멀티스레드에서 사용 가능한 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 객체에 접근할 때 데이터 레이스가 발생하기 때문이며, 이를 해결하기 위해서는 반드시 std::atomic 템플릿의 atomic\_load()/atomic\_store()와 같은 API를 이용해 접근해야 한다. 하지만 이러한 방법은 뮤텍스 사용이 강제되어 멀티스레드 환경에서 큰 성능 저하를 일으킨다. 본 논문에서는 멀티스레드에서 원자적으로 동작하는 Lock-Free shared\_ptr와 weak\_ptr를 제안한다. 제안하는 두 객체는 싱글스레드에서 shared\_ptr를 사용하는 방식 그대로 사용할 수 있으므로 다른 메모리 관리 기법들 보다 낮은 사용난이도를 가진다. 그리고 std::atomic 템플릿을 이용한 shared\_ptr을 사용할 때보다 높은 성능을 보이는 것과 Lock-Free 알고리즘 동작하는 것을 실험을 통해 확인하였다. 이는 본 논문에서 제안한 메모리 관리 객체가 Lock-Free 알고리즘으로 구현된 고성능 C++ 소프트웨어들에 적합<<사용가능하다는 뜻인데 이거말고 사용하면 더 좋다라는 느낌이 필요>>하다는 것을 나타낸다.

키워드

abstract

keyword

1 서론

최근 멀티코어 프로세서의 발전으로 멀티스레드 프로그래밍의 필요성이 높아지면서 Lock-Free 알고리즘이 많은 분야에 사용되고 있다. 그 중 게임서버와 엔진 같은 높은 성능을 요구하는 프로그램은 low level 프로그래밍이 가능한 C++로 구현된다. 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 Lock-Free 알고리즘으로 구현된 C++ 프로그램은 멀티스레드에서 메모리 누수(memory leak)와 ABA 문제**[3]**가 야기된다.

C++11에서는 동적 메모리를 쉽게 관리할 수 있도록 std::shared\_ptr 템플릿을 제공하며, shaerd\_ptr는 사용이 끝난 동적 메모리를 스스로 해제한다. 이는 싱글스레드에서 정상적으로 동작하지만 멀티스레드에서는 스레드가 동기화되지 않아 데이터 레이스(Data Race)가 발생한다. 데이터 레이스는 여러 스레드가 동기화 없이 동일한 메모리에 접근할 때 하나 이상의 쓰기(write)연산이 있는 경우에 발생하기 때문이다[a]. 따라서 멀티스레드에서 shared\_ptr를 사용하기 위해서는 C++11 std::atomic 템플릿의 atomic\_load()/atomic\_store()를 이용해야 데이터 레이스가 발생하지 않는다[10]. 두 메소드는 전역 뮤텍스를 사용해 스레드를 동기화한다. 하지만 뮤텍스는 스레드의 수와 관련없이 한 개의 스레드만 공유 메모리에 접근하게 하므로 멀티스레드 프로그램의 성능을 저하시킨다[11]. 이러한 shared\_ptr의 문제점을 개선한 std::experimental::atomic\_shared\_ptr가 있다. 하지만 이는 C++20부터 제공되기 때문에 현재 상용화되어 있지 않다. 따라서 멀티스레드 프로그램에서 C++11 shared\_ptr를 사용해 동적 메모리를 관리하기 위해서는 std::atomic을 이용해야 하며, 이로 인한 성능 저하를 피할 수 없다.

본 논문에서는 멀티스레드에서 기존의 shared\_ptr보다 높은 성능을 가지는 Lock-Free shared\_ptr와 weak\_ptr를 제안한다. 두 객체는 C++11 shared\_ptr의 동작을 Lock-Free 알고리즘으로 실행하며, 멀티스레드에서 std::atomic 템플릿을 사용하지 않아도 데이터 레이스가 발생하지 않는다. 실험은 포인터 변수와 C++ shared\_ptr가 멀티스레드에서 사용될 때의 문제점을 보여주고 std::atomic을 이용한 C++ shared\_ptr와 Lock-Free shared\_ptr의 성능을 비교한다. 이를 통해 Lock-Free shared\_ptr와 weak\_ptr가 멀티스레드 프로그램에서 스레드의 수에 비례하게 성능이 향상되는 Lock-Free 알고리즘의 특징을 가지며, 기존의 shared\_ptr보다 높은 성능을 보이는 것을 확인하였다.

본 논문의 구성은 다음과 같다. 2장에서 Lock-Free shared\_ptr와 weak\_ptr의 구조와 동작에 대해 설명하였고 정확성을 논증하였으며, 3장에서 메모리 재사용을 위한 FreeList 객체에 대해 논하였다. 4장과 5장에서는 실험과 결론에 대해 기술하였다.

2 관련 연구

멀티스레드에서의 메모리 관리 문제를 해결하기 위해 Hazard Pointer**[4]**와 EBR(Epoch Based Reuse)**[5]** 등 여러 가지 메모리 관리 시스템이 제안되었다. 참고문헌 **[6]**은 효율적인 EBR인 DEBRA를 소개하며, Hazard pointer와 DEBRA를 이용한 코드를 보여준다. 두 코드를 살펴보면, Hazard pointer와 DEBRA는 적용 대상 알고리즘과 밀접하게 연동되어 있어 해당 알고리즘에 대한 이해가 낮으면 이를 적용하기 어려운 것을 알 수 있다. 이는 알고리즘에서 메모리 사용과 해제를 고려해 고유의 API를 적절한 위치에 추가해야 하는 높은 사용 난이도를 단점으로 가지는 것을 의미한다. 따라서 멀티스레드에서 알고리즘과 관련없이 메모리를 관리할 수 있는 객체가 필요한 것을 알 수 있다.

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

2.1 Lock-Free 알고리즘

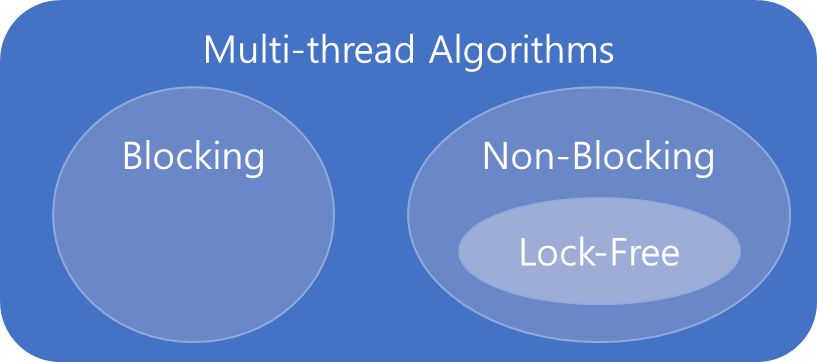
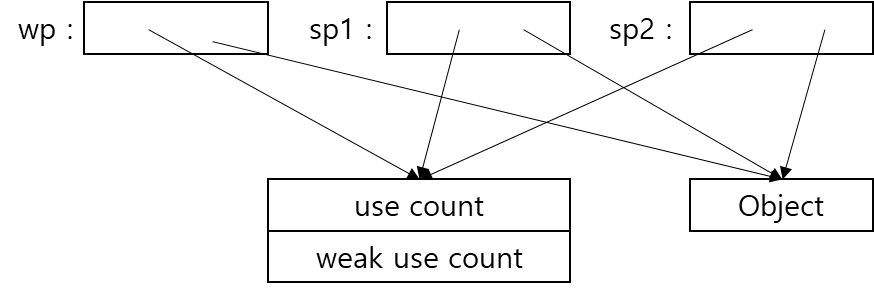
 그림 a

그림 a는 멀티스레드 알고리즘이 Blocking 알고리즘과 Non-Blocking 알고리즘으로 나뉘는 것을 보여준다. Blocking 알고리즘은 모든 스레드가 동일한 메모리에 접근하는 다른 스레드의 완료를 기다리는 알고리즘이며, Non-Blocking 알고리즘은 모든 스레드가 대기 없이 공유 메모리에 접근하고 다른 스레드에 의해 공유 메모리가 수정되면 수정된 값으로 재시도하는 알고리즘이다[2]. 이는 공유 메모리 접근에 대한 스레드의 경쟁이 발생하면 Blocking 알고리즘에서는 하나의 스레드만 실행되며, Non-Blocking 알고리즘에서는 모든 스레드가 실행되는 것을 의미한다. 따라서 Non-Blocking 알고리즘은 Blocking 알고리즘보다 병렬성이 높아 더 높은 성능을 가질 뿐만 아니라 스레드의 수에 비례하게 성능이 향상된다. Lock-Free 알고리즘은 Non-Blocking 알고리즘에 속하는 대표적인 알고리즘으로 여러 종류의 Non-Blocking 알고리즘 중 성능이 높고 프로그램 작성이 쉽다는 장점을 가진다. 이 알고리즘은 여러 스레드에서 동시에 실행될 때 적어도 하나의 스레드는 반드시 완료되는 알고리즘이며, 비교 및 교환을 의미하는 CAS(compare\_and\_swap) 명령어를 사용해 구현된다[2]. 본 논문에서는 C++로 Lock-Free 알고리즘을 구현하기 위해 C++11 <atomic>에서 제공하는 compare\_exchange\_strong()[8]를 이용해 CAS를 구현하였다. 이 메소드는 3가지 인자(메모리 포인터, 비교 값과 수정 값)를 통해 메모리에 기록된 값이 비교 값과 동일하다면 수정 값으로 수정하며, 반환된 true와 false를 통해 수정 성공 여부를 확인할 수 있다.

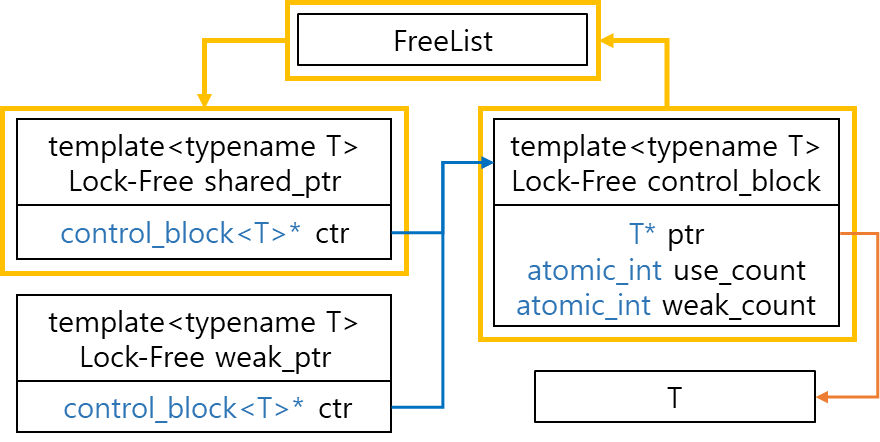
2.2 C++11 shared\_ptr과 weak\_ptr

 그림 b

C++11 shared\_ptr(SP)와 weak\_ptr(WP)는 스마트 포인터의 일종으로 사용 횟수가 0이 되면 원본 객체가 해제(delete)되는 계수 포인터이다. 본 논문에서는 원본 객체를 공유 객체라 하였고 shared\_ptr가 객체를 공유한다고 표현하였다. 그림 b는 두 개의 포인터를 갖는 SP/WP의 구조를 보여준다. 두 포인터는 각각 공유 객체와 control\_block를 참조하며, 여기서 control\_block은 사용 횟수와 관련된 use count와 weak use count를 가진 객체를 말한다. 이를 통해 SP/WP는 use count가 0인 경우 공유 객체를 해제하고 use count와 weak use count가 모두 0인 경우 control\_block을 해제해 동적 메모리를 관리한다. SP/WP가 다른 객체를 공유하려면 두 포인터는 공유할 객체와 공유할 객체를 가리키는 control\_block을 참조해야 하며, control\_block의 use count를 증가시켜야 하는 것을 참조문헌 [7]을 통해 볼 수 있다. 하지만 이 3가지 과정이 동시에 실행되지 않기 때문에 멀티스레드에서 데이터 레이스가 발생하며, 이를 해결하기 위해 std::atomic 템플릿을 이용해야 한다.

2.1 구조

SP/WP에서 발생하는 데이터 레이스를 해결하기 위해 Lock-Free shared\_ptr(LFSP)와 weak\_ptr(LFWP)는 그림 1과 같은 구조를 가진다.

 그림 1

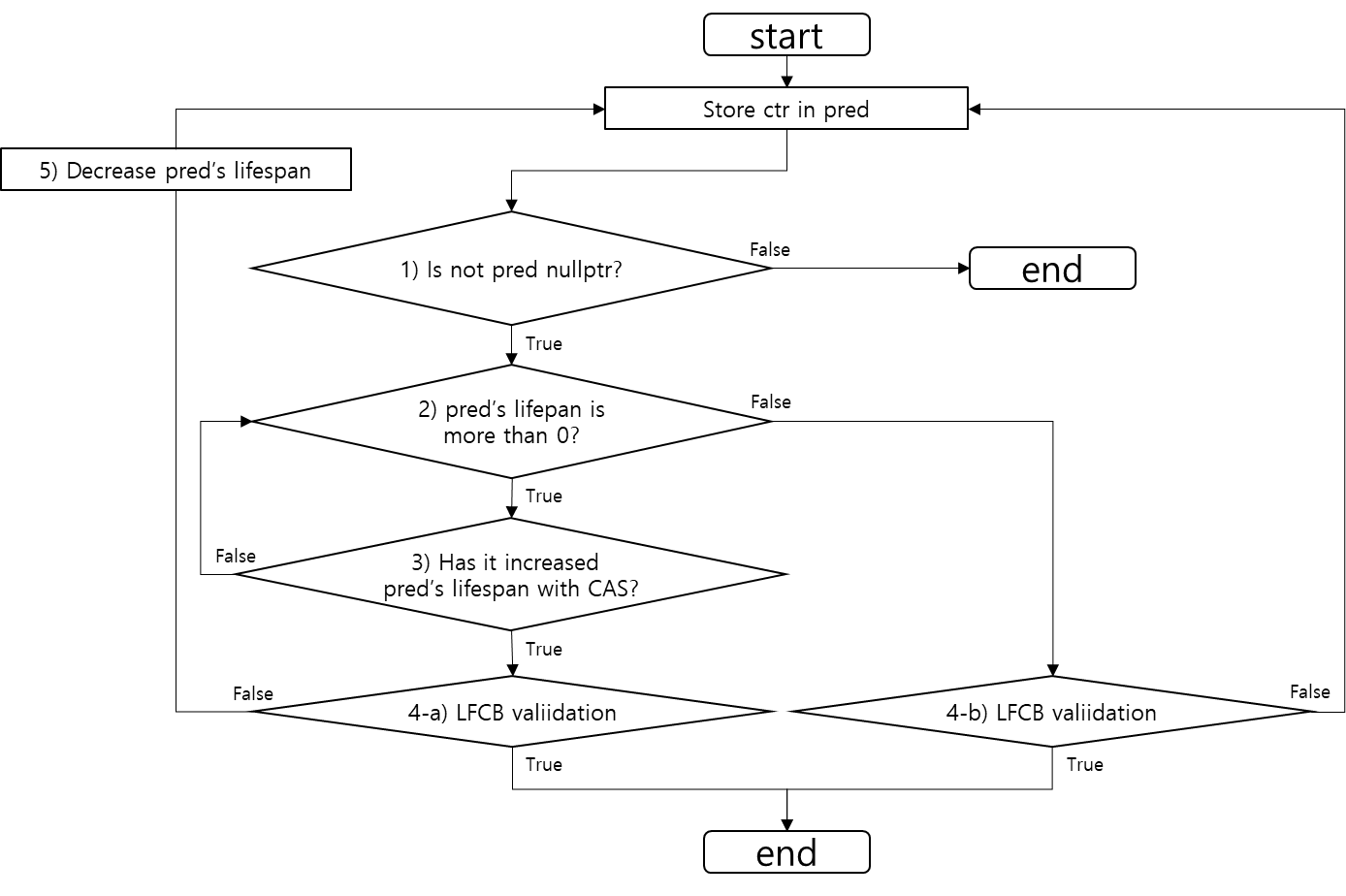
LFSP/LFWP는 SP/WP와는 달리 Lock-Free control\_block(LFCB)을 참조하는 ctr 포인터만을 가지며, LFCB는 공유 객체 T를 가리키는 ptr 포인터와 control\_block의 use count와 weak use count에 해당하는 use\_count와 weak\_count를 가진다. LFSP/LFWP는 사용이 끝난 LFCB(use\_count와 weak\_count가 0인 LFCB)를 해제(delete)해야 하지만, 다른 스레드에서 LFCB를 참조하고 있다면 해제된 메모리를 참조하게 되며 이로 인해 ABA 문제가 야기된다. 본 논문에서는 ABA문제를 방지하기 위해 사용이 끝난 LFCB를 재사용해 스레드가 사용 가능한 LFCB를 이용하게 하였다. 이를 위해 그림 1에서 볼 수 있는 FreeList 객체는 사용하지 않는 LFCB들을 수거하고 LFSP 생성에서 수거한 LFCB를 안전하게 재사용한다. 따라서 LFSP/LFWP가 공유 객체를 참조하지 않는 이러한 구조는 SP/WP가 다른 객체를 공유할 때 실행하는 3가지 과정이 2가지 과정으로 줄었음을 뜻하며, 이를 통해 LFSP/LFWP는 참조할 LFCB의 use\_count/weak\_count를 증가시키고 LFCB를 참조해 다른 객체를 공유한다.

2.2 구현

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

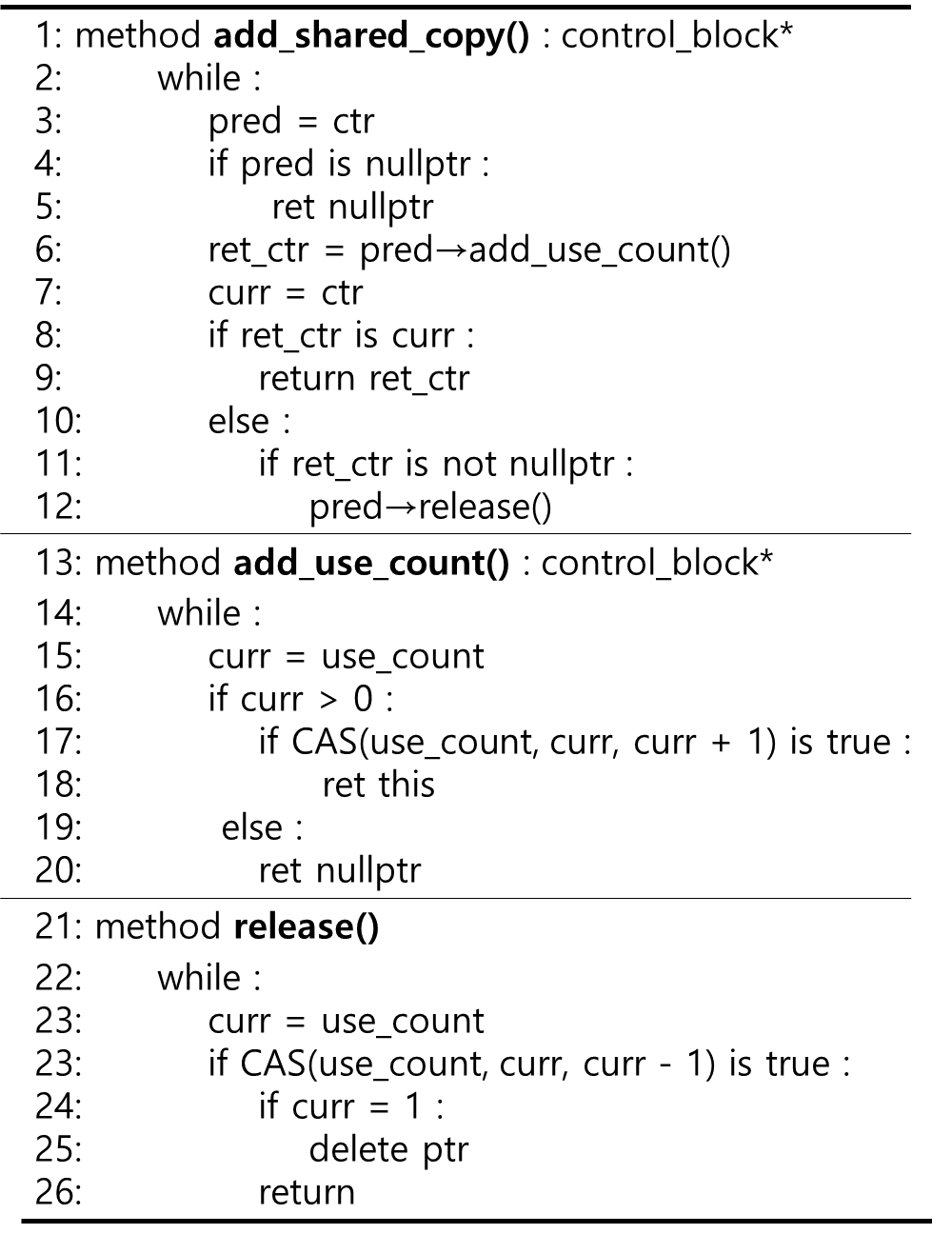
2.2.1 addcopy 알고리즘

addcopy 알고리즘은 공유 객체의 use\_count를 미리 증가시켜 다른 스레드에서 use\_count를 감소시키더라도 LFCB(ctr)를 사용할 수 있도록 보장한다. 알고리즘의 결과를 통해 use\_count 증가의 성공과 실패를 알 수 있으며, 실패 시 LFCB를 사용할 수 없으므로 이때는 공유하는 객체도 이용할 수 없다. 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 lifespan 증가를 보장할 수 있다. 이해를 돕기 위해 lifespan을 use\_count(uc)라 하였다. 그림 2 1)은 LFCB가 LFCB을 참조하는지 확인하기 위해 LFSP의 ctr을 확인하며, 만약 nullptr라면 LFCB를 참조하지 않음으로 알고리즘은 실패로 종료된다. LFSP가 LFCB를 참조하고 있다면 그림 2 2)와 3)의 순서와 같이 LFCB의 uc가 0보다 큰 값인지 검사한 뒤 CAS를 이용해 증가시켜야 한다. 이는 사용 가능한 LFCB의 uc를 증가시켜 다른 스레드에서 uc를 감소시키더라도 항상 1이상인 uc를 가지도록 보장할 수 있기 때문이다. 그림 2)와 3)의 과정 중 다른 스레드에 의해 LFSP가 다른 LFCB를 참조한다면 증가한 uc는 이전 LFCB의 uc임을 뜻한다. 이때 알고리즘은 LFSP가 참조하는 LFCB의 uc를 증가시켰다고 할 수 없다. 따라서 그림 2 4-a)에서는 증가시킨 uc의 LFCB(pred)를 여전히 LFSP가 참조하고 있는지 확인하는 LFCB 유효성을 검사한다. LFCB 유효성에 대한 자세한 내용은 2.2.1.1 LFCB 유효성 검사에서 다룬다. LFCB가 유효하다면 알고리즘의 성공으로 종료되지만, 반대의 경우라면 초기 상태로 돌아가 LFSP가 참조하는 LFCB로 재시도 해야 한다. 이때 LFSP는 초기 상태로 돌아가기 위해 그림 2 5)의 과정을 거치며, 여기서 이전 LFCB(pred)의 uc를 감소시킨다. 그림 2 2)에서 uc가 0인 경우 LFSP가 참조하는 LFCB는 사용할 수 없는 LFCB임을 의미한다. 그러므로 그림 2 4-b) LFCB 유효성 검사에서 LFSP가 LFCB를 참조하지 않는다면 LFCB가 유효함을 뜻하며 알고리즘의 실패로 종료된다. 반대로 그림 2 4-b) LFCB 유효성 검사에서 LFCB가 유효하지 않다면 LFSP가 다른 LFCB를 참조함을 뜻한다. 이때 그림 2 4-a)와 동일하게 초기 상태로 돌아가 LFSP가 공유하는 객체로 재시도하지만 그림 2 4-b)에서는 uc를 증가시키지 않았기 때문에 그림 5)의 과정은 생략한다. 위의 과정을 통해 addcopy 알고리즘은 LFSP가 LFCB를 참조하지 않는 경우 그림 2 1)에서 종료되고 LFCB가 유효한 경우 그림 2 4-a)와 4-b)에서 종료되며, 알고리즘의 결과를 통해 성공과 실패를 알 수 있다.

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

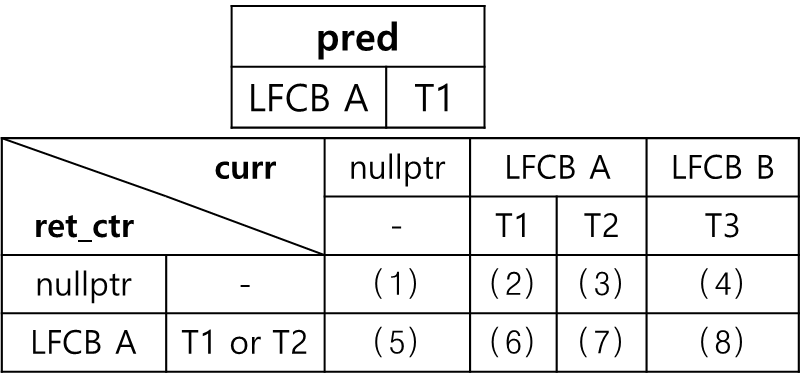
그림 3

add\_shared\_copy()는 uc를 증가시킨 경우 LFCB를 가리키는 curr를 반환하며, uc를 증가시키지 못한 경우 nullptr를 반환한다. 따라서 반환된 값이 nullptr인 경우 addcopy 알고리즘의 실패를 의미한다. add\_shared\_copy()의 이해를 돕기 위해 그림 2와 3을 함께 보자. 그림 2의 1)에 해당하는 line 4~ 5에서는 LFSP가 객체를 공유하지 않는다면 add\_shared\_copy()는 nullptr를 반환한다. 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 add\_use\_count()의 반환 값을 통해 uc 증가의 성공과 실패를 확인할 수 있으며, 이는 각각 그림 2의 3)의 True와 2)의 False를 의미한다. line 6에서 ret\_ctr add\_shared\_copy()의 반환 값을 참조하고 line 7에서 curr은 add\_shared\_copy() 이후 ctr과 동일한 LFCB를 참조하며, 이들은 line 8에서 LFCB 유효성 검사에 이용된다. LFCB 유효성 검사에서는 ret\_ctr과 curr을 비교하며, 동일한 경우 add\_shared\_copy()는 curr을 반환한다. 만약 LFCB 유효성 검사가 실패하면 release()를 호출한다. 그림 2 5)에 해당하는 release()는 LFCB의 멤버 메소드로, uc를 1 감소시키고 uc가 0인 경우 객체의 메모리를 해제한다(line 21~ 26).

2.2.1.1 LFCB 유효성 검사

이번 절에서는 그림 3에서의 LFCB 유효성 검사에서 발생할 수 있는 모든 상황을 가정하며, 이를 통해 LFSP가 참조하는 LFCB의 uc를 증가시키는 addcopy 알고리즘의 정확성을 논증한다. 그림 3 line 8에서 LFCB가 유효하다는 것은 ret\_ctr과 curr이 동일한 경우이며, 이때 LFCB 유효성 검사에서의 가정은 다음과 같다.

그림 3 line 3에서 pred는 LFCB A를 참조하며, T1 객체를 공유한다고 가정하자. 그림 3 line 6에서 ret\_ctr은 pred->add\_use\_count()가 성공한 경우 LFCB A, 실패한 경우 nullptr를 참조한다. line 7에서 curr은 다른 스레드에 의해 ctr이 수정되지 않은 경우 LFCB A를 참조하지만, 여기서 중요한 것은 LFCB A가 T1 객체를 가리키는 것을 알 수 없다는 점이다. 그림 3 line 3 이후 다른 스레드에 의해 LFCB A가 재사용되어 T2 객체를 가리킨 뒤 ctr이 LFCB A를 참조한다면, curr은 T2를 가리키는 LFCB A를 참조하기 때문이다. 따라서 그림 3 line 7에서 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 표현될 수 있다.

 그림 4

LFCB 유효성 검사의 논증을 위해 그림 2와 4를 함께 보자. 그림 2에서 LFCB 유효성 검사를 통해 LFCB가 유효하면 addcopy 알고리즘은 종료되며, 반대의 경우 초기 상태로 돌아가 알고리즘을 재시도한다. 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 LFCB A를 가리키는 상황 (6)과 (7)은 LFCB가 유효하며, nullptr와 LFCB B를 가리키는 상황 (5)와 (8)은 LFCB가 유효하지 않다. 이때 상황 (6)에서는 T1 객체에 대한 uc를 증가시켰음을 알 수 있지만, 상황 (7)에서는 증가한 uc가 T1 객체와 T2 객체 중 어떤 객체의 uc인지 알 수 없다. 이를 확인하기 위해 상황 (7)에 대해 가정해보자. 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 재사용되어 T2를 가리키는 시점에 대한 가정이다.

가정 1] LFCB A가 그림 2 1)과 3) 사이에서 재사용 되었다고 가정해보자. LFCB A는 T2 객체를 가리킨 이후 3)에서 uc가 증가했다. 따라서 4)에서 ret\_ctr(LFCB A)는 T2 객체를 가리키며, curr(LFCB A)도 T2 객체를 가리키고 있으므로 LFCB가 유효하다. 만약 그림 2 1)과 3) 사이에서 LFCB A의 uc가 0이 되었다면, 그림 2 2)와 3)이 실패하므로 상황 (3)과 동일한 결과를 가진다.

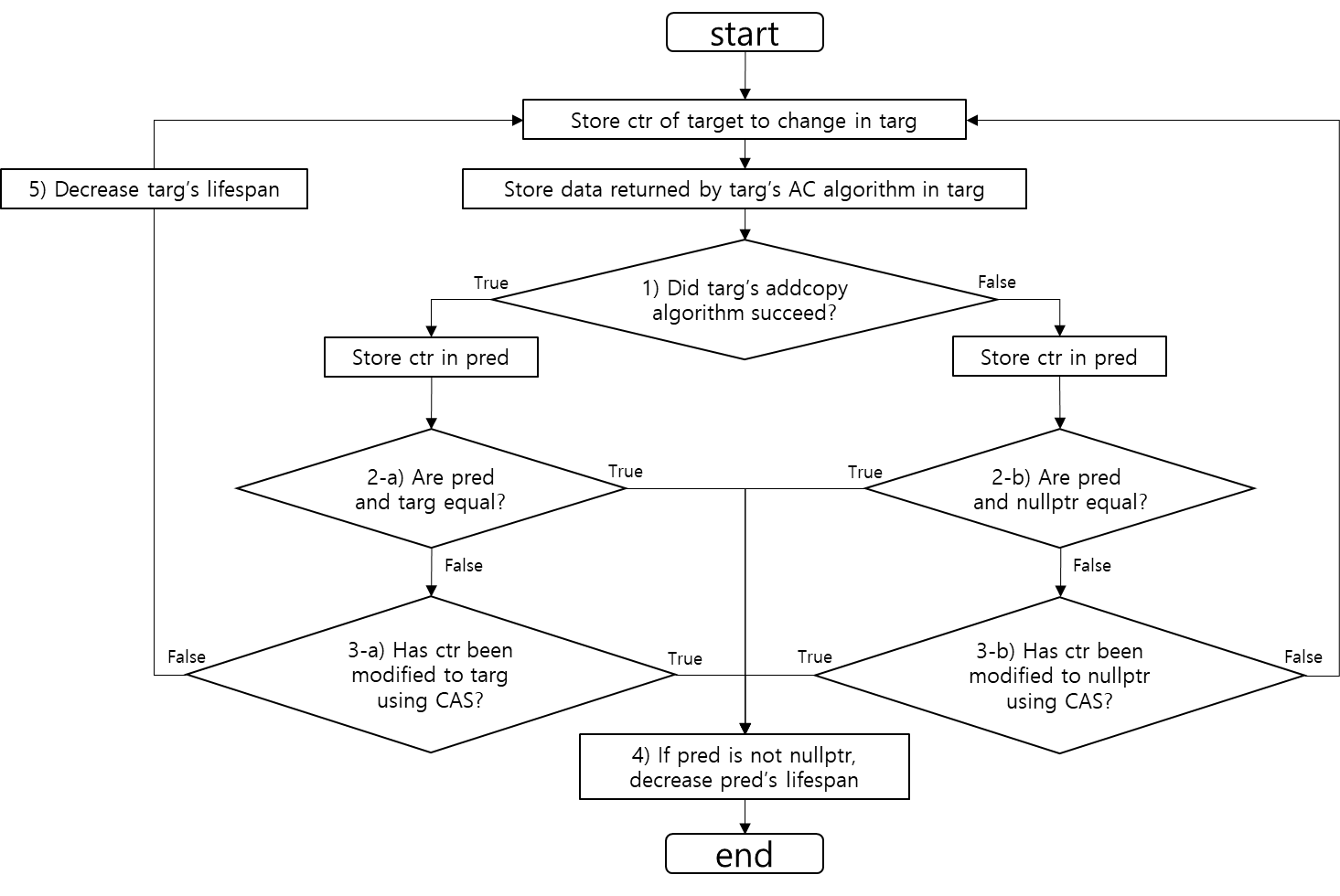
가정 2] LFCB A가 그림 2 3)과 4) 사이에서 재사용 되었다고 가정해보자. 3)에서 T1 객체를 가리키는 LFCB A의 uc가 증가했다. 이때 uc는 1 이상의 값을 가지므로 3) 이후에 LFCB A는 재사용될 수 없다. 따라서 LFCB A는 그림 2 3)과 4) 사이에서 재사용될 수 없다.

가정 1]과 가정 2]를 통해 상황 (7)에서 LFCB A는 1)과 3)사이에서 재사용되며, 3)에서 증가한 uc는 T2 객체의 uc임을 확인하였다. 그러므로 LFCB A가 재사용 되어 T2 객체를 가리키는 상황 (7)도 LFCB가 유효하다는 것을 알 수 있다.

LFCB 유효성 검사는 상황 (1), (6)과 (7)에서 성공하며, 상황 (1)은 nullptr, 상황 (6)과 (7)은 LFCB A를 반환한다. 이때 반환되는 nullptr과 LFCB A는 addcopy 알고리즘의 성공과 실패를 의미한다. 이를 통해 LFCB 유효성 검사로 addcopy 알고리즘의 정확성을 보장할 수 있음을 알 수 있다.

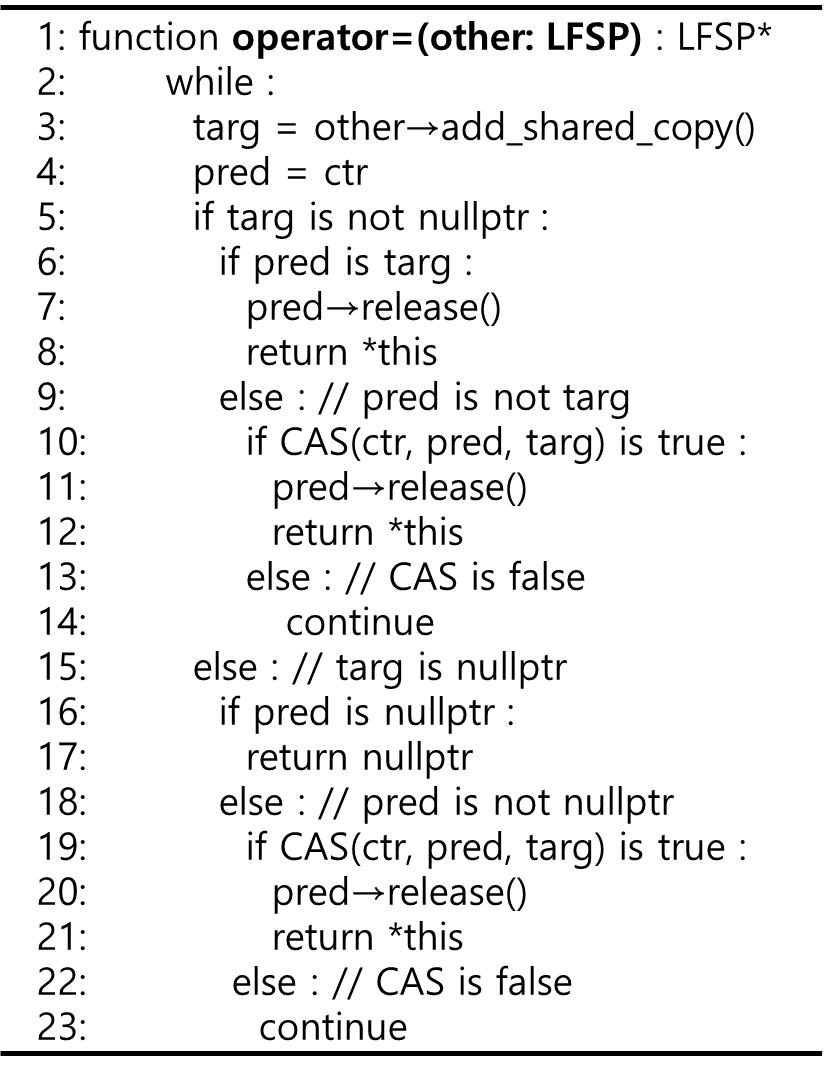
2.2.2 operator 알고리즘

operator 알고리즘은 LFSP가 공유하는 객체를 변경하기 위해 ctr을 수정하는 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 있으며, 여기서 Target은 변경할 객체를 가진 LFSP를 뜻하며, targ는 변경할 LFCB를 뜻한다. 이해를 돕기 위해 lifespan을 use\_count(uc)라 하였다. operator 알고리즘은 targ의 addcopy 알고리즘을 실행해 targ가 재사용되지 않도록 보장한 뒤, addcopy 알고리즘의 결과에 따라 그림 5 1)에서 두가지 과정으로 나뉜다. addcopy 알고리즘의 성공과 실패는 targ의 uc 증가와 targ가 가리키는 객체가 없음을 의미하므로 ctr은 targ와 nullptr를 참조해야 한다. ctr을 수정하기 전, ctr이 targ나 nullptr을 이미 참조하는지 확인하는 그림 5 2-a)와 2-b)의 과정을 거친다. 이는 ctr을 수정하는 과정을 생략하여 CAS 실패로 인한 알고리즘의 반복을 피할 수 있기 때문이다. ctr이 targ와 다르다면 그림 5 3-a)와 3-b)에서 CAS를 이용해 targ와 nullptr로 수정된다. 만약 CAS가 성공하면 LFSP의 공유 객체가 변경되었음을 의미하므로 LFSP가 이전에 공유한 객체의 수명을 관리해야 한다. 따라서 그림 5 4)에서 LFSP가 이전에 객체를 공유했는지 확인(pred==nullptr)한 뒤, 이전에 공유한 객체(pred)의 uc를 감소시킨다. 반대로 CAS가 실패하면 다른 스레드에 의해 LFSP의 공유 객체가 변경되었음을 의미한다. 따라서 초기 상태로 돌아가 변경된 ctr을 이용해 알고리즘을 다시 실행해야 하며, 그림 5 3-a)는 addcopy 알고리즘이 성공했기 때문에 증가된 targ의 uc를 그림 5 5)의 과정에서 감소시킨다.

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

 그림 6

3 FreeList

본 논문에서는 LFCB 해제로 인한 ABA 문제를 해결하기 위해 LFCB를 재사용하였다. LFCB 재사용을 관리하는 FreeList는 사용이 끝난 LFCB를 수거하고 LFSP 생성에서 재사용하는 Lock-Free 연결 리스트이며, 리스트를 구성하는 노드(Node)를 삭제하지 않아 내부적으로도 ABA 문제가 발생하지 않는다. 더하여 이는 LFSP와 LFWP를 구현한 네임스페이스(namespace)에 구현되어 LFCB와 같이 LFSP와 LFWP의 내부에서만 이용되므로 사용자가 이를 직접 이용할 필요가 없다.

FreeList를 구성하는 노드의 변수는 3가지로 리스트를 연결하는 next 포인터, 재사용할 LFCB를 참조하는 ctr 포인터와 노드의 상태를 나타내는 active가 있다. active는 0, 1과 2의 값을 가질 수 있으며, 각각 재사용 가능(활성), 재사용 불가(비활성)와 수정을 의미한다. FreeList의 메소드는 Alloc()과 Regist()가 있으며, 각각 FreeList에 등록된 재사용 가능한 LFCB를 반환하고 재사용할 LFCB를 FreeList에 등록한다.

다음 그림 7은 노드의 구조와 FreeList의 예시를 보여준다.

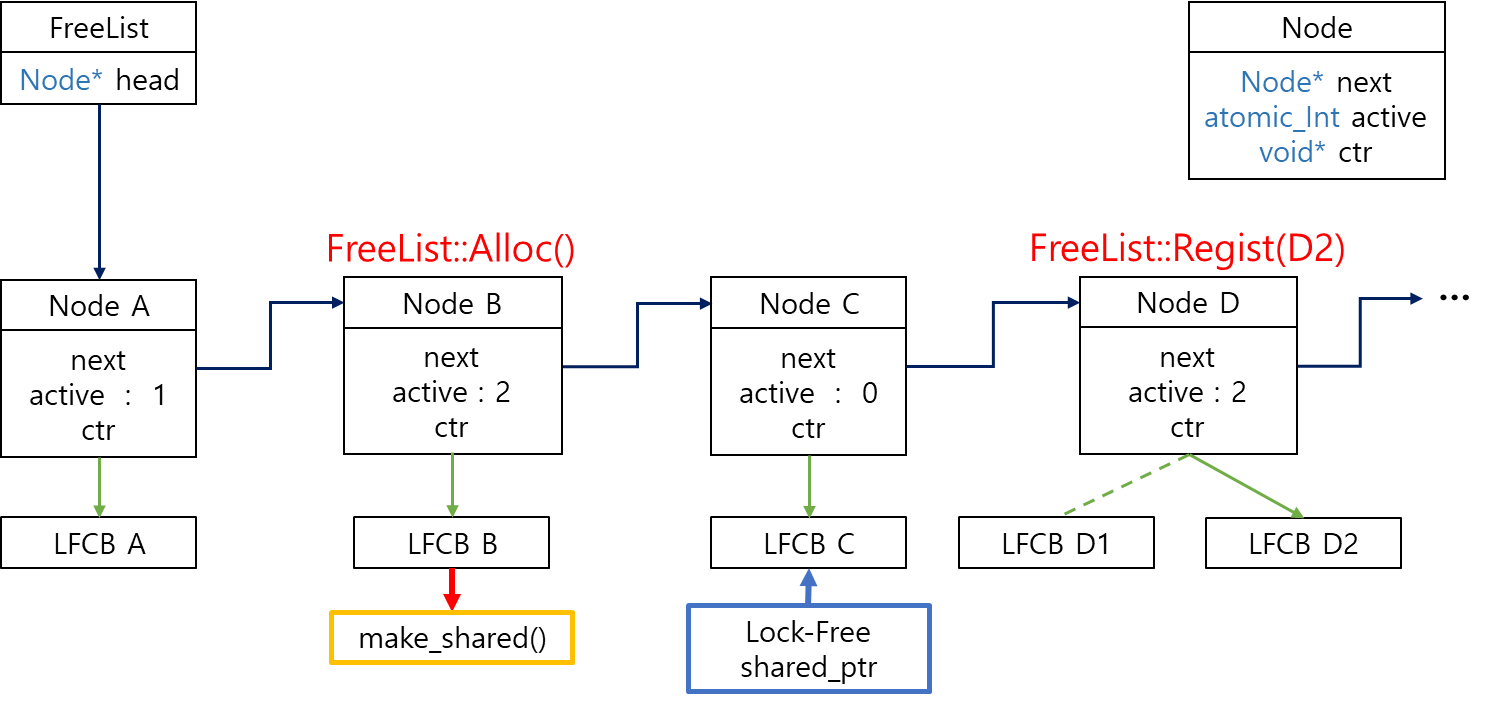
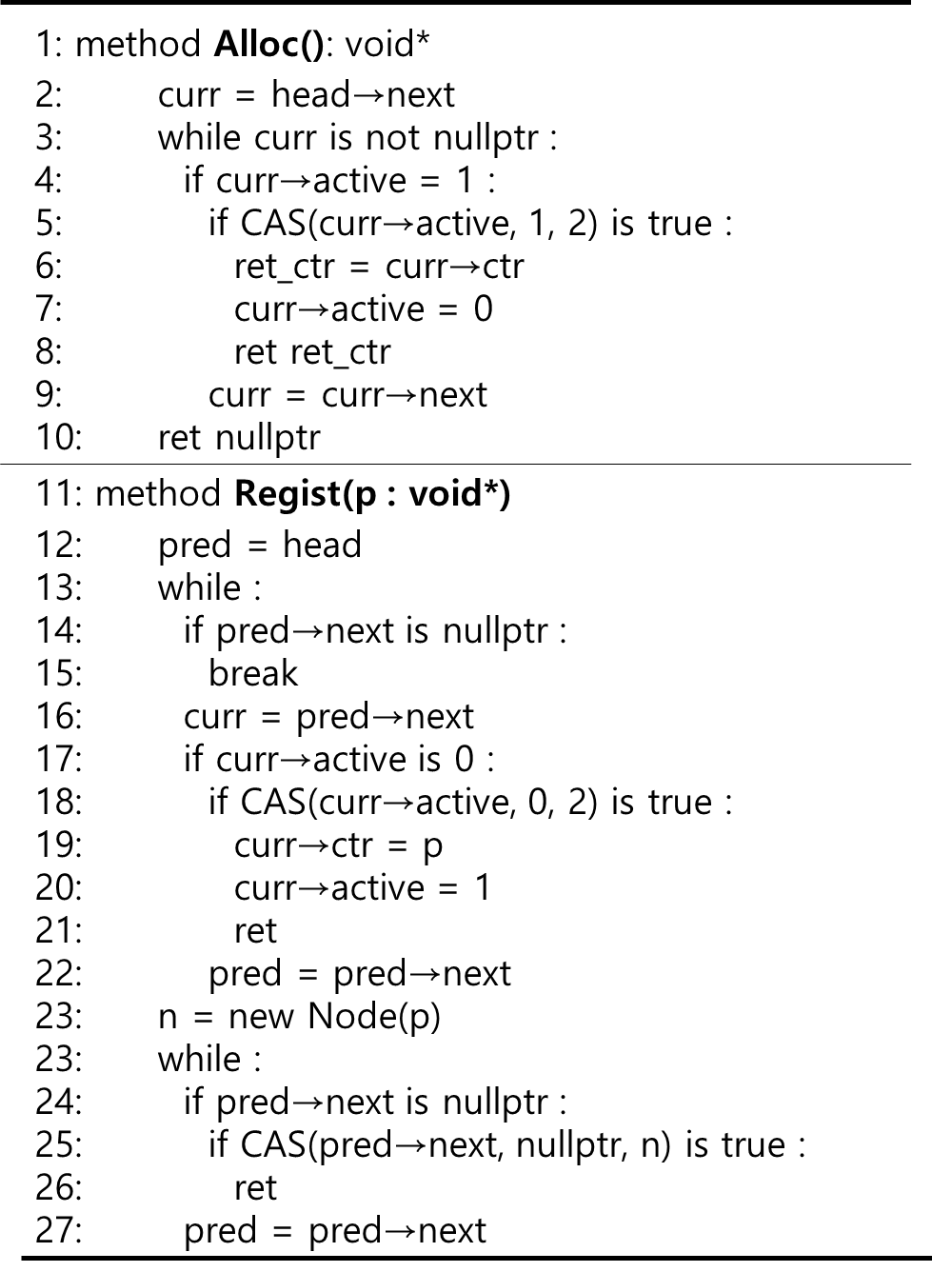
 그림 7

그림 7은 재사용 가능한 LFCB A를 참조하는 Node A(활성 노드), 재사용 불가한 LFCB C를 참조하는 Node C(비활성 노드)와 Alloc()과 Regist()에서 사용중인 Node B와 Node D(수정 노드)를 보여준다. 여기서 활성 노드와 비활성 노드는 Alloc()과 Regist()에서 CAS를 이용해 수정 노드로 변경되므로 수정 노드는 CAS가 성공한 한 개의 스레드에서만 이용될 수 있다. 다음은 FreeList의 메소드 Alloc()과 Regist()이다.

Alloc()은 FreeList에 등록된 재사용 가능한 LFCB를 반환하며, 재사용 가능한 LFCB가 없는 경우 nullptr을 반환한다. 재사용 가능한 LFCB를 검색하기 위해 Alloc()은 연결리스트를 순회하며 활성 노드를 검색한 후, 다른 스레드에서 해당 노드를 이용할 수 없도록 CAS를 이용해 수정 노드로 변경한다. CAS가 성공하면 노드를 재사용하기 위해 비활성 노드로 변경한 뒤 ctr을 반환해야 하지만 비활성 노드로 변경된 이후 다른 스레드에 의해 ctr이 수정된다면 이전의 ctr은 메모리 누수의 원인이 된다. 따라서 지역변수 포인터에 노드의 ctr을 기록하고 노드를 비활성 노드로 변경한 뒤 포인터를 반환해야 한다. 여기서 수정 노드는 다른 스레드가 이용할 수 없기 때문에 비활성 노드로 변경하는 과정은 CAS를 이용하지 않아도 된다.

Regist()는 Alloc()과 동일한 방법으로 비활성 노드를 검색한 후 CAS를 이용해 수정 노드로 변경하며, LFCB를 등록한 뒤 활성 노드로 변경한다. 이때 재사용할 LFCB를 비활성 노드에 등록하지 못하면 메모리 누수가 발생한다. 그러므로 비활성 노드를 찾지 못했다면 새로운 노드를 생성해 LFCB를 등록한 뒤 연결리스트의 말단에 삽입해야 한다. 연결리스트의 말단에 노드를 삽입하는 작업은 다른 스레드와의 경쟁을 야기한다. 따라서 CAS를 이용해 말단 노드의 next를 새로운 노드로 수정해야 한다. 만약 CAS가 실패하면 다른 스레드가 다른 새로운 노드를 삽입했음을 의미하므로 말단 노드를 계속 검색한 뒤 삽입한다.

이를 정리한 Alloc()과 Regist()의 의사코드는 그림 8과 같다.

그림 8

4 실험

실험은 참조문헌 **[12]** 게으른 동기화 연결 리스트(Lazy Synchronization Linked List: ZSL)에서 노드의 next 자료형으로 여러 종류의 포인터를 사용했을 때의 성능을 측정하고 비교한다. 여기서 사용되는 포인터는 포인터 변수, C++11 shared\_ptr, std::atomic 템플릿을 이용한 C++11 shared\_ptr와 본 논문에서 제시한 Lock-Free shared\_ptr가 있으며, 이를 사용해 PZSL, SPZSL, ATSPZSL과 LFSPZSL을 구현하였다. 다음은 ZSL의 성능 측정 방법이다.

1. 0에서 10000(key\_range)까지의 무작위 값을 선택한 후 노드 삽입/삭제/검색 중 하나의 메소드를 실행한다. 이때 각 메소드의 실행 확률은 모두 1/3이다.
2. 1)의 동작을 12만(num\_func)번 실행한다. 이때 각 스레드는 num\_func를 전체 스레드 수로 나누어 동작을 반복하기 때문에 num\_func는 측정할 스레드 수의 최소 공배수로 정한다.
3. 위의 과정을 스레드를 늘려가며 실행하고, 각 ZSL의 실행 소요 시간을 측정한다.

여기서 PZSL의 노드 삭제 메소드가 노드를 해제(delete)하면 노드의 next가 댕글링 포인터(Dangling pointer)가 되어 런타임 에러가 발생하므로 실험에서는 노드를 해제하지 않았다.

실험은 CPU intel® core™ i7-8700K 3.7GHz와 Visual Studio 2017(Release 모드, Win 32) 환경에서 실행하였다. 다음은 실험에서 측정한 각 ZSL의 실행 소요 시간을 이용해 밀리세컨드초당 메소드 실행 횟수(opms)를 계산한 결과를 보여준다.

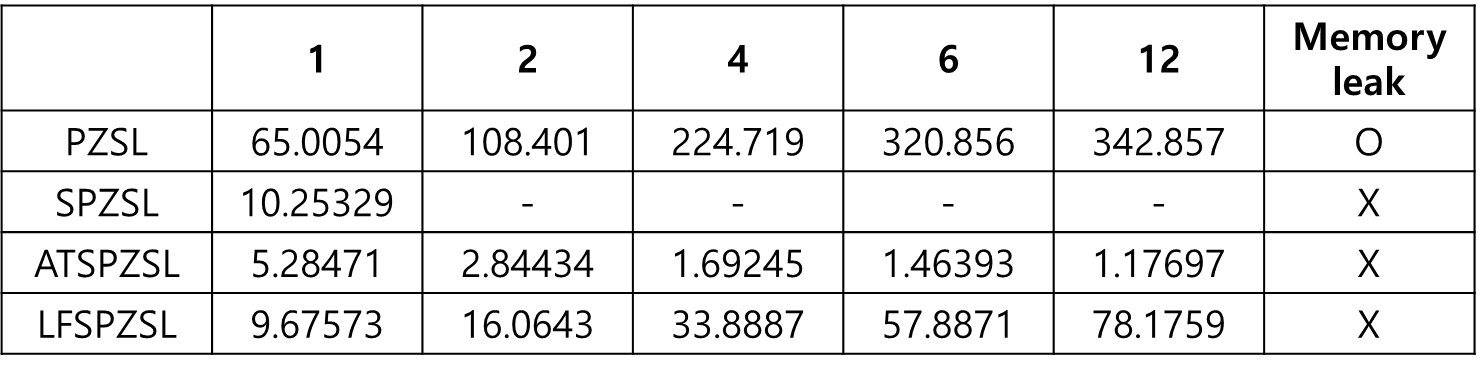
 그림 9

그림 9를 통해 PZSL이 동일한 스레드 수에서 실행될 때의 가장 높은 opms를 보이는 것을 볼 수 있다. 하지만 PZSL은 실제로 노드를 해제하지 않아 메모리 누수가 발생하므로 멀티스레드에서 사용할 수 없는 문제가 있다. 또, 스레드가 1개일 때 PZSL을 제외한 가장 높은 opms는 SPZSL임을 볼 수 있다. SPZSL은 std::atomic 템플릿을 이용하지 않아 다른 포인터보다 높은 성능 opms를 가진다. 하지만 이로 인해 데이터 레이스가 발생하기 때문에 멀티스레드에서 런타임 에러를 발생시킨다. 따라서 PZSL과 SPZSL는 멀티스레드에 적합하지 않은 것을 알 수 있다. ATSPZSL은 전역 뮤텍스를 사용해 shared\_ptr에 접근하며, 이는 shared\_ptr에 접근하는 모든 쓰레드와의 경쟁을 의미한다. 따라서 한 개의 뮤텍스에 대한 여러 스레드의 경쟁은 스레드가 많아질수록 경쟁상대가 많아져 성능 저하의 원인이 된다. 이는 그림 9에서 스레드가 많아질수록 opms가 감소하는 ATSPZSL를 통해 알 수 있다. Lock-Free shared\_ptr는 뮤텍스를 사용하지 않고 CAS를 이용해 동작한다. 이는 다른 스레드와 경쟁하지 않음을 의미하며, 다른 스레드의 영향을 받는 경우 알고리즘을 다시 시도한다. 따라서 이를 이용한 LFSPZSL은 멀티스레드에서 병렬성을 가질 수 있으며, 그림 9를 통해 스레드가 증가할수록 LFSPZSL의 opms가 증가하는 것을 볼 수 있다.

스레드의 수에 대한 LFSPZSL의 opms 향상 비율은 그림 10과 같다.

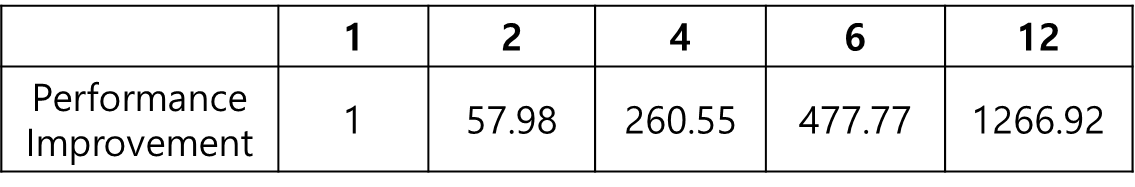
 그림 10

그림 10을 통해 LFSPZSL의 opms는 스레드가 1개인 경우보다 2개인 경우 57.98%, 4개인 경우 260.55% 향상된 것을 볼 수 있으며, 이 수치는 스레드가 많아질수록 스레드 수와 비례하게 증가한다.

이를 통해 본 논문에서 제시하는 Lock-Free shared\_ptr은 멀티스레드에서 std::atomic을 이용한 C++11 shared\_ptr보다 높은 성능을 보이며, 스레드 개수와 비례하게 성능이 향상되는 Lock-Free 알고리즘을 만족함을 볼 수 있다.

5 결론

본 논문은 멀티스레드에서 정상적으로 동작하는 Lock-Free shared\_ptr을 제안한다. Lock-Free shared\_ptr는 원자적으로 실행되어 멀티스레드에서 정상적으로 동작하며, LFCB 유효성 검사의 가정을 통해 공유하는 객체의 수명을 증가시키는 것을 논증하였다. 그리고 실험을 통해 Lock-Free shared\_ptr의 성능이 std::atomic 템플릿을 이용한 C++11 shared\_ptr의 성능보다 높으며, 스레드 수에 비례하게 성능이 향상되는 것을 통해 Lock-Free 알고리즘을 만족한다는 것을 보여준다. 이를 통해 Lock-Free shared\_ptr는 Lock-Free 알고리즘의 자료구조를 구현할 때 활용될 수 있으며, Lock-Free 알고리즘으로 구현된 고성능 C++ 소프트웨어들에 도움을 줄 것이라 기대한다.

reference

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

**[2]** Rainer Grimm, “Blocking and Non-Blocking Algorithms”, https://www.modernescpp.com/index.php/blocking-and-non-blocking,

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

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

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

**[6]** 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

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

[8] https://docs.microsoft.com/ko-kr/cpp/standard-library/atomic-functions?view=vs-2019#atomic\_compare\_exchange\_strong

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

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

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

**[12]** Maurice Herlihy, Nir Shavit, “The Art of Multiprocessor Programming”, Morgan Kaufmann Publishers, p208-213, 2008. 3

**[a] https://software.intel.com/content/www/us/en/develop/documentation/inspector-user-guide-linux/top/problem-type-reference/data-race.html**

**[b]** [Anders Gidenstam](https://www.semanticscholar.org/author/Anders-Gidenstam/2816022), [M. Papatriantafilou](https://www.semanticscholar.org/author/M.-Papatriantafilou/1752071), [H. Sundell](https://www.semanticscholar.org/author/H.-Sundell/2013776), [P. Tsigas](https://www.semanticscholar.org/author/P.-Tsigas/1701362) , Practical and Efficient Lock-Free Garbage Collection Based on Reference Counting

[c] [R Meyer](https://scholar.google.com/citations?user=zRo3ZJ4AAAAJ&hl=ko&oi=sra), S Wolff “[Pointer life cycle types for lock-free data structures with memory reclamation](https://dl.acm.org/doi/abs/10.1145/3371136)”

- Proceedings of the ACM on Programming Languages, 2019