Lock-Free shared\_ptr와 weak\_ptr의 설계 및 구현(적절한가? weak\_ptr도 구현)

요약

기존의 C++11에서 제공하는 메모리 관리 객체인 shared\_ptr는 멀티스레드 환경에서 지원되지 않는 문제가 있다. 이는 내부의 두 멤버 변수 사이에서 데이터 레이스가 발생하기 때문이다. 이러한 이유로 C++11 std::atomic 템플릿의 멤버함수 atomic\_store()/atomic\_load()와 함께 사용한다. 하지만 이러한 방법은 뮤텍스 사용으로 인해 성능 저하를 야기한다. 본 논문에서는 기존과 동일한 방법으로 사용 가능한 Lock-Free shared\_ptr와 순환 참조 문제를 해결할 Lock-Free weak\_ptr를 제안한다. 제안하는 두 객체는 내부적으로 데이터 레이스가 발생하지 않을 뿐만 아니라 메모리 누수도 방지할 수 있으며, 이는 lock-free 알고리즘으로 구현된 C++ 소프트웨어 시스템에 유용할 것으로 기대된다.

키워드

abstract

The shared\_ptr provided by the existing C++11 is not supported in multithreaded environments. This is because a data race occurs between two member variables inside C++11 shared\_ptr. For this reason, to use C++11 shared\_ptr, the member functions atomic\_store() and atomic\_load() of C++11 std::atomic template must be used together. In this paper, we propose a Lock-Free shared\_ptr that can be used in multithreaded programs. The proposed Lock-Free shared\_ptr not only prevents data races from occurring internally, but also prevents memory leak, which is expected to contribute to the development of Lock-Free programs.

keyword

1 서론

최근 멀티코어 프로세서의 발전으로 lock-free 알고리즘이 많은 분야에 사용되고 있다. C++11은 다른 언어보다 (왜?) 높은 성능을 가지지만 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++11로 구현된 프로그램은 lock-free 기법을 사용할 때 메모리 누수(memory leak)와 ABA문제[1]가 발생한다.

이러한 문제를 해결하기 위해 Hazard Pointer[a]와 EBR(Epoch Based Reuse)[b] 등 (비교 대상이 적은가?) 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. (필요한 문단인가?)

C++11에서는 동적 메모리를 쉽게 관리할 수 있도록 shared\_ptr 객체를 제공하지만, 내부의 두 멤버 변수(비문인가?) 사이에서 데이터 레이스가 발생하기 때문에 멀티스레드 환경에서 단독으로 사용할 수 없다[2]. C++11은 멀티스레드 환경에서 데이터 레이스를 방지하기 위해 std::atomic 템플릿을 제공하며, C++11 shared\_ptr를 사용해 멀티스레드 프로그램을 구현하기 위해서 std::atomic에 정의된 atomic\_load()/atomic\_store()를 함께 사용해야 한다[3]. std::atomic 템플릿의 모든 메소드는 일반적으로 전역 해시 테이블에 저장된 뮤텍스를 사용한다. 뮤텍스 사용은 스레드의 수와 관련없이 한 스레드만 공유 메모리에 접근할 수 있어 멀티스레드를 통한 성능 향상을 악화시킬 뿐만 아니라, 빈번한 문맥 전환(context switch)으로 인해 캐시 사용률을 저하시킨다[4].(1)(필요한 문장인가?) 따라서 std::atomic을 이용한 C++11 shared\_ptr는 병렬성을 높여 성능을 향상시키는 lock-free 기법에 부합하지 않는다. Lock-Free 알고리즘의 shared\_ptr는 C++20에서 제공될 예정이다. (내용이 적절한가? + 이것의 필요성에 대한 사이트)

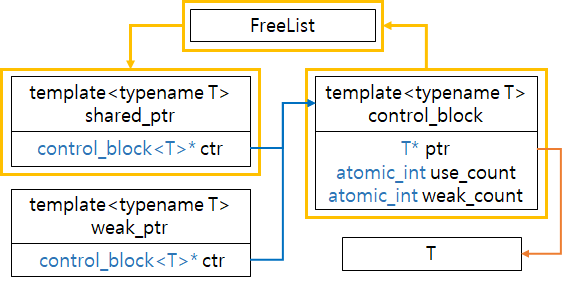
본 논문에서는 C++11의 shared\_ptr와 동일한 방법으로 사용할 수 있는 Lock-Free shared\_ptr와 순환 참조 문제를 해결할 Lock-Free weak\_ptr를 제안한다. 두 객체는 멀티스레드 환경에서 메모리 누수와 ABA문제를 발생하지 않으며, 기존과 동일한 방법으로 사용할 수 있기 때문에 다른 메모리 관리 시스템보다 사용 난이도가 낮다. 그리고 멀티스레드 환경에서의 메소드 실행 속도 비교 실험을 통해 std::atomic을 이용한 C++11 shared\_ptr보다 우수한 성능을 가진 lock-free shared\_ptr임을 증명하였다.

본 논문의 구성은 다음과 같다. 2장에서 제안하는 LFSP와 LFWP의 동작에 대해 설명하고, 3장과 4장에서는 정확성과 ABA문제 해결에 대해 논한다. 그리고 5장에서 실험을 통해 멀티스레드 환경에서의 기존 shared\_ptr와 비교하고, 6장에서는 결론 및 향후 계획에 대해 소개하였다.

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

2.1 구조

참고문헌 [d]를 통해 C++11 shared\_ptr(SP)와 weak\_ptr(WP)의 구조를 볼 수 있으며, 가리키는 객체를 변경하는 과정에서 cp와 uc(혹은 wuc)가 각각 수정되는 것을 볼 수 있다. 두 변수는 동시에 수정되지 않기 때문에 멀티스레드 환경에서 다른 스레드의 영향을 받을 수 있어 데이터 레이스를 야기한다. 이러한 문제를 해결하고 lock-free 알고리즘으로 동작하기 위한 Lock-Free shared\_ptr(LFSP)와 Lock-Free weak\_ptr(LFWP)의 구조는 다음과 같다.

 그림 1

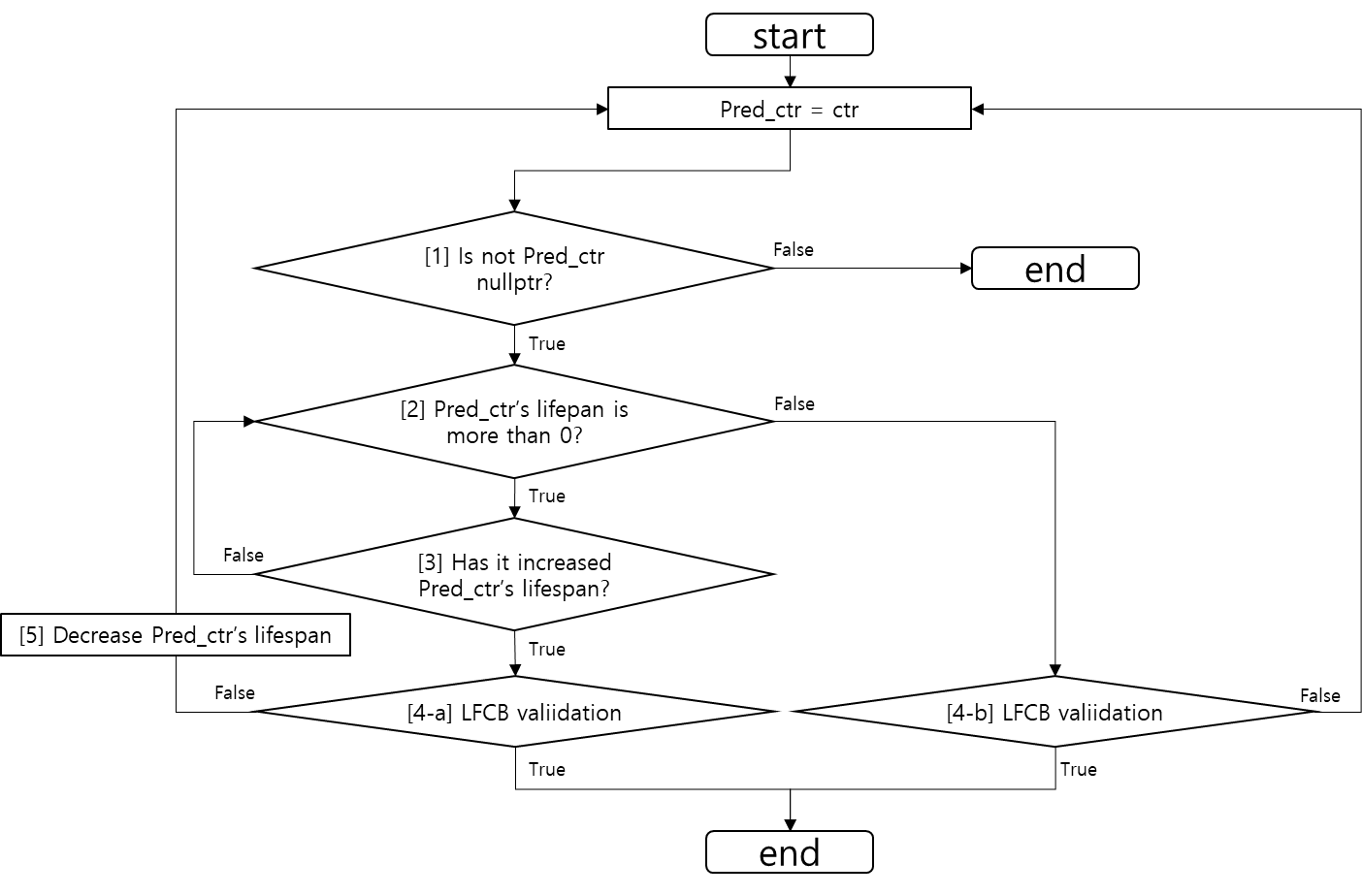
SP/WP와 다르게 LFSP/LFWP에는 유일하게 Lock-Free control\_block(LFCB)을 가리키는 포인터 ctr이 있으며, LFCB에는 객체를 가리키는 ptr 포인터와 SP의 uc와 WP의 wuc에 해당하는 use\_count와 weak\_count가 있다. LFSP/LFWP는 ctr의 ptr을 이용해야 참조하는 객체에 접근할 수 있기 때문에 ctr==nullptr이라면 객체를 참조하지 않음을 의미한다. 추가된 FreeList 객체는 멀티스레드 환경에서의 LFCB의 해제(delete)로 인한 ABA문제를 해결하기위해 LFCB를 재사용하며, weak\_count가 0이된 경우 FreeList에 해당 LFCB를 추가한다. 이러한 구조는 멀티스레드 환경에서 LFSP/LFWP가 가리키는 객체는 ctr 수정만으로 변경될 수 있음을 만족한다.

2.2 Lock-Free 알고리즘

LFSP/LFWP이 가리키는 객체의 변경을 위한 Lock-Free 알고리즘은 크게 addcopy(AC)와 operator(OP) 두가지가 있으며, AC는 use\_count나 weak\_count를 증가, OP는 ctr을 수정하는 알고리즘이다. LFSP와 LFWP에서의 AC와 CS는 사용하는 변수(use\_count 혹은 weak\_count)의 차이만 있으므로, 2.2.1절과 2.2.2절에서는 LFCB 재사용과 관련된 LFWP에 대해서만 논하였다.

2.2.1 addcopy 알고리즘

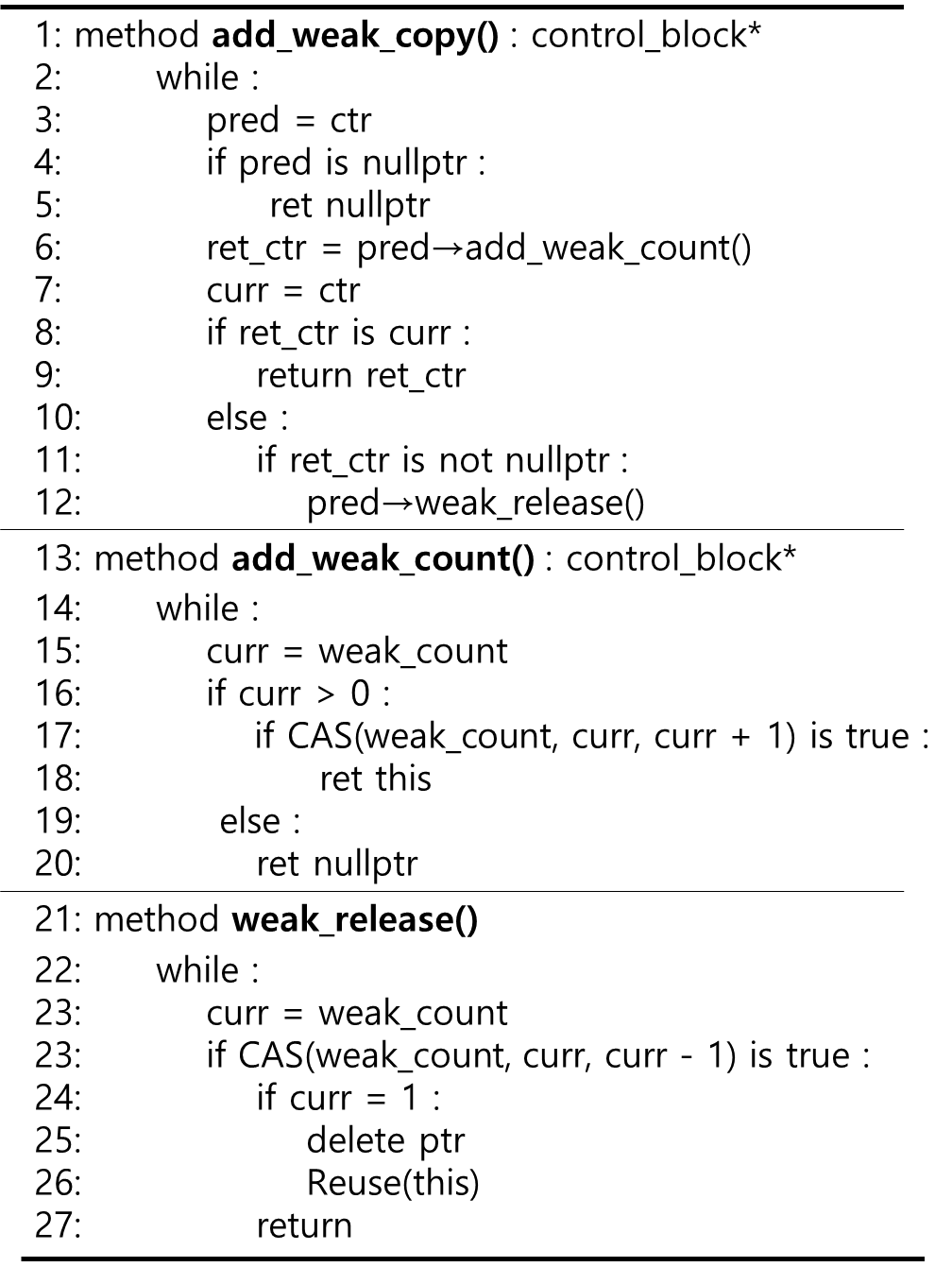
AC 알고리즘은 LFWP가 참조하는 LFCB(ctr)의 weak\_count를 1씩 증가시킨다. 이는 미리 weak\_count를 증가시킴으로써 LFCB 재사용으로부터 LFCB 접근의 안전성을 보장한다. AC 알고리즘의 LFWP가 다른 스레드의 영향을 받지 않았다는 정확성은 LFCB 유효성검사(LFCB validation)를 통해 보장한다. 다음의 그림 2는 AC알고리즘의 순서도를 보여준다.

그림 2

위와 같이 AC 알고리즘은 5개의 과정이 있으며, 그림 2 [1]부터 [4]까지의 과정이 만족되어야 안전하게 lifespan가 증가했음을 보장한다. 순서도의 이해를 돕기 위해 lifespan을 weak\_count(wc)라고 하겠다.

LFWP가 객체를 가리키지 않는 [1]은 Pred\_ctr =! nullptr을 검사하며, 증가시킬 wc가 없음을 의미한다. LFWP가 가리키는 객체가 존재한다면 그림 2 [2]와 [3]의 순서와 같이 wc가 0보다 큰 값인지 검사한 뒤 CAS를 이용해 wc를 증가시켜야 한다. 이는 다른 스레드로부터 LFCB가 재사용되지 않았음을 확인한 뒤 wc를 증가시킴으로써, 다른 스레드가 해당 LFCB를 재사용할 수 없음을 보장할 수 있기 때문이다. 그림 2 [1]과 [3] 사이에서 LFWP는 다른 스레드에 의해 다른 LFCB를 참조할 수 있으며, 이를 위해 그림 2 [4]에서는 LFWP가 미리 증가시킨 wc에 대한 LFCB를 가리키고 있음을 확인하는 LFCB 유효성 검사가 요구된다. 그림 2 [5]는 그림 2 [3] 이후 LFCB 유효성 검사에서 실패한 경우 wc를 감소시켜 이전의 상태로 돌아가는 과정이다. LFCB 유효성 검사의 자세한 내용은 2.2.1.1 LFCB 유효성 검사에서 논한다.

다음은 AC 알고리즘을 이용한 대표적인 메소드인 LFWP::add\_weak\_copy()의 의사코드다.

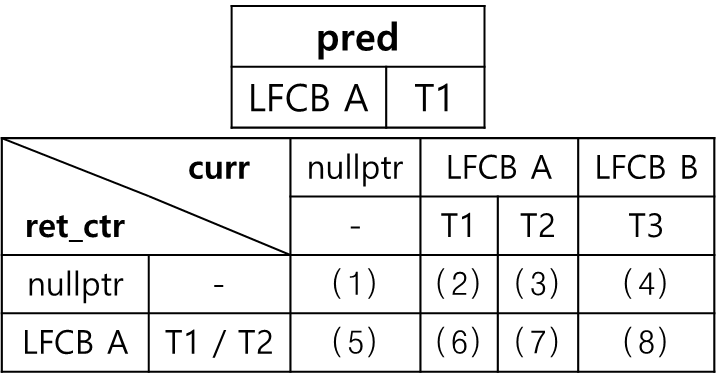
그림 3

위의 함수가 AC 알고리즘을 만족함을 보기위해 그림 2와 3을 함께 보자. 그림 3의 LFWP::add\_weak\_copy()는 wc를 증가시킨 경우 LFCB 포인터를 반환하고 반대의 경우 nullptr을 반환하며, 각각 AC 알고리즘의 성공과 실패를 의미한다. line 4는 그림 2 [1]에 해당하며, pred가 nullptr인 경우 nullptr을 반환한다. line 6의 add\_weak\_count()는 그림 2 [2]와 [3]에 해당하며, LFCB의 멤버 메소드이다. 이는 uc가 0보다 큰 경우에만 CAS를 이용해 wc를 1 증가시키고 LFCB 포인터를 반환하며, 0 이하인 경우 nullptr을 반환한다. ret\_ctr과 curr은 각각 add\_weak\_copy()와 ctr을 가리키는 지역 변수이며, LFCB 유효성 검사에 이용된다. line 8은 LFCB 유효성 검사로 그림 2 [4]에 해당한다. 이는 ret\_ctr과 curr을 비교해 동일한 경우 curr을 반환하며, AC 알고리즘의 성공과 실패를 결정한다. LFCB 유효성 검사가 실패하면 line 12의 weak\_release()를 통해 그림 2 [5]의 과정을 가진다. weak \_release()는 add\_use\_count()와 동일하게 LFCB의 멤버 메소드로 wc를 1 감소시키고 wc가 0인 경우 객체의 메모리를 해제하고 LFCB를 재사용한다.

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. 그림 3에서 line 8은 LFCB 유효성 검사에 해당하며, 이를 만족하는 경우 LFCB가 유효하다고 판단한다. LFCB가 유효한 경우 uc를 증가시키는 과정에서 LFSP가 다른 스레드의 영향을 받지 않았음을 보장할 수 있다. AC 알고리즘의 정확성을 위한 그림 3의 LFCB 유효성 검사는 다음과 같다.

line 3에서 pred는 LFCB A를 가리키며, T1 객체를 참조한다고 가정하자. line 6에서의 ret\_ctr은 uc 증가의 실패와 성공을 의미하며, 각각 nullptr와 LFCB A를 가리킬 수 있다. 이때 LFCB A는 재사용되어 T2 객체를 참조할 수 있으며, 이 경우 add\_use\_count()의 반환된 값이 nullptr이 아니므로 T2 객체에 대한 uc를 증가시켰음을 알 수 있다. line 7에서의 curr은 다른 스레드에 영향을 받지 않은 경우 T1 객체를 참조하는 LFCB A를 가리킬 수 있으며, 반대의 경우 nullptr, 재사용되어 T2 객체를 참조하는 LFCB A와 LFCB B를 가리킬 수 있다. 다음 그림 4는 ret\_ctr과 curr이 참조하는 LFCB에 대한 모든 상황을 보여준다.

 그림 4

설명을 돕기 위해 그림 2와 4를 함께 보자. 8개의 상황 중 그림 2 [2]의 False를 나타내는 ret\_ctr=nullptr는 uc의 증가가 실패했음을 의미한다. 이때 LFSP가 가리키는 객체가 없는 경우에만 AC 알고리즘이 정확하다고 할 수 있다. 따라서 그림 2 [4-b]에서 상황 (1)만 LFCB가 유효하며, 이를 제외한 상황 (2), (3)과 (4)는 LFCB 유효성 검사의 실패를 의미한다. 그림 2 [3]의 True에 해당하는 ret\_ctr=LFCB A는 LFCB A의 uc 증가의 성공을 의미한다. 그러므로 그림 2 [4-a]에서 curr이 LFCB A를 가리키는 상황 (6)은 LFCB가 유효하지만, nullptr과 LFCB B를 가리키는 상황 (5)와 (8)은 LFCB가 유효하지 않다. 상황 (5)와 (8)은 그림 2 [3]에서 curr과 관계없는 uc를 증가시켰으므로, 이를 다시 감소시키기 위해 그림 2 [5]의 과정이 필요하다. 이제 curr이 재사용된 LFCB A를 가리키는 상황 (7)에 대해 논해보겠다. pred가 LFCB A를 가리키기 때문에 상황 (7)에서의 LFCB A의 재사용은 그림 2 [1]과 [4] 사이에서 발생함을 알 수 있다. 다음은 LFCB A의 재사용 시기를 가정해 LFCB 유효성을 보여준다.

가정 1) LFCB A가 그림 2 [2] 이전에 재사용되었다고 가정해보자. LFSP는 LFCB A가 재사용 되어 T2를 참조한 이후 uc를 증가시켰다. 따라서 curr이 LFCB A인 경우 LFSP가 참조하는 LFCB A는 T2를 가리키고 있으며, T2에 대한 uc를 증가시켰으므로 LFCB가 유효하다.

가정 2) LFCB가 그림 2 [2]와 [3] 사이에서 재사용 되었다고 가정해보자. 이는 가정 1)과 동일한 결과를 얻는다. 만약 LFCB A의 uc가 0이고 재사용되기 전이라면, 그림 2 [3]은 실패하고 그림 2 [2]에서 False를 의미하므로 상황 (3)과 동일한 결과를 가진다.

가정 3) LFCB A가 그림 2 [3] 이후에 재사용되었다고 가정해보자. LFSP는 T1을 참조하는 LFCB A의 uc를 증가시켰으며, LFCB 유효성 검사에서 T1을 참조하는 LFCB A와 T2를 참조하는 LFCB A를 비교해야 한다. 하지만 그림 2 [3]에서 uc가 증가했다면 다른 스레드의 영향을 받아도 최소 1이상의 값을 갖기 때문에 LFCB A는 재사용될 수 없다. 따라서 LFCB A는 그림 2 [3] 이후에 재사용될 수 없다.

이를 통해 상황 (7)에서의 LFCB A는 그림 2 [1]과 [3]사이에서 재사용되며, 가정을 통해 항상 LFCB A가 유효함을 볼 수 있다. 따라서 AC 알고리즘의 정확성은 LFCB 유효성 검사를 통해 보장할 수 있다.

2.2.2 operator 알고리즘

OP 알고리즘은 LFSP의 ctr을 수정하는 알고리즘이다. 이는 AC 알고리즘을 통해 uc를 미리 증가시킨 뒤 ctr을 수정해 다른 스레드로부터의 영향을 받지 않음을 보장한다.

4 ABA문제 해결

5 실험

6 결론

reference

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

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

[b] 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)