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 Lock-Free 알고리즘을 이용한 C++로 구현된 프로그램은 메모리 누수(memory leak)와 ABA문제[1]가 야기된다. 이러한 문제를 해결하기 위해 Hazard Pointer[a]와 EBR(Epoch Based Reuse)[b] 등 여러 가지 메모리 관리 시스템이 제안되었다. 하지만 이 시스템들은 알고리즘에서 메모리 사용 패턴을 고려해 고유의 API를 적절한 위치에 추가해야 하는 높은 사용 난이도를 단점으로 가지고 있다.[ref: 있으면 좋고, 없으면 보충설명 : HP나 EBR을 사용한 A, B, C알고리즘들이 있는데, 살펴보면 HP나 EBR이 알고리즘과 밀접하게 연동되어 있어서, 알고리즘에 대한 이해가 깊지 않으면 적용하기가 어려운 것을 알 수 있다..] <https://www.it.uu.se/research/publications/reports/2013-025/2013-025-nc.pdf>, <http://www.cs.technion.ac.il/~erez/Papers/batch-spaa18.pdf>

C++에서는 동적 메모리를 쉽게 관리할 수 있도록 std::shared\_ptr 템플릿을 제공한다. shared\_ptr는 별도의 control\_block에 저장된 reference\_count로 메모리 재사용을 관리한다. 하지만 Load나 Store연산 실행 시 Control Block의 동시 업데이트를 필요로 하기때문에, 멀티쓰레드에서 Load/Store가 atomic하게 동작하지 않아 Data Race가 발생한다.[2]. C++11은 멀티스레드 환경에서 데이터 레이스를 방지하기 위해 std::atomic 템플릿을 제공하며, shared\_ptr는 멀티스레드 환경에서 안전하게 사용되기 위해서 std::atomic에 정의된 atomic\_load()/atomic\_store()가 함께 사용되야 한다[3].<<문장이 어색>> std::atomic 템플릿의 모든 메소드는 일반적으로 전역 해시 테이블에 저장된 뮤텍스를 사용한다.<<Visual Studio는 해시테이블 같은 것 없다. G++는? 알아보고 쓸것>> 뮤텍스의 사용은 스레드의 수와 관련없이 한 스레드만 공유 메모리에 접근할 수 있어 멀티스레드 환경의 성능을 악화시킬 뿐만 아니라, 빈번한 문맥 전환(context switch)으로 인해 캐시 사용률을 저하시킨다[4]<<요새는 틀린 사실>> .<<Refrence가 너무 옛날 이야기 현실성이 떨어진다, 1999년 이후 mutex구현에 많은 발전이 있었다.>>(1) 따라서 std::atomic을 이용한 C++ shared\_ptr의 사용은 병렬성을 높여 성능을 향상시키는 멀티스레드 환경에 부합하지 않는다. 이를 개선한 std::experimental::atomic\_shared\_ptr는 C++20부터 제공되고 현재 구현된 컴파일러가 없기 때문에 지금 구현하는 멀티스레드 프로그램에서 shared\_ptr를 사용하는데 어려움이 따른다.

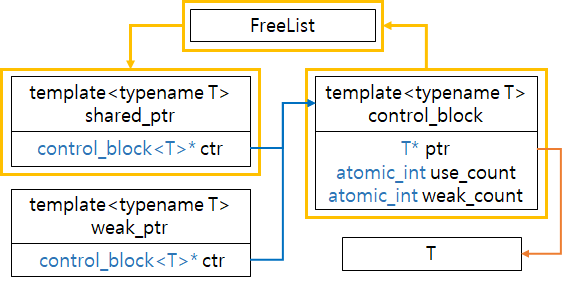
본 논문에서는 멀티스레드 환경에서 뮤텍스를 사용하지 않는<<좋은 표현이 아니다, 읽는 사람에게 나는 “Lock을 사용하지 않는 프로그램이 Lock-Free 프로그램이다”라는 잘못된 지식을 가지고 있다는 인상을 준다>> Lock-Free shared\_ptr와 Lock-Free weak\_ptr를 제안한다. 두 객체는 멀티스레드 환경에서 메모리 누수와 ABA문제를 발생시키지 않으며, 싱글 스레드 프로그램에 shared\_ptr를 적용시키는 것과 똑 같은 방법으로 멀티쓰레드에서 사용할 수 있으며, 대상 프로그램에 대한 깊은 이해가 필요 없어, 다른 메모리 관리 시스템보다 사용 난이도가 낮다. 그리고 멀티스레드 환경에서의 실험을 통해 Lock-Free shared\_ptr의 동작을 확인할 수 있다.<<벤치마크 프로그램은 성능을 재는 것이지, 정확성을 파악하는 도구가 아니다. 정확성은 반드시 증명으로만 판별가능>>

본 논문의 구성은 다음과 같다. 2장에서 Lock-Free shared\_ptr와 Lock-Free weak\_ptr의 동작에 대해 설명하고, 3 대해 논한다. 그리고 4장과 5장에서는 실험 방법과 결론 및 향후 계획에 대해 소 장에서는 ABA문제 해결에 개하였다.

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

2.1 구조

참고문헌 [d]를 통해 C++17 shared\_ptr(SP)와 weak\_ptr(WP)의 구조를 볼 수 있으며, SP/WP가 가리키는 객체를 변경하기 위해 cp와 uc(혹은 wuc)를 수정하는 것을 볼 수 있다. 하지만 두 변수는 동시에 수정되지 않기 때문에 멀티스레드 환경에서 다른 스레드의 영향을 받을 수 있으며, 이는 데이터 레이스를 야기한다. 이러한 문제를 해결하기 위해 Lock-Free 알고리즘으로 동작하는 Lock-Free shared\_ptr(LFSP)와 Lock-Free weak\_ptr(LFWP)의 구조는 다음과 같다.

 그림 1

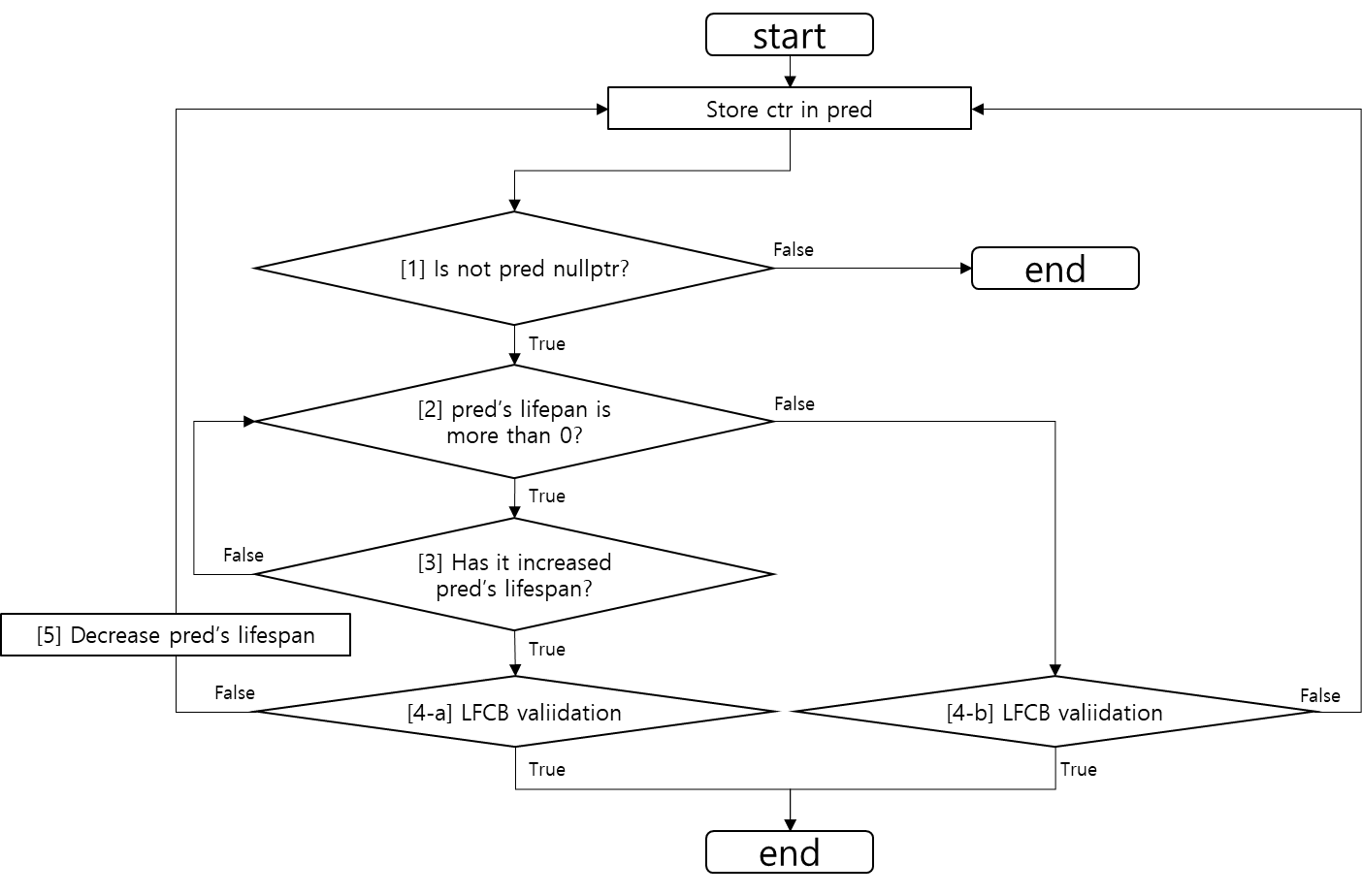
LFSP/LFWP는 유일하게 Lock-Free control\_block(LFCB)을 참조하는 포인터 ctr를 가지며, LFCB에는 객체를 참조하는 ptr 포인터, uc와 wuc에 해당하는 use\_count와 weak\_count가 있다. LFSP/LFWP는 ctr의 ptr을 이용해야 참조하는 객체에 접근할 수 있기 때문에 ctr이 nullptr를 가리킨다면 객체를 가리키지 않음을 의미한다. LFCB의 해제(delete)로 인해 발생하는 ABA문제를 해결하기위해 LFCB를 재사용하며, weak\_count가 0이된 LFCB를 FreeList 객체에 추가하는 것을 그림 1을 통해 볼 수 있다. 이러한 구조는 멀티스레드 환경에서 LFSP/LFWP의 ctr 수정만으로 가리키는 객체를 변경할 수 있음을 만족한다.

2.2 Lock-Free 알고리즘

LFSP/LFWP가 객체를 변경하기 위한 Lock-Free 알고리즘은 크게 두가지로 나뉜다. 하나는 참조할 LFCB의 count(use\_count와 weak\_count)를 증가시키는 addcopy(AC)이며, 다른 하나는 참조하는 LFCB를 변경하는 operator(OP)이다. Lock-Free 알고리즘을 만족하기 위해서 AC의 count와 ctr, OP의 ctr은 알고리즘이 완료될 때까지 다른 스레드의 영향을 받지 않아야 한다. LFSP와 LFWP에서의 AC와 CS는 count의 차이만 있으므로, 2.2.1 addcopy 알고리즘과 2.2.2 operator 알고리즘에서는 LFSP에 대해서만 논하였다.

2.2.1 addcopy 알고리즘

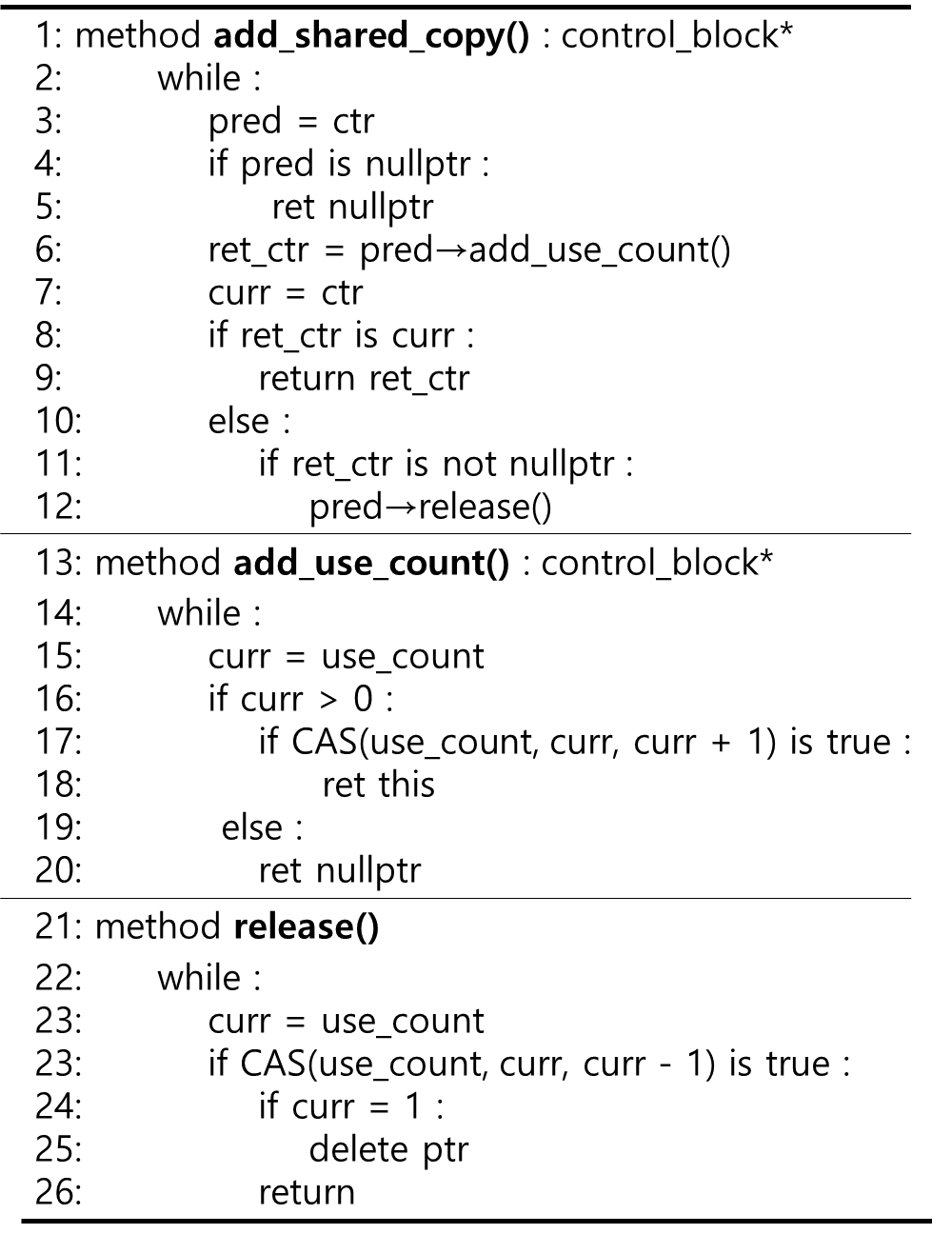
AC는 LFSP가 참조하는 ctr의 use\_count를 증가시키는 알고리즘이다. AC 알고리즘은 use\_count를 미리 증가시켜 LFCB가 재사용되지 않음을 보장하며, ctr이 다른 스레드의 영향을 받지 않았음을 LFCB 유효성검사(LFCB validation)를 통해 보장한다. 다음 그림 2는 AC알고리즘의 순서도를 보여준다.

그림 2

AC 알고리즘은 위와 같이 5개의 과정이 있으며, [1]부터 [4]까지의 과정이 만족되어야 안전하게 lifespan를 증가시킬 수 있다. 이해를 돕기 위해 lifespan을 use\_count(uc)라 하였다.

[1]은 LFSP가 객체를 가리키는지 확인한다. 만약 ctr이 nullptr인 경우 AC 알고리즘은 종료되며, 이 경우 AC 알고리즘은 실패했음을 의미한다. LFSP가 객체를 가리키고 있다면, [2]와 [3]의 순서와 같이 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 ctr이 pred와 동일한지 확인하는 LFCB 유효성 검사가 [4]에서 요구된다. [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 AC 알고리즘을 이용한 대표적인 메소드인 LFSP::add\_shared\_copy()의 의사코드다.

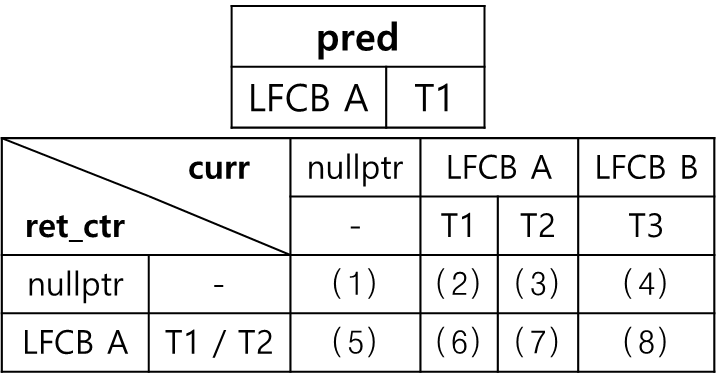
그림 3

add\_shared\_copy()는 uc를 증가시킨 경우 nullptr가 아닌 curr, 이외의 경우 nullptr를 반환하며, 각각 AC 알고리즘의 성공과 실패를 의미한다. 위의 함수가 AC 알고리즘을 만족함을 보기위해 그림 2와 3을 함께 보자. [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. 이는 uc가 0보다 큰 경우만 CAS를 이용해 uc를 1 증가시키고 LFCB를 참조하는 포인터를 반환하며, uc가 0 이하인 경우 nullptr을 반환한다(line 13~ 20). 따라서 add\_use\_count()는 [2]와 [3]에 해당하는 것을 알 수 있다. line 6에서 ret\_ctr은 add\_shared\_copy()의 반환 값, line 7에서 curr은 add\_shared\_copy() 이후 ctr을 참조하는 지역 포인터이며, line 8의 LFCB 유효성 검사에 이용된다. [4]에 해당하는 LFCB 유효성 검사인 line 8에서는 ret\_ctr과 curr을 비교하며, 동일한 경우 curr을 반환한다. 여기서 ret\_ctr과 curr은 nullptr로 동일할 수 있다. 따라서 AC 알고리즘의 성공과 실패를 알기 위해서 반환된 curr이 nullptr가 아닌지 확인해야 한다. 만약 LFCB 유효성 검사가 실패하면 release()를 호출한다. release()는 [5]에 해당하는 LFCB의 멤버 메소드로, uc를 1 감소시키고 uc가 0인 경우 객체의 메모리를 해제한다(line 21~ 26). 다음으로 LFCB 유효성 검사에 대해 자세히 알아보자.

2.2.1.1 LFCB 유효성 검사

이번 절에서는 그림 3에서의 LFCB 유효성 검사에서 발생할 수 있는 모든 상황을 가정하며, 이를 통해 AC 알고리즘의 정확성을 논한다. LFCB가 유효하다는 것은 ret\_ctr==curr인 경우로, uc를 증가시키는 과정동안 LFSP의 ctr이 다른 스레드의 영향을 받지 않음을 보장한다. LFSP와 LFWP의 LFCB 유효성 검사는 동일하며, 대표적으로 그림 3에서의 LFSP의 LFCB 유효성 검사는 다음과 같다.

line 3에서 pred는 LFCB A를 가리키며, T1 객체를 참조한다고 가정하자. line 6에서의 ret\_ctr은 uc 증가의 성공과 실패를 의미하며, 각각 LFCB A와 nullptr를 가리킨다. line 7에서의 curr은 다른 스레드에 영향을 받지 않은 경우 T1 객체를 참조하는 LFCB A를 가리키며, 반대의 경우 nullptr, LFCB A와 LFCB B를 가리킬 수 있다. 다른 스레드의 영향을 받은 경우 재사용된 LFCB A는 T2 객체, LFCB B는 T3 객체를 참조한다고 가정하자. 위의 모든 가정은 그림 4와 같이 표현할 수 있다.

 그림 4

설명을 위해 그림 2와 4를 함께 보자. 그림 4에서 ret\_ctr가 nullptr인 경우는 [2]의 False에 해당하며, LFSP가 참조하는 LFCB가 재사용될 것임을 의미한다. 따라서 [4-b]에서 LFSP가 가리키는 객체가 없는 상황 (1)에서는 LFCB가 유효하며, 이를 제외한 상황 (2), (3)과 (4)에서는 LFCB가 유효하지 않다. ret\_ctr가 LFCB A인 경우는 [3]의 True에 해당하며, LFCB A의 uc가 증가했음을 의미한다. 그러므로 [4-a]에서 curr이 LFCB A를 가리키는 상황 (6)에서는 LFCB가 유효하며, nullptr과 LFCB B를 가리키는 상황 (5)와 (8)에서는 LFCB가 유효하지 않다. 마지막으로 curr이 재사용된 LFCB A를 가리키는 상황 (7)에 대해 논해보자. pred가 LFCB A를 가리키기 때문에 [1]이 성공했음을 알 수 있다. 따라서 상황 (7)에서의 LFCB A의 재사용은 [1]과 [4] 사이에서 발생한다. 다음은 LFCB A가 재사용된 시점에 대한 가정들이며, 각각의 LFCB 유효성을 보여준다.

가정 1) LFCB A가 [2] 이전에 재사용되었다고 가정해보자. LFCB A가 재사용 되어 T2 객체를 참조한 이후 [3]에서 uc가 증가했다. 따라서 [3]은 T2 객체에 대한 uc를 증가시켰으며, curr이 가리키는 LFCB A는 T2 객체를 가리키고 있으므로 LFCB가 유효하다.

가정 2) LFCB가 [2]와 [3] 사이에서 재사용 되었다고 가정해보자. 이때 [3]이 성공했다면 가정 1)과 동일한 결과를 얻는다. 만약 LFCB A의 uc가 0이고 재사용되기 전이라면, [3]이 실패한 이후 [2]도 실패하므로 상황 (3)과 동일한 결과를 가진다.

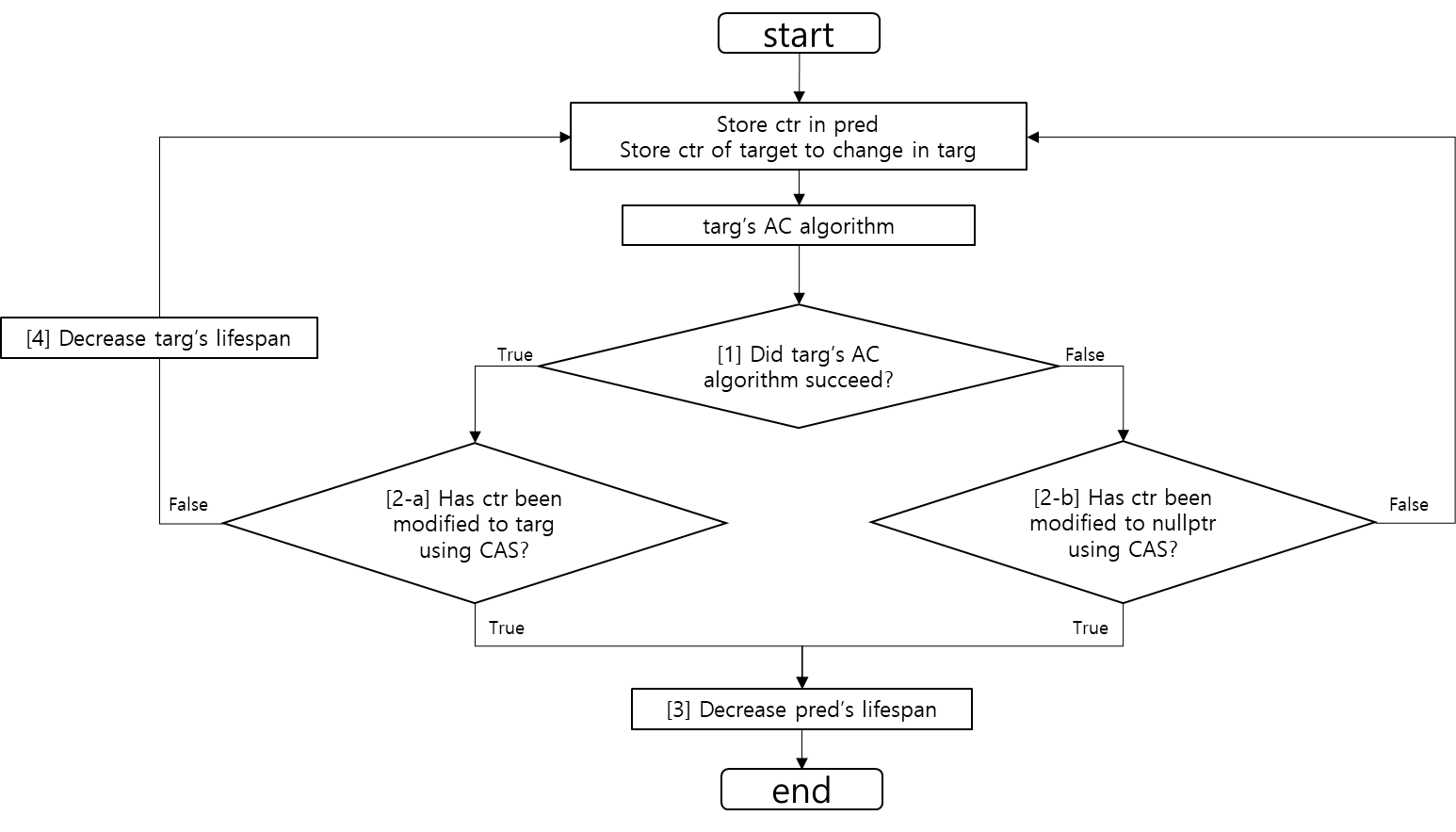
가정 3) LFCB A가 [3]과 [4] 사이에서 재사용되었다고 가정해보자. [3]에서 T1 객체를 참조하는 LFCB A의 uc가 증가했다. 이때 uc는 1이상의 값을 가지므로 LFCB A는 재사용될 수 없다. 따라서 [3] 이후에 LFCB A가 재사용되었다는 가정 3)은 모순이다.

위의 가정들을 통해 상황 (7)에서 LFCB A는 [1]과 [3]사이에서 재사용되며, [3]에서 T2 객체를 참조하는 LFCB A의 uc가 증가하기 때문에 항상 LFCB A가 유효하다는 것을 알 수 있다.

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

2.2.2 operator 알고리즘

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

 그림 5

위와 같이 OP 알고리즘은 AC 알고리즘의 결과로 [1]부터 [3]까지의 과정이 완료되야 안전하게 ctr을 수정했음을 보장할 수 있다. 그림 5를 보자. [1]에서 AC 알고리즘의 결과를 확인하며, 성공한 경우 [2-a]에서 ctr은 targ을 가리킨다. AC 알고리즘의 실패는 targ가 참조하는 객체가 없거나 targ의 재사용을 의미하기 때문에 [2-b]에서 ctr은 nullptr을 가리킨다. [2-a]와 [2-b]가 성공한 경우 pred의 uc는 [3]에서 감소되며, 이를 통해 LFCB의 uc는 정확한 값을 가진다. [4]는 [2-a]에서 CAS가 실패한 경우 targ의 AC 알고리즘에서 증가한 uc를 감소시켜 초기 상태로 돌아가는 과정이다. OP 알고리즘은 ctr을 수정하는 알고리즘이므로 실패가 없다. 따라서 OP 알고리즘은 ctr이 수정될 때까지 반복하며, ctr이 수정된 후 알고리즘이 종료된다. 다음은 그림 6은 OP 알고리즘을 이용한 대표적인 메소드인 LFSP::operator=(LFSP)의 의사코드다.

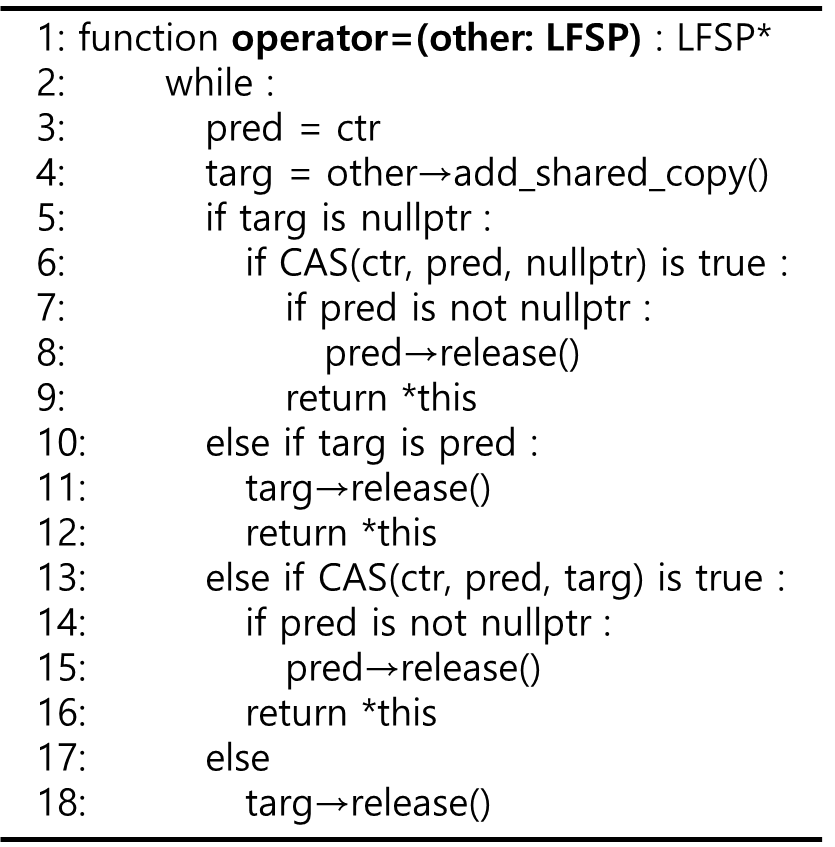
 그림 6

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, 2003.

[b] James G. Peterson, Nisha Talagala, Swamin athan Sundararaman, Sriram Subramanian, “Epoch Based Storage Management For A Storage Device”, 2016, 05

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

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

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

[4] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, "Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system," Proc. of the 3rd Symposium on Operating System Design and Implementation, pp. 87-100, 1999.

[5] K. Fraser, "Practical Lock-Freedom," Ph.D. dissertation. King's College, University of Cambridge, pp. 14, 2003.

-> “Lock-Free”

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

[d] Bjarne Stroustrup, “The C++ Programming Language (Fourth Edition)”, Addison-Wesley, pp990-995, 2013

교수님 확인용 ref

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