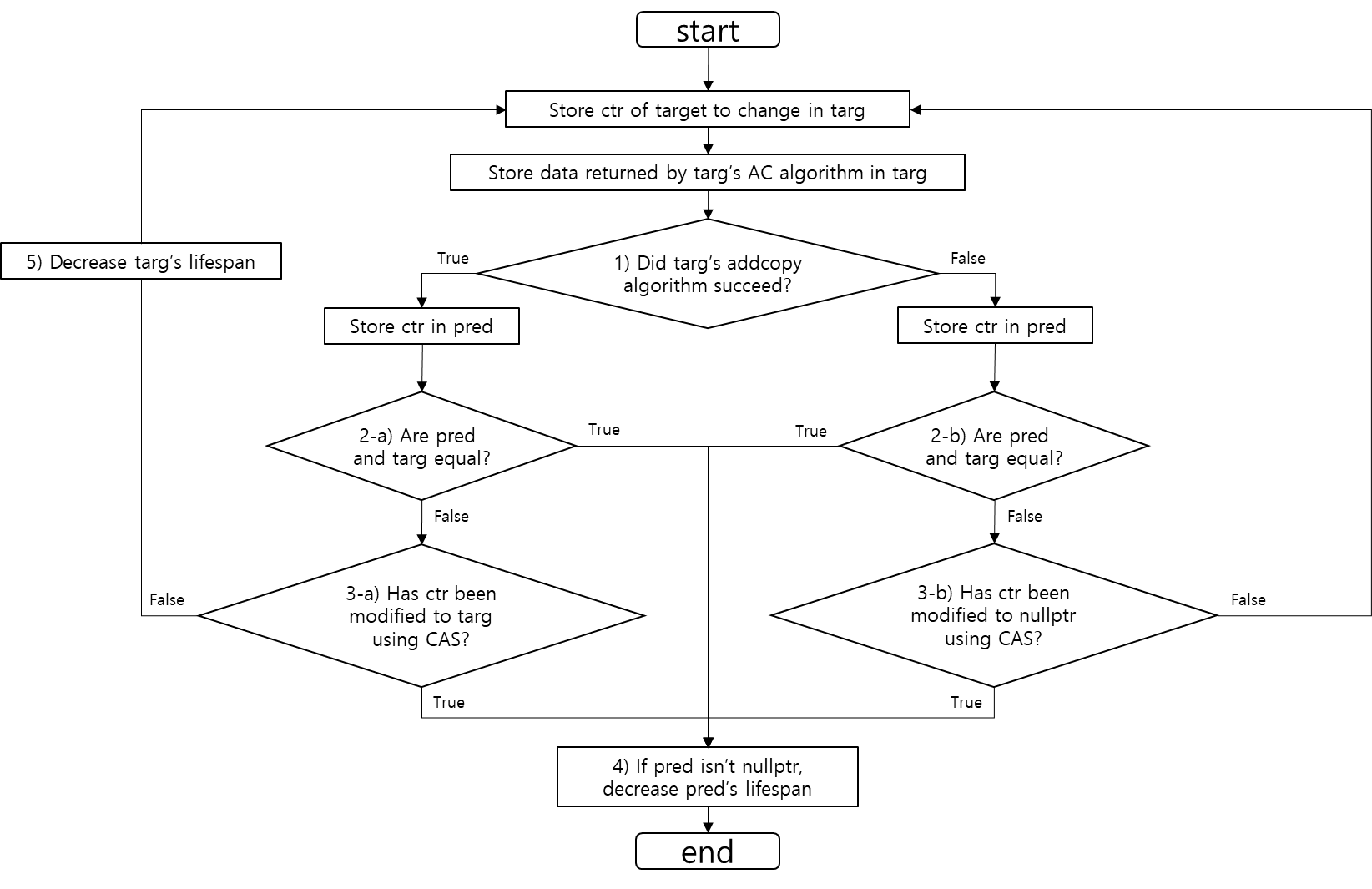
가정 2) LFCB A가 그림 2 3)과 4) 사이에서 재사용 되었다고 가정해보자. 3)에서 T1 객체를 가리키는 LFCB A의 uc가 증가해 1 이상의 값을 가지므로 3) 이후에 LFCB A는 재사용될 수 없다. 따라서 3) 이후에 LFCB A가 재사용되었다는 가정 2)는 모순이다.

가정 1)과 가정 2)를 통해 상황 (7)에서 LFCB A는 1)과 3)사이에서 재사용되며, 3)에서 T2 객체에 대한 uc를 증가시키는 것을 알 수 있다. 그러므로 ret\_ctr과 curr이 참조하는 LFCB A가 T2 객체를 가리키는 상황 (7)은 LFCB가 유효하다.

따라서 LFCB 유효성 검사는 상황 (1), (6)과 (7)에서 성공한다. 상황 (1)은 nullptr, 상황 (6)과 (7)은 LFCB A를 반환하며, 이는 addcopy 알고리즘의 성공과 실패를 의미한다. LFCB 유효성 검사를 통해 addcopy 알고리즘에서 증가된 uc는 LFSP가 공유하는 객체에 대한 uc임을 보장할 수 있다.

2.2.2 operator 알고리즘

operator 알고리즘은 LFSP가 참조하는 LFCB(ctr)를 변경하는 알고리즘이다. 다음 그림 5는 operator 알고리즘의 순서도를 보여준다.

 그림 5

operator 알고리즘은 위와 같이 addcopy 알고리즘 이후 그림 5의 1)부터 4)까지의 과정이 만족되어야 안전하게 ctr을 수정했음을 보장할 수 있다. 이는 ctr을 수정하기 전 addcopy 알고리즘을 실행해 참조할 LFCB가 재사용되지 않음을 보장하며, 그림 5 1)과 같이 addcopy 알고리즘의 결과에 따라 두가지 과정으로 나뉜다. addcopy 알고리즘의 성공은 공유할 객체의 uc가 증가되었음을 의미하므로 LFSP가 해당 객체를 공유해야 하며, 실패는 공유할 객체가 없음을 의미하므로 LFSP는 어떠한 객체도 공유하면 안된다. 그림 5 2-a)와 2-b)는 알고리즘의 최적화를 위한 과정으로, 이들은 각각 LFSP의 공유 객체가 공유할 객체와 동일한지 확인하고 LFSP가 이미 객체를 공유하지 않는지 확인한다. 두 과정이 만족된다면 ctr이 이미 targ와 nullptr를 참조하고 있으므로 ctr 수정 과정을 생략할 수 있다. ctr이 수정되어야 한다면 그림 5 3-a)와 3-b)에서 targ와 nullptr로 수정된다. 여기서 ctr은 CAS를 이용해 수정되며, CAS가 성공하면 LFSP의 공유 객체가 변경되었음을 의미한다.

LFSP의 pred(ctr)와 targ가 동일한지 확인한다. 2-a)와 2-b)의 True는 을 의미한다.

이때 ctr(pred)은 targ를 가리키고 있다. 따라서 addcopy 알고리즘에 증가시킨 targ의 uc를 감소시키는 4) 과정을 거친다. 2-b)의 True는 LFSP가 공유할 객체가 없고, LFSP도 이미 객체를 공유하지 않음을 의미한다. 이때 addcopy 알고리즘에 증가시킨 uc가 없으므로 알고리즘은 종료된다. LFSP의 ctr은 3-a)와 3-b)에서 CAS를 이용해 수정된다. 3-a)의 CAS가 성공한 경우 LFSP의 공유 객체가 변경됨을 의미하므로 이전 공유 객체(pred)의 uc를 감소시킨다. 반대로 CAS가 실패한 경우 ctr이 다른 스레드에 의해 변경됨을 의미한다. 이는 알고리즘의 실패를 의미하므로 addcopy 알고리즘에서 증가한 targ의 uc를 감소시켜 초기 상태로 돌아가는 5)의 과정을 거친다. 3-b)는 ctr을 nullptr로 수정하는 CAS를 이용한다. 따라서 CAS 성공은 LFSP가 객체를 공유하지 않음을 의미하며 3-a)의 CAS 성공과 동일하게 4) 과정을 거친다. 3-b)의 CAS 실패는 addcopy 알고리즘이 실패했으므로 5)의 과정이 필요하지 않다.

다음은 그림 6은 OP 알고리즘을 이용한 대표적인 메소드인 LFSP::operator=(LFSP)의 의사코드다.

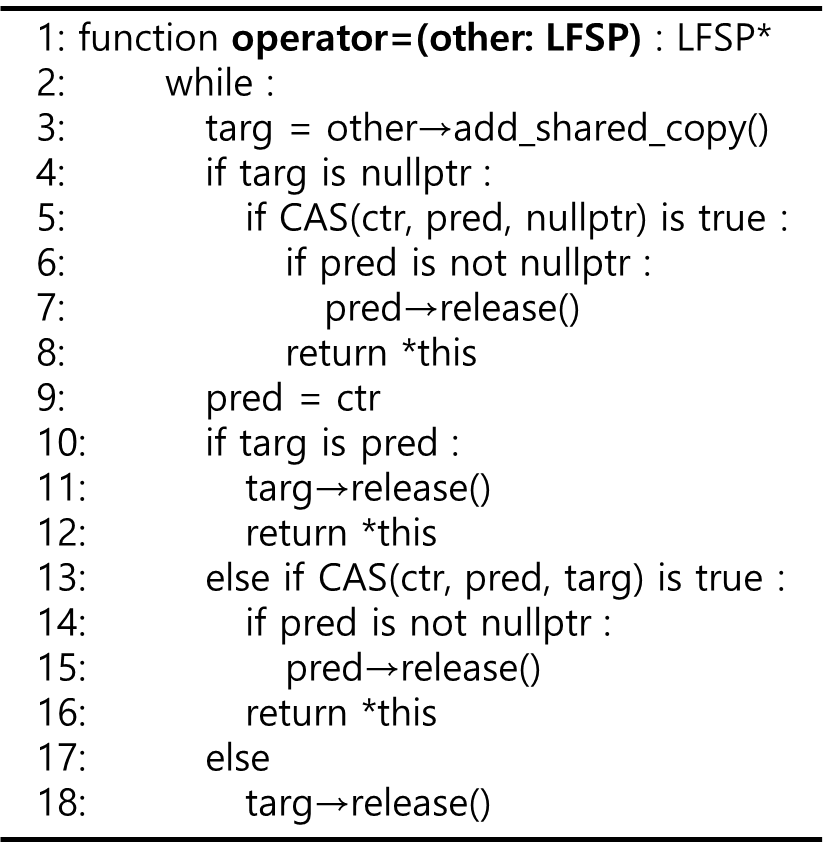
 그림 6

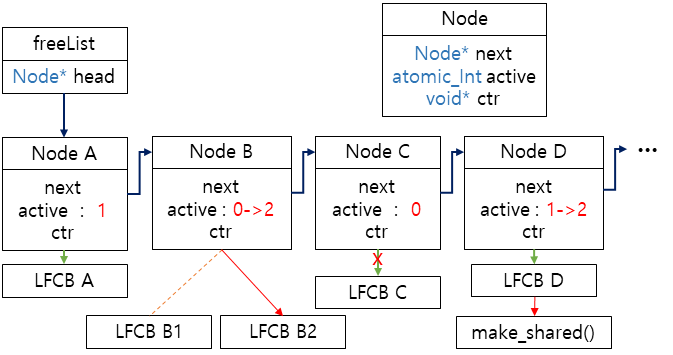
그림 6에서의 release()는 OP 알고리즘에서 pred와 targ의 use\_count를 감소시키는 [3]과 [4]에 해당된다. release()에 대한 자세한 설명은 그림 3을 참고하자.

operator=(LFSP)가 OP 알고리즘을 만족함을 보기위해 그림 5와 6을 함께 보자. line 4에서 targ는 AC 알고리즘인 add\_shared\_copy()의 반환 값을 가리키고 있다. line 5와 같이 targ가 nullptr를 가리키는 경우 AC 알고리즘의 실패를 의미하기 때문에 [2-b] 과정을 가진다. [2-b] 이후 ctr은 nullptr을 가리키며, [3]의 과정을 거친다(line 6~ 9). line 10과 같이 targ와 pred가 동일하다면, LFSP가 이미 동일한 LFCB를 참조하고 있음을 의미하므로 [2-a]의 과정이 요구되지 않는다. 따라서 targ의 release()를 호출해 line 4에서 증가시킨 targ의 uc를 감소시킨다(line 11~ 12). line 13은 CAS를 이용해 ctr을 targ로 수정하는 [2-a]에 해당한다. CAS의 성공여부에 따라 [3]과 [4]의 과정을 거친다(line 13~ 18).

3 ABA문제 해결

1. aba를 freelist로 해결함.
2. freelist는 ~로 동작하는 객체다.
3. freelist가 LFSP/LFWP를 구현한 namespace에 구현되어있어 사용자는 FreeList를 직접 이용할 필요가 없다
4. 그림
5. 구조 설명
6. 코드
7. 코드설명

FreeList는 Lock-Free 알고리즘으로 구현된 연결리스트로, Node를 재사용하여 ABA문제가 발생하지 않는다. namespace LF에 정의된 freelist 변수 한 개로 프로그램 전체의 LFCB를 관리하며, 수명이 다한 LFCB는 freelist에 등록된 뒤 LF:: make\_shared()에서 재사용된다.



[Fig. 10] LF::FreeList freelist

freelist를 구성하는 Node의 멤버변수는 리스트를 연결하는 next, 재사용할 LFCB를 저장하는 ctr, Node의 상태를 나타내는 active가 있다. [Fig. 10]을 보면, active는 ctr이 재사용이 가능(활성)하면 1(Node A), 이미 재사용 되어서 더 이상 ctr의 값이 의미가 없는(비활성) 경우 0(Node C), ctr이 수정중이라면 2의 값을 가지며(Node B, D), 이는 원자적으로 수정되어 멀티쓰레드에서의 안정성을 보장한다.

FreeList는 Node의 active값을 변경하기 위한 **CAS()**(FCA)와 리스트에 Node를 삽입하기 위한 **CAS()**(FCN)를 가진다.

**Add\_FreeListNode()**는 재사용할 LFCB를 비활성 Node에 등록한다. 리스트를 순회하며 비활성 Node를 검색(FAN6)해 수정 Node로 변경(FAN7)하고, 재사용할 LFCB를 ctr에 등록(FAN8, [Fig, 10] Node B)한 뒤, 활성 Node로 변경(FAN9)한다. 리스트를 순회하며 비활성 Node를 검색하지 못한 경우(FAN3,4) 재사용할 LFCB을 등록한 Node를 생성(FAN12)해 리스트의 말단 Node를 검색(FAN14)한 뒤 삽입(FAN15)한다.

**Alloc()**은 리스트에 등록된 재사용 가능한 LFCB를 반환하는 메소드이다. 리스트를 순회하며 활성 Node를 검색(FAL3)하고, 이를 수정 Node로 변경(FAL4)하며, 해당 Node의 ctr이 가리키는 LFCB의 포인터를 반환(FAL5, 7, [Fig.10] Node D)한다. 활성 Node를 찾지 못한 경우 nullptr를 반환(FAL9)해 재사용할 LFCB가 없음을 알린다.

FreeList는 재사용할 LFCB를 등록하기 위해 비활성 Node를 재사용하며, 비활성 Node가 없는 경우 리스트에 Node를 원자적으로 삽입(FAN15)한다. Node의 제거가 없는 FreeList는 이러한 이유로 ABA문제가 발생하지 않는다.

4 실험

표 일반포인터, shared\_ptr, atomic shared\_ptr, lock-free shared\_ptr

4.1 shared\_ptr vs atomic 이용한 shared\_ptr

4.2 atomic 이용한 shared\_ptr vs Lock-Free shared\_ptr

5 결론

reference

**[1]** A lock-free multiprocessor OS kernel

**[1]** K. Fraser, "Practical Lock-Freedom," Ph.D. dissertation. King's College, University of Cambridge, pp. 33, 2003

**[2]** Maged M. Michael, “Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects”, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 15, NO. 2004, 08

**[3]** K. Fraser, "Practical Lock-Freedom," Ph.D. dissertation. King's College, University of Cambridge, pp. 79-81, 2003

**[4]** Trevor Alexander Brown, “Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way”, PODC '15: Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, July 2015, https://doi.org/10.1145/2767386.2767436

**[5]** Bjarne Stroustrup, “The C++ Programming Language (Fourth Edition)”, Addison-Wesley, pp 990-995, 2013

**[6]** https://docs.microsoft.com/ko-kr/windows/win32/dxtecharts/lockless-programming?redirectedf-rom=MSDN

**[7]** std::shared\_ptr : https://en.cppreference.com/w/cpp/memory/shared\_ptr

**[8]** std::atomic : https://en.cppreference.com/w/cpp/memory/shared\_ptr/atomic

**[9]** H. Massalin, “C. Pu, A lock-free multiprocessor OS kernel”, Tech. Rep. CUCS-005-91, Columbia University, New York, 1991.