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]. 하지만 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)

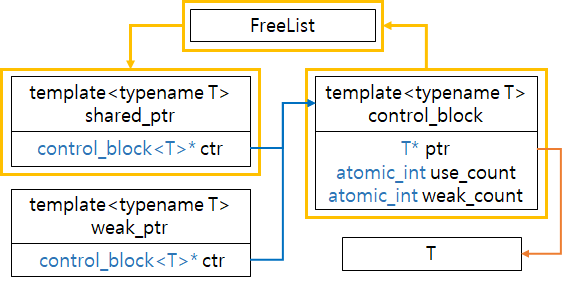
본 논문에서는 C++11의 shared\_ptr와 동일한 방법으로 사용할 수 있는 Lock-Free shared\_ptr와 순환 참조 문제를 해결할 Lock-Free weak\_ptr를 제안한다. 두 객체는 멀티스레드 환경에서 메모리 누수와 ABA문제를 발생하지 않으며, 기존과 동일한 방법으로 사용할 수 있기 때문에 다른 메모리 관리 시스템보다 사용 난이도가 낮다. 그리고 멀티스레드 환경에서의 메소드 실행 속도 비교 실험을 통해 std::atomic을 이용한 C++11 shared\_ptr보다 우수한 성능을 가진 lock-free shared\_ptr임을 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 다른 구조를 가진 LFSP/LFWP를 그림 1을 통해 볼 수 있다. LFSP/LFWP의 유일한 포인터 ctr은 Lock-Free control\_block(LFCB)을 가리키며, ctr의 ptr을 이용해야 참조하는 객체에 접근할 수 있기 때문에 ctr==nullptr이라면 객체를 참조하지 않음을 의미한다. LFCB는 객체를 가리키는 ptr 포인터와, SP의 uc와 WP의 wuc에 해당하는 use\_count와 weak\_count가 있다. 멀티스레드 환경에서 LFCB의 해제(delete)로 인한 ABA문제를 해결하기위해 LFCB를 재사용하며, 이를 관리하기 위한 FreeList 객체가 추가하였다. use\_count가 0이 된 경우 가리키는 객체(T)의 메모리를 반환하지만, weak\_count가 0이된 경우 FreeList에 해당 LFCB를 추가한다. 이러한 구조는 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, AC 알고리즘은 use\_count와 weak\_count를 증가시키고 OP 알고리즘은 ctr을 수정한다. LFSP와 LFWP에서의 AC과 CS는 사용하는 변수(use\_count 혹은 weak\_count)의 차이만 있으므로, 2.2.1절과 2.2.2절에서는 LFSP에 대해서만 논한다.

2.2.1 addcopy 알고리즘

LFSP에서의 AC는 다른 스레드의 영향 없이 ctr의 use\_count(uc)를 1씩 증가시킨다. 이는 uc를 증가시킴으로써 메모리 해제로부터의 LFCB 메모리 접근의 안전성을 보장하며, LFCB 유효성검사(LFCB validation)를 통해 증가한 use\_count의 ctr이 다른 스레드로부터 영향을 받지 않았음을 보장한다. 알고리즘의 순서도는 다음과 같으며, 여기서 lifespan은 uc를 의미한다.

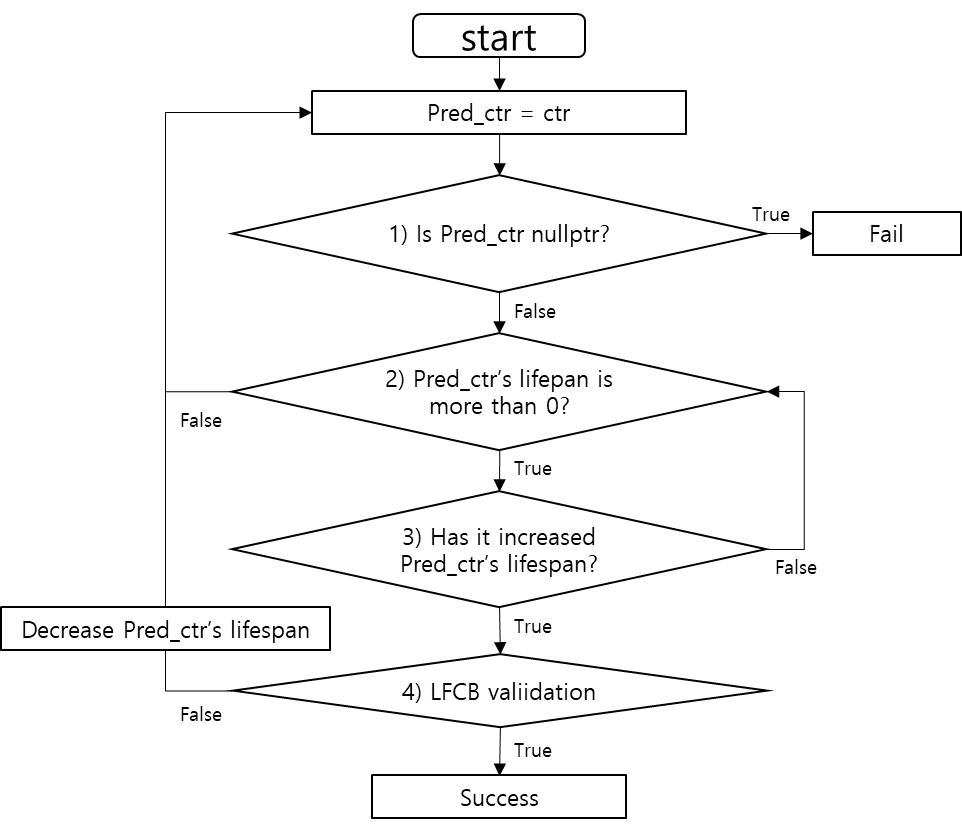
그림 2

그림 2의 1)은 LFSP의 ctr이 nullptr일 수 있으므로 ctr(LFCB)의 메모리에 접근할 수 있는지 확인하는 작업이다. ctr에 접근할 수 있다면 uc가 0 이상인지 확인한다. 만약 uc가 0이라면 해당 객체 메모리가 해제됨(혹은 해제 중)을 의미하기 때문이다. uc가 0 이상인 경우 CAS를 이용해 uc를 1 증가시켜 LFCB의 재사용을 방지한다. 여기서 CAS의 성공은 Pred\_ctr의 uc가 증가했음을 보장하지만, LFSP가 가리키는 객체를 Pred\_ctr가 참조한다는 보장을 할 수 없다. 그림 2의 1)과 2) 사이에서 다른 스레드에 의해 ctr이 재사용 되어 다른 객체를 참조한다면, ctr과 동일한 LFCB를 가리키는 Pred\_ctr은 LFSP와 관계없는 LFCB를 참조하기 때문이다. 이러한 이유로 Pred\_ctr==ctr인 LFCB 유효성 검사가 요구된다. LFCB 유효성 검사의 성공/실패와 관련된 자세한 내용은 3절 LFCB 유효성 검사에서 자세하게 논한다.

AC는 그림 2의 3)과 4)과 같이 다른 스레드의 영향을 받을 수 있다. 3)에서의 CAS의 실패는 다른 스레드에 의해 uc가 변경된 경우로 이미 uc가 0이 되어 해당 LFCB가 FreeList에 추가되었을 수 있음을 의미하며, 4)에서의 LFCB 유효성 검사의 실패는 LFSP와 관련이 없는 객체의 uc의 증가를 의미한다. 그러므로 두 경우 모두 AC 알고리즘을 재시도하며, uc를 이미 증가시킨 4)의 실패는 재시도 이전에 CAS를 이용한 uc의 감소가 요구된다.

LFSP의 멤버 메소드 add\_shared\_copy()는 AC 알고리즘을 이용한 대표적인 메소드로 다음과 같다.

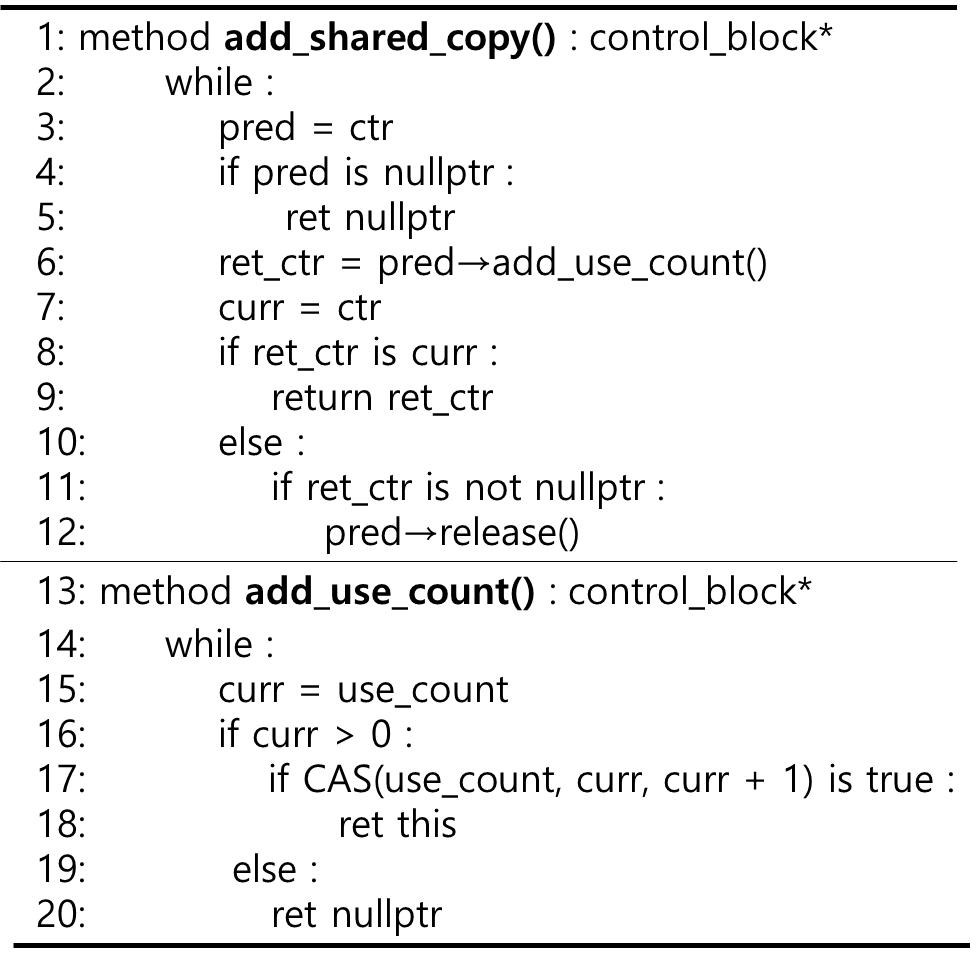
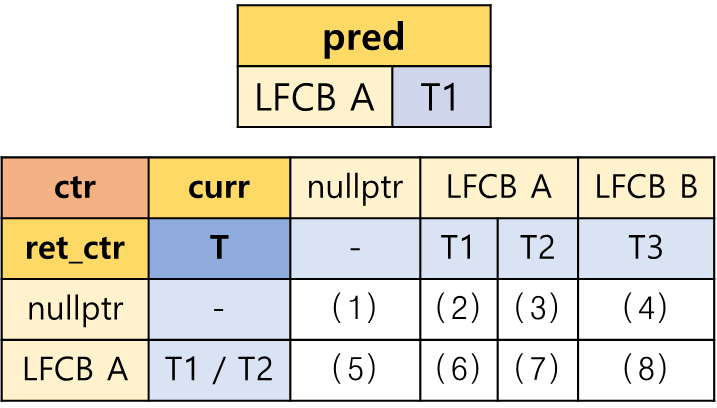
그림 3

그림 3의 line 6에서 볼 수 있는 add\_use\_count()는 LFCB의 멤버 메소드로, line 16 ~ 20을 통해 그림 2의 2)와 3)을 실행하는 것을 볼 수 있다. 이는 uc가 0 이상이 아닌 경우 nullptr, CAS가 성공한 경우 LFCB를 반환하며, 각각 그림 2에서의 2)의 실패와 3)의 성공을 의미한다. line 11에서 볼 수 있는 LFCB의 release()는 uc를 1씩 감소시키는 함수로, uc가 감소된 후 0이 되었다면 참조하는 객체의 메모리를 반환한다.

2.2.2 construct 알고리즘

3 정확성

이번 절에서는 AC 알고리즘의 LFCB 유효성 검사에서 발생할 수 있는 모든 상황을 가정하고 이를 통해 LFSP/LFWP가 참조하는 LFCB의 정확성을 논한다. 그림 4는 그림 3의 line 6과 7에서 발생하는 모든 상황을 보여준다.

 그림 4

pred는 LFCB A를 참조하며, T1 객체를 소유한다고 가정하자. 이때 ret\_ctr은 LFCB A나 nullptr를 가리킬 수 있으며, curr은 nullptr, T1을 참조하는 LFCB A, T2를 참조하는 재사용된 LFCB A, T3을 참조하는 재사용된 LFCB B를 가리킬 수 있다.

그림 4의 (2), (3), (4)는 ret\_ctr이 nullptr이므로 0이상이 아닌 객체에 대해 그림 3

(1)은 소유한 객체가 없으므로 LFCB가 유효하며, curr (nullptr)을 반환(SAS7)해 소유권 획득 실패를 알린다. (4)는 소유권 획득을 실패했지만 LFSP가 T3을 소유하고 있으므로 재시도(SAS11)한다. T1의 소유권을 획득했지만 LFSP가 T1을 소유하지 않는 (5)와 T3을 소유한 (8)은 소유하는 객체가 변경되었기 때문에 획득한 T1의 소유권을 포기(SAS10)한 후, 재시도(SAS11)한다. (4),(5),(8)은 소유하지 않은 객체의 소유권 획득시도를 의미하므로 LFCB가 유효하지 않음을 의미한다.

세 가지 (add\_use\_count(), release()로 인해 T1의 use\_count가 0인 경우, A의 재사용)의 실행 순서에 따라 (2),(3),(6),(7)이 발생할 수 있다. T1의 소유권을 획득했고 LFSP가 여전히 T1을 소유한 (6)은 ctr이 변경되지 않았으므로 LFCB가 유효함을 의미한다. (2)는 release()로 use\_count가 0이 된 T1의 소유권 획득을 시도한 상황으로 LFCB A가 재사용 될 수 있음을 의미하며, (3)은 (2) 이후 A가 재사용되어 T3을 가리키는 상황으로 LFCB가 재사용되었음을 의미한다. (7)은 (2)의 release()와 add\_use\_count()사이에 A가 재사용되는 상황으로, 소유권포기 이후 A가 재사용되어 T2를 참조하고 LFSP가 재사용된 A를 소유한 뒤, 소유권을 획득한 상황으로 T2의 소유권 획득을 의미한다. (2)와 (3)은 소유권 획득을 실패(SAS4)했으며 LFCB가 유효하지 않기 때문에 재시도(SAS11)하며, (6)과 (7)은 각각 LFSP가 소유한 T1과 T2의 소유권을 획득했고, 이를 통해 LFCB가 유효함을 확인했으므로 curr을 반환(SAS7)해 소유권 획득을 알린다.

이처럼 add\_shared\_copy()는 LFCB 유효성 검사를 통해 ABA문제를 회피하며, 소유한 객체의 소유권 획득을 보장한다.

4 ABA문제의 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)