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. 공유되는 shared\_ptr 객체 접근 시 데이터 레이스가 발생하기 때문이다. 이러한 이유로 std::atomic 템플릿의 멤버함수 atomic\_store()/atomic\_load()와 함께 사용해야 한다. 하지만 이러한 방법은 단일 뮤텍스 내부구현으로 인해 멀티스레드 환경에서 큰 성능저하를 일으킨다. 본 논문에서는 뮤텍스를 사용하지 않고 멀티스레드 환경에서 atomic 템플릿의 도움 없이 사용 가능한 Lock-Free shared\_ptr와 Lock-Free weak\_ptr를 제안한다. 제안하는 두 객체는 내부적으로 데이터 레이스가 발생하지 않아 <<않을 뿐만 아니라 ABA문제로부터도 자유로우며 이는 >> Lock-Free 알고리즘으로 구현된 고성능 C++ 소프트웨어 시스템에 유용할 것으로 기대된다.

키워드

abstract

keyword

1 서론

최근 멀티코어 프로세서의 발전으로 Lock-Free 알고리즘이 많은 분야에 사용되고 있다. C++은 low level 프로그래밍 언어로 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)**[1]** 등 여러 가지 메모리 관리 시스템이 제안되었다. 이 시스템들은 알고리즘에서 메모리 사용과 해제를 고려해 고유의 API를 적절한 위치에 추가해야 하는 높은 사용 난이도를 단점으로 가지고 있다. **[4]**는 효율적인 EBR인 DEBRA를 소개하며, Hazard pointer와 DEBRA를 이용한 search() 코드를 보여준다. 두 코드를 살펴보면, HP와 DEBRA이 알고리즘과 밀접하게 연동되어 있어 알고리즘에 대한 이해가 낮으면 HP와 DEBRA을 적용하기 어려운 것을 알 수 있다.

C++에서는 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 객체와 control\_block를 참조하며, 여기서 control\_block은 사용 횟수와 관련된 use count, weak use count와 deleter를 가진 객체를 말한다. 두 포인터가 동시에 수정되지 않는 이유로, shared\_ptr는 멀티스레드에서 Load/Store연산 실행 시 데이터 레이스가 발생한다. 이러한 멀티스레드에서의 데이터 레이스를 방지하기위해 C++11은 std::atomic 템플릿을 제공한다. 따라서 shared\_ptr를 멀티스레드에서 데이터 레이스 없이 사용하기 위해서는 std::atomic에 정의된 atomic\_load()/atomic\_store()를 이용해야 한다**[6]**. atomic\_load()/atomic\_store()는 객체의 load/store에서 발생하는 데이터 레이스를 방지하기위해 전역 뮤텍스를 사용하는 API이다**[7]**. 뮤텍스를 획득하지 못한 스레드들은 대기상태로 전환되며, 뮤텍스를 획득한 스레드의 작업이 완료될 때까지 잠들게 된다. 뮤텍스를 사용하면 스레드의 수와 관련없이 한 개의 스레드만 공유 메모리에 접근하며, Convoying과 우선순위역전 등의 문제를 발생시켜 멀티스레드에서의 성능을 악화시킨다. 따라서 std::atomic을 이용한 C++ shared\_ptr의 사용은 병렬성을 높여 성능을 향상시키는 멀티스레드 환경에 부합하지 않는다. 앞서 말한 shared\_ptr의 문제점을 개선한 std::experimental::atomic\_shared\_ptr가 있지만, 이는 C++20부터 제공되며 현재 컴파일러가 구현되어 있지 않다. 따라서 현재 멀티스레드 프로그램에서 shared\_ptr를 사용할 때 std::atomic의 도움을 받아야 한다.

본 논문에서는 기존 shared\_ptr의 데이터 레이스를 해결하는 Lock-Free shared\_ptr와 weak\_ptr를 제안한다. 두 객체는 싱글 스레드 프로그램에 shared\_ptr를 적용시키는 것과 동일한 방법으로 멀티스레드에서 사용할 수 있어, 구현하는 알고리즘에 대한 깊은 이해가 필요 없으므로 다른 메모리 관리 시스템보다 사용 난이도가 낮다. 그리고 멀티스레드 환경에서의 실험을 통해 기존 shared\_ptr와의 성능을 비교해, Lock-Free shared\_ptr을 이용한 멀티스레드 프로그램의 성능 향상을 확인할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서 Lock-Free shared\_ptr와 weak\_ptr의 동작에 대해 설명하고 정확성을 논증한다. 3장에서 LFCB 재사용을 통한 ABA문제 해결에 대해 논하며, 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\_ptr(SP)와 weak\_ptr(WP)에서 데이터 레이스는 두 개의 포인터가 동시에 수정되지 않기 때문에 발생한다. 따라서 이를 해결하기 위한 Lock-Free shared\_ptr(LFSP)와 weak\_ptr(LFWP)는 다음과 같은 구조를 가진다.

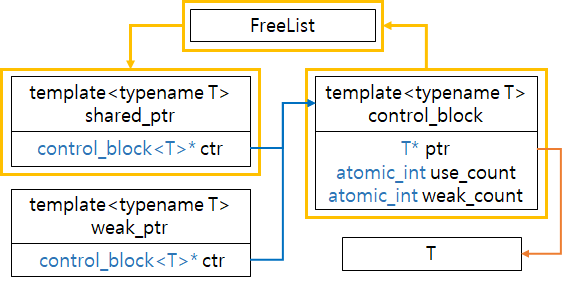
 그림 1

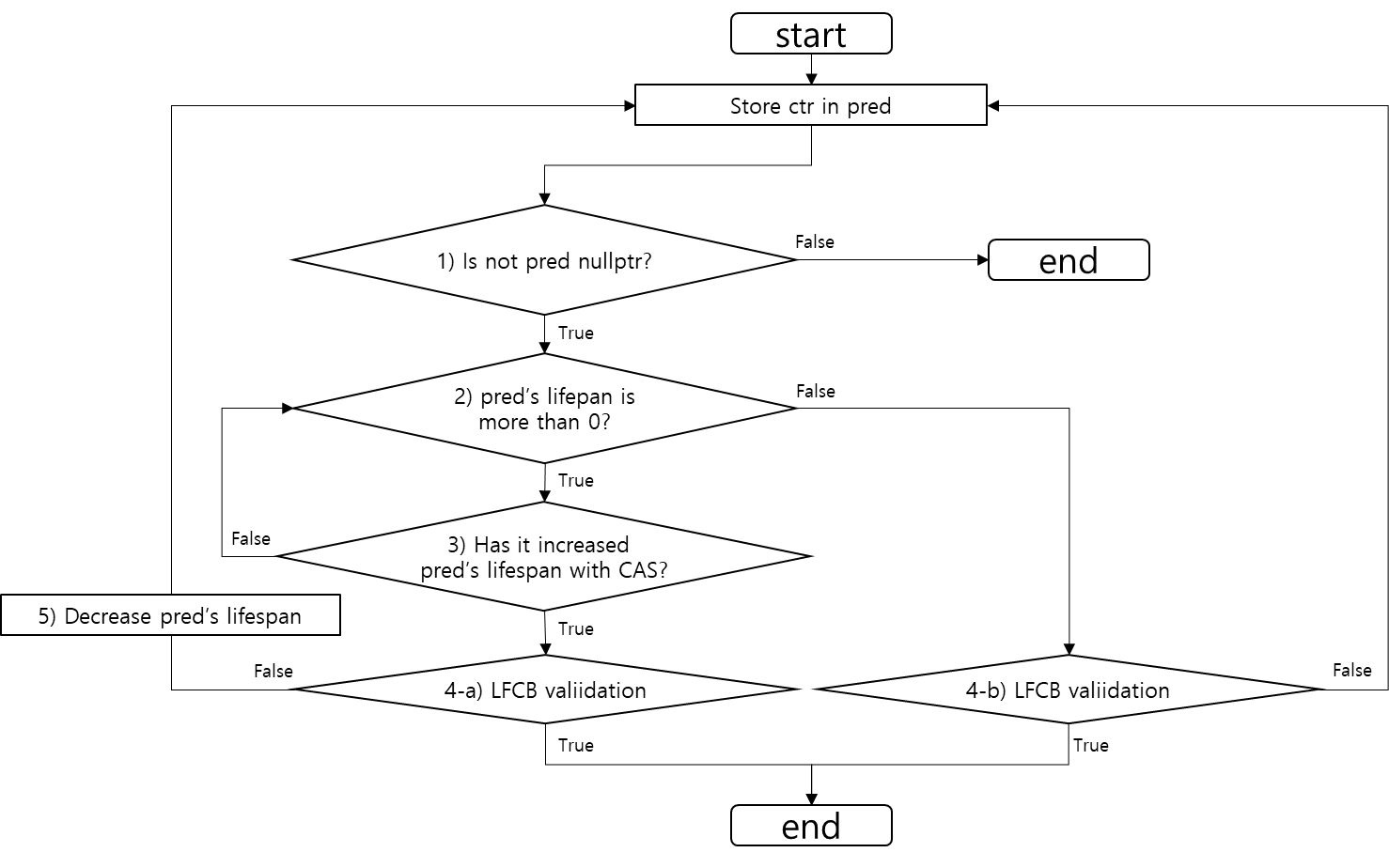
그림 1을 보면 LFSP/LFWP에는 유일하게 Lock-Free control\_block(LFCB)을 참조하는 포인터 ctr가 있으며, LFCB에는 객체를 참조하는 ptr 포인터, 참고문헌 **[5]**에서의 use count와 weak use count에 해당하는 use\_count와 weak\_count가 있음을 볼 수 있다. LFSP/LFWP는 공유하는 객체 접근에 참조하는 LFCB(ctr)의 ptr을 이용하기 때문에 ctr이 nullptr를 가리킨다면 객체를 공유하지 않음을 의미한다. 만약 LFCB를 해제(delete)한다면, 이는 LFSP의 생성과 함께 ABA문제를 발생시킬 수 있다. 그러므로 LFCB는 재사용되며, 그림 1에서 볼 수 있는 FreeList는 사용이 끝난 LFCB들을 수거하고 이를 LFSP 생성에서 재사용한다. 이러한 구조는 LFSP/LFWP의 ctr 포인터 수정만으로 데이터 레이스와 내부의 ABA문제 없이 공유하는 객체를 변경할 수 있음을 보여준다.

2.2 Lock-Free 알고리즘

LFSP/LFWP의 공유 객체 변경을 위한 Lock-Free 알고리즘은 크게 두가지로 나뉜다. 첫 번째는 참조할 LFCB의 use\_count와 weak\_count를 증가시키는 addcopy 알고리즘이고, 두 번째는 참조하는 LFCB를 변경하는 operator 알고리즘이다. LFSP와 LFWP의 addcopy 알고리즘과 operator 알고리즘은 사용하는 use\_count와 weak\_count의 변수 차이만 있으므로, 2.2.1 addcopy 알고리즘과 2.2.2 operator 알고리즘에서는 LFSP에 대해서만 논하였다.

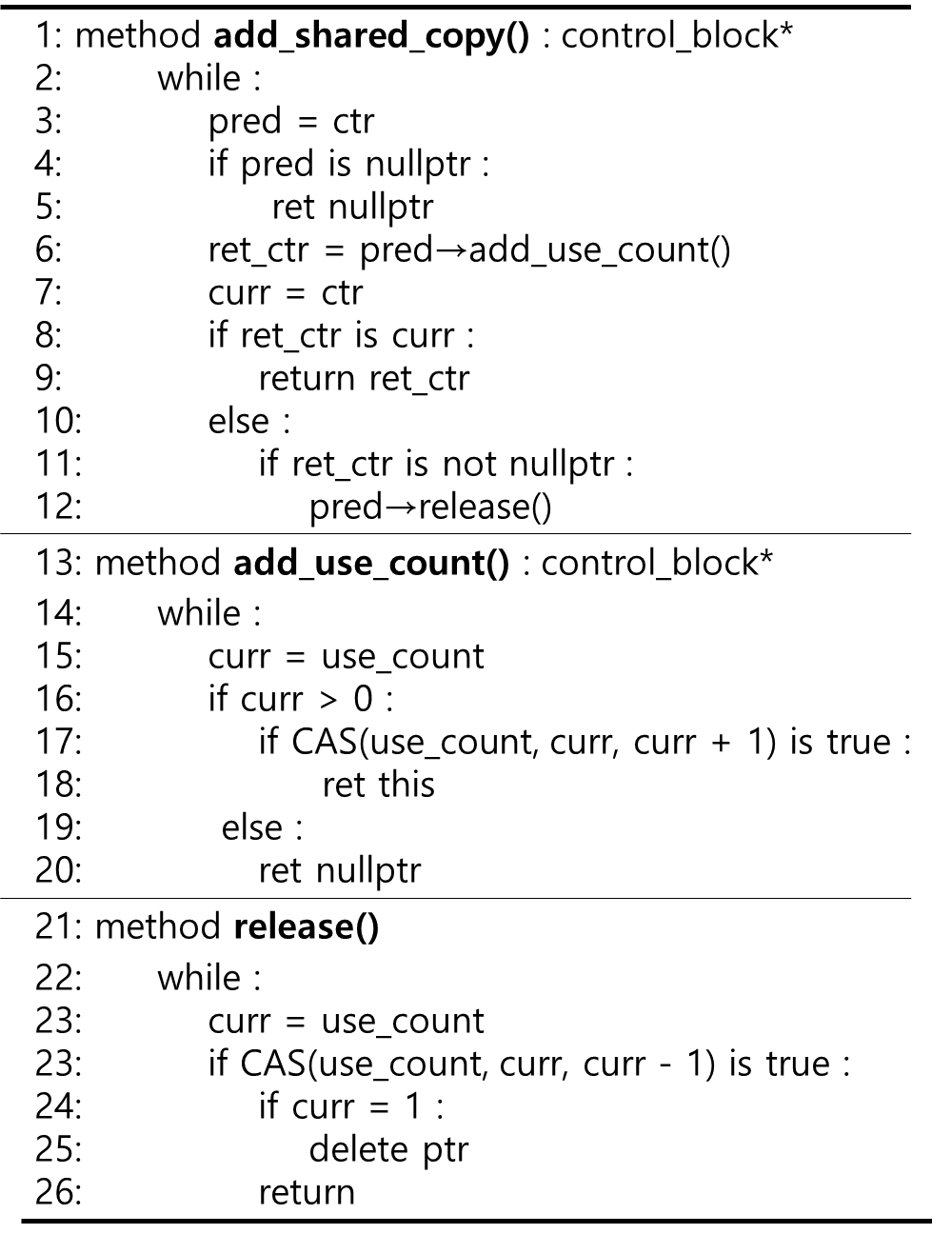
2.2.1 addcopy 알고리즘

addcopy 알고리즘은 LFSP가 참조하는 ctr의 use\_count를 증가시키는 알고리즘이다. 이는 use\_count를 미리 증가시켜 LFCB가 재사용되지 않음을 보장하며, 그동안 ctr이 다른 스레드의 영향을 받지 않았음을 보장하는 LFCB 유효성 검사(LFCB validation)를 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 안전하게 lifespan를 증가시킬 수 있다. 이해를 돕기 위해 lifespan을 use\_count(uc)라 하였다. 1)은 LFSP가 객체를 공유하는지 확인한다. 만약 ctr이 nullptr인 경우 알고리즘이 종료되며, 이 경우 알고리즘의 실패를 의미한다. LFSP가 객체를 공유하고 있다면, 2)와 3)의 순서와 같이 LFCB의 uc가 0보다 큰 값인지 검사한 뒤 CAS를 이용해 증가시켜야 한다. 이는 다른 스레드로부터 LFCB가 재사용되지 않음을 확인한 뒤 uc를 증가시켜, 1 이상의 uc를 가진 LFCB가 다른 스레드에 의해 재사용되지 않음을 보장할 수 있기 때문이다. 4) 이전의 과정 실행 중 ctr은 다른 스레드에 의해 수정될 수 있다. 따라서 4)에서 ctr이 3)의 LFCB를 참조하는지 확인하는 LFCB 유효성 검사를 실행한다. 5)는 4-a)에서 LFCB 유효성 검사가 실패한 경우 uc를 감소시켜 초기 상태로 돌아가는 과정이다. LFCB 유효성 검사의 자세한 내용은 2.2.1.1 LFCB 유효성 검사에서 다룬다.

다음은 addcopy 알고리즘을 이용한 대표적인 메소드인 LFSP::add\_shared\_copy()의 의사코드다.

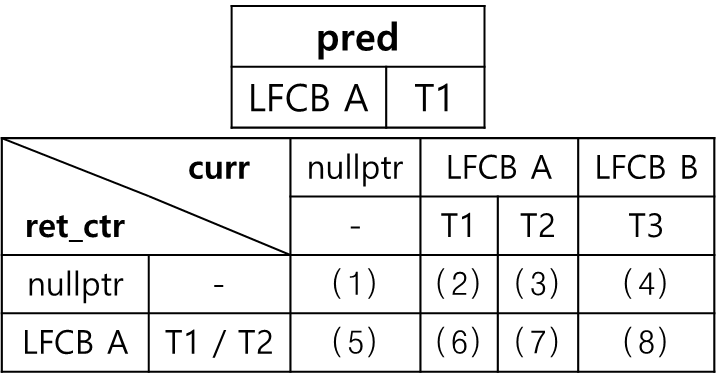
그림 3

add\_shared\_copy()는 uc를 증가시킨 경우 curr를 반환하며, 이때 curr은 nullptr가 아니다. 반대로 uc를 증가시키지 못한 경우 nullptr를 반환한다. 위 메소드에서 반환된 값이 nullptr가 아닌 경우 메소드가 성공했음을 의미한다. add\_shared\_copy()가 addcopy 알고리즘을 만족함을 보기위해 그림 2와 3을 함께 보자. 그림 2의 1)에 해당하는 line 4에서는 ctr이 LFCB를 참조하고 있지 않는 경우 nullptr를 반환하는 것을 볼 수 있다. line 6의 add\_use\_count()는 LFCB의 멤버 메소드이다. line 13~ 20을 살펴보면, add\_use\_count()는 uc가 0보다 큰 경우만 CAS를 이용해 uc를 1 증가시키고 LFCB를 참조하는 포인터를 반환하며, uc가 0 이하인 경우 nullptr을 반환하는 것을 볼 수 있다. 따라서 add\_use\_count()는 그림 2의 2)와 3)에 해당한다. line 6에서 ret\_ctr은 add\_shared\_copy()의 반환 값, line 7에서 curr은 add\_shared\_copy() 이후 ctr을 참조하는 지역 포인터이며, 두 포인터는 LFCB 유효성 검사에 이용된다. 그림 2 4)의 과정인 LFCB 유효성 검사는 line 8에서 실행된다. LFCB 유효성 검사는 ret\_ctr과 curr을 비교하며, 동일한 경우 curr을 반환한다. 여기서 ret\_ctr과 curr은 nullptr로 동일할 수 있다. 따라서 add\_shared\_copy()의 성공과 실패를 확인하기 위해서 반환된 curr을 확인해야 한다. 만약 LFCB 유효성 검사가 실패하면 release()를 호출한다. release()는 그림 2)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. LFCB가 유효하다는 것은 ret\_ctr과 curr이 동일한 경우로, LFSP가 참조하는 LFCB의 uc를 증가시키는 동안 LFSP의 ctr이 다른 스레드의 영향을 받지 않음을 보장한다. LFSP에 대한 그림 3 line 8에서의 LFCB 유효성 검사는 다음과 같다.

그림 3 line 3에서 pred는 LFCB A를 가리키며, T1 객체를 공유한다고 가정하자. line 6에서 ret\_ctr은 LFCB A와 nullptr를 가리킬 수 있으며, 각각 uc 증가의 성공과 실패를 의미한다. line 7에서 curr은 다른 스레드에 의해 ctr이 수정되지 않은 경우 LFCB A를 가리키지만, 여기서 LFCB A가 T1 객체를 공유하는 것은 알 수 없다. curr이 ctr의 값을 저장하기 전, 다른 스레드에 의해 LFCB A가 재사용되어 T2 객체를 공유한 뒤, ctr이 이를 가리킬 수 있기 때문이다. 따라서 curr은 T1과 T2를 공유하는 LFCB A를 가리킬 수 있으며, 다른 스레드에 의해 수정된 nullptr와 T3 객체를 공유하는 LFCB B를 가리킬 수 있다. 위의 모든 가정은 그림 4와 같이 표현할 수 있다.

 그림 4

설명을 위해 그림 2와 4를 함께 보자. 그림 4에서 ret\_ctr가 nullptr인 경우는 그림 2)의 False에 해당하며, ctr이 가리키는 LFCB가 재사용될 것임을 의미한다. 따라서 그림 2 4-b) LFCB 유효성 검사에서 LFSP가 공유하는 객체가 없는 상황 (1)은 LFCB가 유효하며, 이를 제외한 상황 (2), (3)과 (4)는 LFCB가 유효하지 않다. ret\_ctr가 LFCB A인 경우 그림 2 3)의 True에 해당하며, LFCB A의 uc가 증가했음을 의미한다. 그러므로 그림 2 4-a) LFCB 유효성 검사에서 curr이 LFCB A를 가리키는 상황 (6)은 LFCB가 유효하며, nullptr과 LFCB B를 가리키는 상황 (5)와 (8)은 LFCB가 유효하지 않다. 그렇다면 curr이 재사용된 LFCB A를 가리키는 상황 (7)에 대해 논해보자. 그림 2 1)이 성공했으므로 pred가 LFCB A를 가리키기 때문에 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 1)과 3) 사이에서 재사용 되었다고 가정해보자. LFCB A의 uc는 T2 객체를 공유한 이후 증가했다. 따라서 3)은 T2 객체에 대한 uc 증가를 의미하며, curr(LFCB A)은 T2 객체를 가리키고 있으므로 LFCB가 유효하다. 만약 LFCB A의 uc가 0으로 아직 재사용되지 않았다면, 3)과 2)가 순서대로 실패하므로 상황 (3)과 동일한 결과를 가진다.

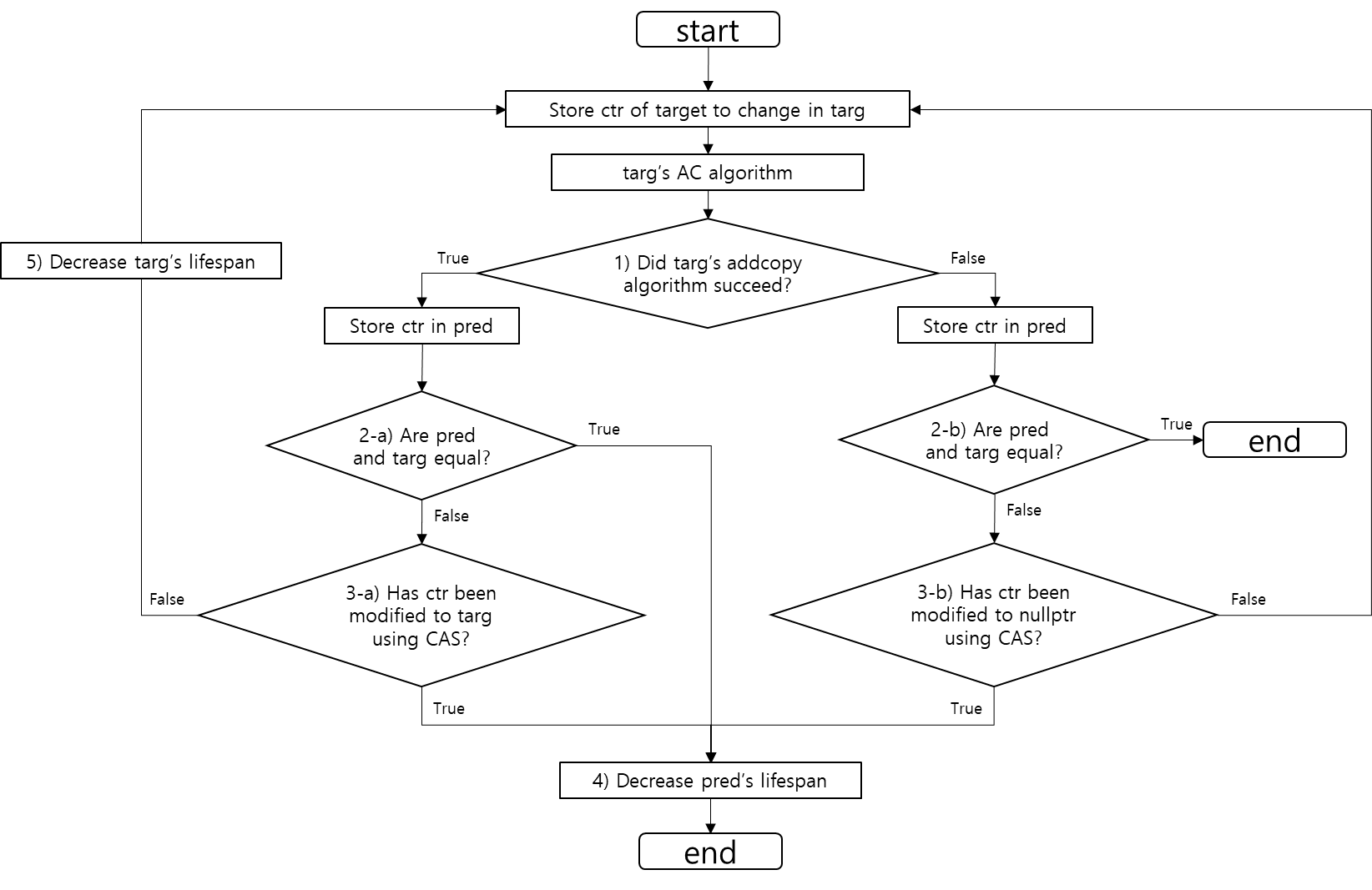
가정 2) LFCB A가 3)과 4) 사이에서 재사용 되었다고 가정해보자. 3)에서 T1 객체를 공유하는 LFCB A의 uc가 증가해 1 이상의 값을 가지므로 3) 이후에 LFCB A는 재사용될 수 없다. 따라서 3) 이후에 LFCB A가 재사용되었다는 가정 2)는 모순이다.

가정 1)과 가정 2)를 통해 상황 (7)에서 LFCB A는 1)과 3)사이에서 재사용되며, 3)은 T2 객체에 대한 uc의 증가임을 알 수 있다. 그러므로 T2 객체에 대한 LFCB A의 uc를 증가시킨 상황 (7)은 LFCB가 유효한 것을 알 수 있다.

따라서 LFCB 유효성 검사는 상황 (1), (6), (7)에서 성공하며, 상황 (1)은 addcopy 알고리즘의 실패를 의미하는 nullptr, 상황 (6)과 (7)은 성공을 의미하는 LFCB A를 반환한다. 이를 통해 LFCB 유효성 검사는 LFSP가 공유하는 객체의 uc를 증가시키는 addcopy 알고리즘의 정확성을 보장할 수 있다.

2.2.2 operator 알고리즘

operator 알고리즘은 LFSP가 참조하는 LFCB를 변경하는 알고리즘이다. operator 알고리즘은 CAS를 이용해 ctr을 수정하며, 이는 다른 스레드의 영향을 받지 않고 ctr을 수정했음을 보장한다. 다음 그림 5는 operator 알고리즘의 순서도를 보여준다.

 그림 5

operator 알고리즘은 위와 같이 addcopy 알고리즘 이후 그림 5의 1)부터 4)까지의 과정이 완료되야 안전하게 ctr을 수정했음을 보장할 수 있다. operator 알고리즘은 ctr을 수정하기 전 addcopy 알고리즘을 실행해 참조할 LFCB가 재사용되지 않음을 보장하며, 1)과 같이 addcopy 알고리즘의 결과에 따라 두가지 과정으로 나뉜다. addcopy 알고리즘이 성공한 경우 공유할 객체의 uc가 증가되었음을 의미하므로 LFSP가 해당 객체를 공유해야 하며, 실패한 경우 공유할 객체가 없음을 의미하므로 LFSP는 어떠한 객체도 공유하면 안된다. 2-a)와 2-b)는 LFSP의 ctr과 수정할 값이 동일한지 확인하는 알고리즘 최적화 과정이다. 2-a)의 True는 LFSP가 이미 객체를 공유하고 있음을 의미하며, 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.

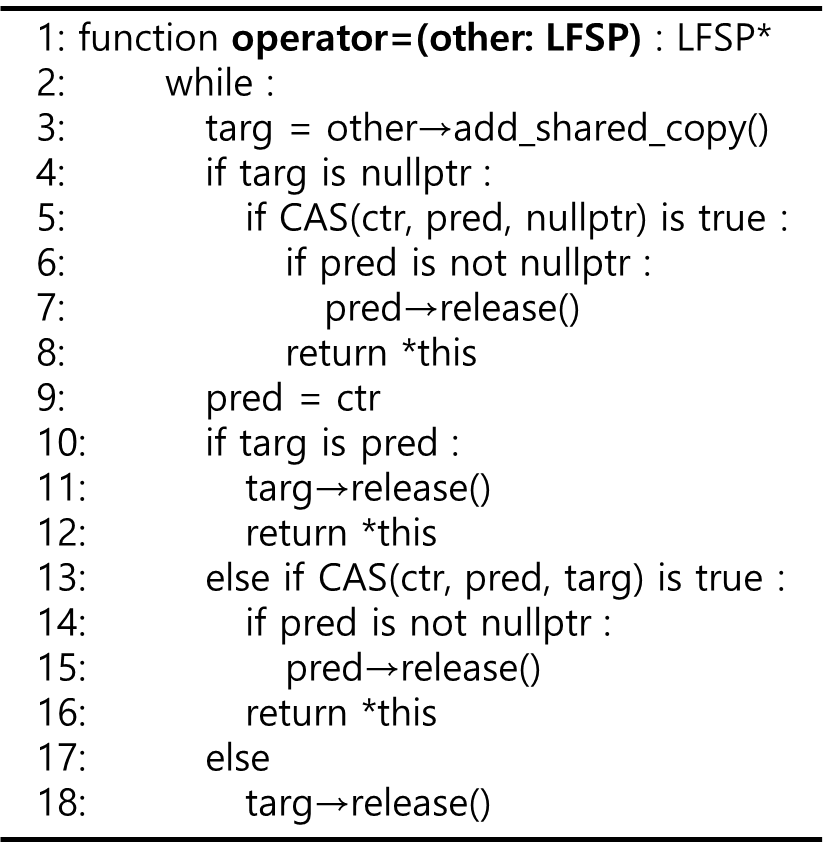
 그림 6

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

4 실험

5 결론

reference

**[1]** K. Fraser, "Practical Lock-Freedom," Ph.D. dissertation. King's College, University of Cambridge, pp. 33, pp. 79-81, 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]** James G. Peterson, Nisha Talagala, Swamin athan Sundararaman, Sriram Subramanian, “Epoch Based Storage Management For A Storage Device”, 2016, 05

**[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, pp990-995, 2013

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

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

[c] https://www.modernescpp.com/index.php/atomic-smart-pointers

교수님 확인용 ref

1. 연속할당 기법을 이용한 효과적인 Lock-Free 메모리 할당자 설계 및 구현(서론)