|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | **작성자** | **2014132002 구태균** | **팀명** | - |
| **주차** | **24 주차** | **기간** | **2020.06.25 ~ 2020.07.01** | **지도교수** | **정 내 훈** (서명) |
| **이번주 한일** | * Lock-free weak\_ptr * Lock-free 설명 | | | | |

**[ Lock-free weak\_ptr ]**

1. weak\_ptr() // x : 하나의 thread에서만 생성 // 변경
2. weak\_ptr(nullptr\_t) // x : weak\_ptr() 와 동일
3. weak\_ptr(const shared\_ptr<Tp>& other) // o : other.ctr.weak\_add\_ref() / ctr=other.ctr // 변경
4. weak\_ptr(const weak\_ptr<Tp>& other) // o : other.ctr.weak\_add\_ref() / ctr=other.ctr // 변경
5. ~weak\_ptr() // x : ctr.weak\_release()만 구현 // 변경
6. weak\_ptr& operator=(nullptr\_t) // o : ctr.weak\_release() / CAS(ctr = nullptr) // 변경
7. weak\_ptr& operator=(const shared\_ptr<Tp>& other)

// o : ctr.weak\_release() / other.ctr.weak\_add\_ref() / CAS(ctr = other.ctr) // 변경

1. weak\_ptr& operator=(const weak\_ptr& other)

// o : ctr.weak\_release() / other.ctr.weak\_add\_ref() / CAS(ctr = other.ctr) // 변경

1. shared\_ptr<Tp> lock() // ? : CAS-method 사이에서 읽기는? (ctr->ptr) // 변경
2. void reset() // o : ctr->weak\_release() // CAS(ctr = nullptr)
3. void swap(weak\_ptr& other) // o : ctr = other.ctr & other.ctr = ctr을 동시에
4. Tp\* get() const // 추가
5. control\_block<Tp>\* weak\_add\_ref() const // 추가

**lock-free control\_block**

Bool control\_block::CAS(std::atomic\_int\* memory, int old\_data, int new\_data) const

{

int old\_value = old\_data;

int new\_value = new\_data;

return std::atomic\_compare\_exchange\_strong

(reinterpret\_cast<std::atomic\_int\*>(memory), &old\_value, new\_value);

}

**[ control\_block<Tp>\* control\_block::add\_ref\_copy() ]**

Val = Use\_count

Val == 0

Val > 0

CAS(… , +1) == true

CAS(… , +1) == false

Return this

Return nullptr

1. use\_count를 val에 store한다.
2. val == 0 인 경우, delete 중이다. 따라서, val == 0 인 control\_block은 더 이상 쓰이지 않을(메모리 해제중인) 메모리이다. 그러므로 return nullptr
3. val > 0 인 경우, 의미있는 Use\_count를 저장(할당된 메모리)하고 있다.

CAS(… ,+1)을 통해 true라면 control\_block 본인을 return , false라면 use\_count가 변경되었기 때문에 처음으로 돌아가 다시 루프를 돈다.

control\_block<Tp>\* control\_block::add\_ref\_copy()

{

// 지역변수 pred\_value

int pred\_value;

while (true)

{

pred\_value = use\_count;

// 1) Pred\_value > 0 인 경우, delete가 되지 않은(사용중인) 메모리이다.

// -> CAS()를 통해 공유메모리 use\_count를 1 증가를 시도한다.

if (pred\_value > 0)

{

// 1.1) 다른 쓰레드의 간섭없이 use\_count를 1증가시키기 성공했다.

if (true == CAS(&use\_count, pred\_value, pred\_value + 1))

return this;

// 1.2) 다른 쓰레드에 의해 공유 메모리 use\_count가 변경되어 pred\_val와 다를수 있다.

// -> CAS()의 false 리턴으로, 다시 루프를 돌게 된다.

else

continue;

}

// 2) Pred\_value == 0 인 경우, use\_count는 절대로 증가하지 않는다.

// -> 더 이상 쓰이지 않을 control\_block이므로 nullptr를 리턴한다.

else

{

return nullptr;

}

}

}

**[void release()]**

Curr\_use\_count = use\_count

CAS(… , +1) == true

CAS(… , +1) == false

Curr\_use\_count == 1

Destroy(…)

Return

Curr\_use\_count != 1

1. use\_count를 curr\_use\_count에 store한다.
2. CAS(… ,-1)을 통해 true라면 계속, false라면 use\_count가 변경되었기 때문에 다시 루프를 돈다.
3. Curr\_use\_count == 1 이라면, CAS()를 통해 use\_count == 0 이 된것이고 더 이상 쓰이지 않을 ptr이므로 destroy()를 통해 메모리를 해제한다.
4. Control\_block에서 counter를 가지고 계산하기 때문에, Release()는 CAS()가 실패해도 계속 시도하면 언젠가는 성공한다. 따라서, release()가 return했다면 1 감소(혹은 destroy)가 성공했다는 의미이다.

void release()

{

// 지역변수 curr\_use\_count

int curr\_use\_count;

while (true)

{

curr\_use\_count = use\_count;

// 1) CAS()를 통해 공유메모리 use\_count를 1 감소를 시도한다.

if (true == CAS(&use\_count, curr\_use\_count, curr\_use\_count - 1))

{

// 1.1) curr\_use\_count == 1 이라면, 감소된 use\_count는 0이다.

if (curr\_use\_count == 1)

destroy(std::bool\_constant<std::conjunction\_v<\_Can\_enable\_shared<Tp>>>{});

// 1.2) CAS() 성공

return;

}

// 2) 다른 쓰레드에 의해 공유 메모리 use\_count가 변경되어 curr\_use\_count와 다를수 있다.

// -> 처음으로 돌아가 use\_count를 다시받아 CAS를 시도한다.

else

continue;

}

}

**[Tp\* get()]**

Tp\* get()

{

// 지역변수 curr\_use\_count

int curr\_use\_count = use\_count;

// 1) curr\_use\_count > 0 이라면, 사용중인 control\_block

if (use\_count > 0)

return ptr;

// 2) curr\_use\_count == 0 이라면, 더 이상 사용되지 않을 control\_block

else

return nullptr;

}

**[control\_block<Tp>\* weak\_add\_ref()]**

: control\_block<Tp>\* control\_block::add\_ref\_copy() 와 동일한 알고리즘

control\_block<Tp>\* weak\_add\_ref() 에서는 공유 변수로 weak\_count를 사용한다.

**[void weak\_release()]**

: void release() 와 동일한 알고리즘

void weak\_release() 에서는 공유변수로 weak\_count를 사용하고, weak\_count가 0일 경우 delete control\_block

**lock-free shared\_ptr**

bool CAS(control\_block<Tp>\*\* memory, control\_block<Tp>\* oldaddr, control\_block<Tp>\* newaddr)

{

int old\_addr = reinterpret\_cast<int>(oldaddr);

int new\_addr = reinterpret\_cast<int>(newaddr);

return std::atomic\_compare\_exchange\_strong

(reinterpret\_cast<std::atomic\_int\*>(memory), &old\_addr, new\_addr);

}

**[control\_block<Tp>\* shared\_ptr::add\_ref\_copy()]**

Return Ret\_ctr

Ret\_ctr != nullptr

Ret\_ctr == nullptr

Ret\_ctr = curr\_ctr->add\_ref\_copy()

curr\_ctr == nullptr

curr\_ctr != nullptr

Curr\_ctr = ctr

Return nullptr

1. Curr\_ctr에 ctr을 store한다.
2. Curr\_ctr이 nullptr이라면 use\_count를 증가시킬수 없으므로 return nullptr,

Curr\_ctr이 존재한다면 ret\_ctr에 curr\_ctr의 add\_ref\_copy() return값을 store한다.

1. Add\_ref\_copy()의 return값은 두가지가 존재하는데,

nullptr일 경우 control\_block의 use\_count == 0(의미없는 control\_block)이므로, 처음으로 돌아가 루프를 돌면서 curr\_ctr == nullptr이 되어 return한다.

nullptr이 아닐 경우 use\_count를 1 증가시킨 ret\_ctr를 return한다.

control\_block<Tp>\* shared\_ptr::add\_ref\_copy()

{

// 지역변수 curr\_ctr, ret\_ctr

control\_block<Tp>\* curr\_ctr;

control\_block<Tp>\* ret\_ctr;

do {

curr\_ctr = ctr;

// 1) curr\_ctr이 nullptr라면, 가리키는 control\_block(ptr)이 없다.

if (curr\_ctr == nullptr)

return nullptr;

// 2) curr\_ctr의 add\_ref\_copy()를 ret\_ctr에 store한다.

ret\_ctr = curr\_ctr->add\_ref\_copy();

// ret\_ctr이 nullptr일수 있을까?? (이미 본인이 shared\_ptr을 1 증가시켜놓은 상태)

// 2.1) ret\_ctr이 nullptr이라면, 중간에 control\_block이 delete되었다.

} while (ret\_ctr == nullptr);

// 2.2) 공유메모리 use\_count를 1 증가시킨 ret\_ctr(ctr)을 리턴.

return ret\_ctr;

}

**[shared\_ptr(const shared\_ptr& other)]**

shared\_ptr(const shared\_ptr& other)

{

// ctr을 CAS() 없이 그냥 넣어도 괜찮을까?

ctr = other.add\_ref\_copy();

}

**[shared\_ptr& operator=(nullptr\_t)]**

shared\_ptr& operator=(nullptr\_t)

{

// 지역변수 pred\_ctr

control\_block<Tp>\* pred\_ctr;

while (true)

{

pred\_ctr = ctr;

// 1) CAS()로 ctr을 nullptr로의 변경을 시도한다.

if (true == CAS(&ctr, pred\_ctr, nullptr))

{

// 2) CAS()가 성공하면 기존에 가지고 있던 ctr(pred\_ctr)을 release한다.

if(pred\_ctr != nullptr)

pred\_ctr->release();

return \*this;

}

// 2.1) CAS가 실패하면 다시시도.

}

}

**[shared\_ptr& operator=(const** **shared\_ptr& other)]**

Pred\_ctr = ctr

Other\_ctr = other.add\_ref\_copy()

Other\_ctr == nullptr

CAS(ctr, nullptr) ==false

CAS(ctr, nullptr) == true

Other\_ctr == pred\_ctr

Other\_ctr->release()

Return \*this

CAS(ctr, nullptr) == true

Pred\_ctr->release()

Return \*this

Pred\_ctr->release()

Return \*this

CAS(ctr, nullptr) == false

Other\_ctr->release()

1. Pred\_ctr에 ctr을 other\_ctr에 other.add\_ref\_copy()의 return값을 store한다.
2. Other\_ctr을 순서대로 검사한다.

2.1) other\_ctr == nullptr 경우, ctr을 nullptr로 CAS()하고 실패하면 처음부터 다시시도, 성공하면 기존의 pred\_ctr을 release()한 뒤 return \*this.

2.2) other\_ctr == pred\_ctr 경우, add\_ref\_copy()를 통해 other\_ctr의 use\_count를 1증가시켰기 때문에 other\_ctr(pred\_ctr)을 다시 release()한 뒤 return \*this. (pred\_ctr, other\_ctr에서 이미 use\_count를 증가시켰기 때문에 add\_ref\_copy()에서 올린 use\_count를 내린다.)

2.3) 위의 두 순서를 지나면, other\_ctr은 nullptr이 아니며 pred\_ctr과 다른 ctr이다.

CAS(ctr,pred\_ctr,other\_ctr)를 시도해, 성공할 경우 pred\_ctr을 release()한 뒤 return \*this.

2.4) CAS(ctr,pred\_ctr,other\_ctr)가 실패할 경우, 2.2)와 동일하게 other\_ctr을 다시 release()하고, ctr이 변경되었으므로 처음부터 다시한다.

shared\_ptr& operator=(shared\_ptr& other)

{

// 지역변수 pred\_ctr, other\_ctr

control\_block<Tp>\* pred\_ctr;

control\_block<Tp>\* other\_ctr;

while (true)

{

// 반복문이 돌때마다 other\_ctr은 other의 use\_count를 1씩 증가시킨다.

// -> 반복할 경우, 증가시킨 other의 use\_count를 감소시켜야한다.

pred\_ctr = ctr;

other\_ctr = other.add\_ref\_copy();

if (other\_ctr == nullptr)

{

// 1) CAS(ctr,…,nullptr) 성공할 경우, 기존의 ctr(pred\_ctr)을 release()한다.

if (true == CAS(&ctr, pred\_ctr, nullptr))

{

if(pred\_ctr != nullptr)

pred\_ctr->release();

return \*this;

}

// 1.1) CAS() 실패할 경우, 반복문을 반복한다.

// -> add\_ref\_copy()의 return이 nullptr이면, use\_count증가는 없다.

else

continue;

}

if (other\_ctr == pred\_ctr)

{

// 2) other\_ctr과 pred\_ctr이 동일하면, 함수가 불리기 전과 동일한 상태여야 한다.

// -> add\_red\_copy()로 올린 other\_ctr의 use\_count를 다시 내린다.

other\_ctr->release();

return \*this;

}

if (true == CAS(&ctr, pred\_ctr, other\_ctr))

{

// 3) CAS(ctr,…,other\_ctr) 성공한 경우, 기존의 ctr(pred\_ctr)을 release()한다.

if (pred\_ctr != nullptr)

pred\_ctr->release();

return \*this;

}

else

{

// 4) CAS() 실패할 경우, 반복문을 반복한다.

// -> add\_ref\_copy()로 증가한 use\_count를 감소시킨다.

other\_ctr->release();

continue;

}

}

}

|  |  |  |  |
| --- | --- | --- | --- |
| **문제점 정리** |  | **해결 방안** |  |
| **다음 주차** | **25 주차** | **다음 기간** | **2020.07.02 ~ 2020.07.08** |
| **다음주 할 일** |  | | |
| **지도교수**  **Comment** |  | | |