참조 카운팅 스마트 포인터의 속도 보완 방법 연구

박경준* · 서민석** · 박현주***

A Study on Method to Complement Speed of the Reference Counting Smart Pointer

Park Kyung-Joon* · Seo Min-Seok** · Park Hyun-Ju***

이 논문은 한국연구재단(과제번호 : 2012-0090)의 지원을 받아 수행된 연구임

요 약

최근의 C++ 개발자들은 개발 시 메모리 관리에 장점을 지닌 참조 카운팅 방식의 스마트 포인터를 많이 활용한다. 그러나 스마트 포인터는 빠른 응답성을 요구하는 시스템에서는 일반 포인터에 비해 효율적이지 못한 동작 방식 때문에 문제가 생길 수 있다. 본 논문에서는 이러한 점을 보완하여 참조 카운팅 방식 스마트 포인터가 가리키는 객체에 대한 삭제 작업을 수행 시 삭제 작업을 위한 스레드를 생성하여 성능을 개선하는 방법을 제안 하였다. 또한 이를 일반화 시켜서 모든 자료 형에서 동작시키기 편리한 방법을 제안하였다.

ABSTRACT

Recent C++ developers take advantage of the many advantages in memory management in the development of reference counting smart pointer. However, these have a problem more than the normal pointer. In this paper, we not only complement these by reference counting smart pointer which pointing a large dynamic object to perform a delete operation to improve performance by creating a thread for the delete operation but also Generalize all data types, we propose a convenient way to operate.

키워드

Smart Pointer, C++, Heap Memory

Key word

Smart Pointer, C++, Heap Memory

* 준회원 : 한밭대학교 전파공학과 석사(교신저자, atrocite@nate.com)
** 정회원 : 한밭대학교 전파공학과 박사과정
*** 정회원 : 한밭대학교 전파공학과 교수

접수일자 : 2012. 12. 28
심사완료일자 : 2013. 01. 14

Open Access <http://dx.doi.org/10.6109/jkiice.2013.17.4.878>

©This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License(<http://creativecommons.org/licenses/by-nc/3.0/>) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.
Copyright © The Korea Institute of Information and Communication Engineering.

I. 서 론

소프트웨어 기술은 컴퓨터가 발명된 이후로 놀랄만한 속도로 향상되어 오고 있다. 또한 이에 비례하여 소프트웨어 구현의 복잡도 역시 증가 하고 있다. 이러한 어려움에 따른 소프트웨어 개발 생산성, 품질의 저하를 보완하고자 프로그래밍 언어들은 여러 가지 방법들을 이용하여 소프트웨어 개발자들에게 편리한 구현 방법을 제공한다.

그중 C++는 최근에 Java와 C#언어에서 제공하는 가비지 컬렉션 (Garbage Collection)과 상당히 흡사한 형태의 메모리 관리 객체인 스마트 포인터를 다양한 종류로 제공하여 메모리 관리 방법의 편의성을 향상 시켰다.

그러나 스마트 포인터는 메모리관리 측면에서 여러 장점을 가지고 있음에도 불구하고, 동작 방식이 일반 포인터에 비해 효율적이지 못하다는 단점이 있다.

특히 데이터 베이스로부터 대용량의 데이터를 가져와서 여러 Client들에게 전송할 필요가 있는 다중 쓰레드 방식으로 동작하는 서버의 경우 스마트 포인터는 응답 속도 측면에서 취약점을 가질 수 있다[1].

이에 본 논문에서는 현재 가장 널리 쓰이고 있는 참조 카운팅 방식 스마트 포인터 (Reference - Counting Smart Pointer : RCSP)를 빈번하게 사용하는 시스템에서 응답성을 높일 수 있는 방안에 대해 제안하고, 성능을 검증 하기 위한 환경을 구현하여 성능 평가를 실시하였다.

II. 관련 연구

2.1. 스마트 포인터

힙 메모리 (Heap Memory) 또는 동적 메모리라 불리는 메모리는 연산자 new에 의해서 생성된 객체와 변수들을 포함한다. 이 메모리상에 할당된 객체들과 변수들은 이후의 연산자 delete에 대한 호출에 의해 명시적으로 해제될 때까지 존재한다. 이 메모리는 delete 호출이 없으면 프로세스가 종료 되지 않으며 운영체제로 반환되지 않는다. 그 결과 new를 사용해서 할당된 메모리를 해제하는데 실패하는 것은 일반적으로 메모리 누수

(Memory Leak)를 일으킨다. 이 동적 메모리에 의해 할당되는 객체의 주소는 실행시간 (Runtime)에 결정된다. 메모리 누수가 발생하고 점차 증가하게 되면, 해당 프로세스는 성능이 저하되거나 메모리 고갈 등의 장애가 발생할 수 있고, 심할 경우 다른 프로세스가 작동하지 않고 사용자가 문제의 원인조차 찾을 수 없는 상황이 발생하기도 한다[2].

C#과 Java의 경우 언어 차원에서 필요 없어진 동적으로 할당했던 메모리 영역을 자동으로 해제 하는 기능을 가진 가비지 컬렉터가 있기 때문에 메모리 누수에 대해서 특별히 신경 쓸 필요가 없지만, C++는 개발자가 객체를 힙 메모리 영역에 할당할 경우 필수적으로 직접 해제를 해주어야 하기 때문에 프로그램이 복잡해질수록 delete호출 시점에 대해 고민해야 하는 어려움이 생긴다.

스마트 포인터는 이러한 어려움을 손쉽게 해결할 수 있도록 C++ 라이브러리에서 제공하며 힙 메모리 영역에 할당된 객체나 변수들에 대한 delete 호출을 함수가 끝날 경우 자동으로 수행시켜 메모리 누수를 막아주는 역할을 하는 객체이다[3].

대표적인 스마트 포인터는 auto_ptr과 shared_ptr이 있다.

2.2. shared_ptr

shared_ptr은 최근에 표준 C++ 라이브러리에 추가된 TR1에서 제공하는 현재 가장 널리 사용되는 스마트 포인터 이다. 기본적인 동작은 일반 포인터나 auto_ptr과 비슷하며, 특징으로는 동일한 하나의 객체를 가리키는 여러 shared_ptr이 있으면 shared_ptr은 가리키고 있는 shared_ptr의 숫자를 내부적으로 저장한다. 이러한 기법을 참조 카운팅 (Reference counting)이라 부른다. 어떤 객체를 가리키는 최후의 스마트 포인터가 소멸될 때, 즉 참조 카운트가 0이 될 때, 그 객체도 자동으로 삭제가 되는 방식으로 동작 한다. 그러므로 복사나 대입연산을 수행하게 되어도 개발자들의 상식 안에서의 ‘정상적인’ 복사 동작을 수행한다. 본 논문에서는 shared_ptr을 이용한 메모리 관리 방법을 제안한다[4][5].

III. 본 론

3.1. 제안하는 deleter_thread 설계

shared_ptr은 삭제자(deleter) 지정을 허용한다. 여기서 삭제자란 shared_ptr이 유지하고 있는 참조 카운트가 0이 되었을 때 바로 동적 메모리 영역을 삭제하는 대신 개발자가 자신의 의도에 맞게 삭제할 수 있도록 동작하는 함수 혹은 함수 객체를 말한다. 삭제자는 shared_ptr 생성자의 두 번째 매개변수로 선택적으로 넣어 줄 수 있다.

deleter_thread방식은 shared_ptr의 삭제자에서 쓰레드를 새로 생성하여 그 안에서 실제 객체를 삭제하는 과정으로 동작한다. 필요가 없어져 삭제하고자 하는 shared_ptr로 가리키고 있는 객체는 삭제자에서 생성되는 쓰레드로 넘겨져 삭제가 진행되고, 기존의 흐름에는 영향을 주지 않는 방식으로 동작한다. deleter_thread방식에서 쓰레드를 생성해서 처리하는 가장 큰 이유는 빠른 응답성을 보장할 수 있다는 점이다. 네트워크 서버의 경우 Client들에게 제공하는 빠른 응답성은 가장 중요한 부분이다. 만약 shared_ptr로 가리키고 있는 객체가 모든 작업을 마치고 같은 쓰레드 흐름에서 삭제가 된다면 삭제에 걸리는 시간 때문에 나머지 작업에서 지연을 유발하여 원래의 속도로 응답할 수 없는 경우가 생길 수 있다.

그림 1. 스마트 포인터의 일반적인 동작 방식
Fig. 1 The general behavior of smart pointer

이 부분을 다중 쓰레드화 하여 삭제작업을 다른 쓰레드에서 처리하게 만들면 Client에 대한 응답성을 보장할 수 있어 신뢰성을 높일 수 있다. 또한 멀티 프로세서에서 처리를 한다면 각각의 쓰레드를 다른 프로세서에서 병렬로 수행할 수 있으므로 속도적인 측면에서 이점이 배가된다.

그림 2. deleter_thread의 동작 방식
Fig. 2 The behavior of the deleter_thread

deleter_thread을 이용하기 위한 필요조건은 세 가지로 정리 할 수 있다.

첫째, shared_ptr를 이용할 수 있는 환경이다. deleter_thread 스마트 포인터 중 참조 카운팅 방식으로 동작하는 shared_ptr가 삭제자를 허용한다는 특징을 이용한다.

둘째, 복잡한 구성을 가진 객체들을 많이 포함하고 있거나, 대용량의 데이터를 담고 있는 객체를 다룬다. 스마트 포인터를 통해 이러한 객체를 조작 할 때 속도저하가 두드러지므로, 제안하는 방법을 이용하면 효과를 극대화 할 수 있다.

셋째, 쓰레드를 생성할 수 있어야 한다. 제안하는 방법은 shared_ptr의 카운팅이 0이 되었을 때 쓰레드를 생성하여 대용량의 객체를 삭제하는 방식이다. 그러므로 쓰레드를 생성할 수 있는 환경이어야 한다.

또한 이와 같은 설계를 위해서는 다음 두 가지를 고려한다.

먼저 쓰레드의 소멸 시점에서의 동작 설정이다. 삭제자에서 생성되는 쓰레드는 reference count가 0이 되어 필

요가 없어진 동적 객체를 삭제하는 작업만 수행하므로, 종료 시 `main` 스레드에서 이 스레드의 흐름이 종료될 때까지 기다릴 필요가 없다.

그다음으로 스레드 동기화 (Synchronization) 문제이다. 일반적으로 스레드 동기화가 필요한 상황은 크게 다음 두 가지 측면으로 나뉜다.

- 동일한 메모리 영역으로부터 동시접근이 발생하는 상황
- 동일한 메모리 영역에 접근하는 스레드의 실행 순서를 지정해야 하는 상황

앞에서 설명한 대로 `deleter_thread`는 참조 카운트가 0이 된 참조하지 않는 메모리 영역의 객체를 해제하는 역할을 수행한다. 다른 스레드에서 해당 객체를 참조할 이유가 없으므로 동시 접근 상황을 고려할 필요가 없다. 또한 스레드의 실행 순서 역시 독자적으로 수행 후 종료되는 방법을 따르기 때문에 지정할 필요가 없다. 그러므로 스레드 동기화에 대한 고민은 필요하지 않다는 결론을 얻을 수 있다.

그러므로 리눅스에서는 `pthread_join` 함수를 호출하지 않고, `pthread_detach` 함수를 호출한다. 마찬가지로 이유로 윈도우에도 스레드를 생성과 종료시 커널 오브젝트 관련 함수를 이용하지 않는다.

여기에 더해 `deleter_thread`는 C++에서 제공하는 템플릿을 이용하여 모든 자료형에서 하나의 `deleter_thread` 모듈로 기능을 이용할 수 있도록 일반화 프로그래밍 기법을 적용하였다.

3.2. deleter_thread 구현

`deleter_thread`는 리눅스에서 구현하였다. 그리고 구현 순서는 총 세 단계로 나뉜다. `shared_ptr`의 선언 부분을 구현하고, 삭제자를 구현하고, 마지막으로 스레드에서 동작할 함수를 구현하였다.

선언부는 [그림 3]과 같다.

```
tr1::shared_ptr<P_Dataset> ptr ( new P_Dataset(), deleter_thread<P_Dataset>() );
```

그림 3. deleter_thread의 선언 부분
Fig. 3 Declared part of deleter_thread

삭제자를 `shared_ptr` 생성자의 두 번째 매개변수로 넣어준다. 그리고 삭제자의 이름 뒤에 `<>` 연산자 안에 동적으로 생성할 객체의 자료형을 명시적으로 선언해준다.

삭제자 구현은 [그림 4]와 같다. 삭제자 안의 `operator()` 연산자의 매개변수를 `T*` 자료형으로 넣어준다. 지정한 삭제자 안에서 `pthread_create` 함수로 스레드를 생성하고, 세 번째 매개변수로 스레드 안에서 수행될 동작을 설정한 함수 명을 넣는다. 그리고 함수명뒤의 `<>` 연산자에 `T` 자료형을 입력하여 선언부에서 입력한 객체의 자료형이 그대로 연결될 수 있도록 한다. 네 번째 매개변수로 는 삭제할 동적 객체의 포인터주소를 `void*` 형태로 형 변환하여 지정한다.

```
template <typename T>
struct deleter_thread{

    void operator()(T* ptr)
    {
        pthread_t t_id;

        pthread_create(&t_id, NULL, Cthread<T>, static_cast<void*>(ptr) );
        pthread_detach(t_id);
    }
};
```

그림 4. deleter_thread의 삭제자 연산
Fig. 4 Deleter operation of deleter_thread

마지막으로 스레드에서 동작할 함수 구현 부는 [그림 5]와 같다. `void*` arg를 다시 원래의 자료형으로 형 변환하여 해당 포인터를 통해 동적 객체를 삭제한다.

```
template <typename T>
void*
Cthread(void* arg)
{
    T* ptr = (T*) arg;

    delete ptr;

    return (void* ) 0;
}
```

그림 5. 스레드의 동작 구현
Fig. 5 Behavior implementation of thread

IV. 성능 평가

본 장에서는 실제 4개의 쓰레드로 동작하는 서버 시스템에 `deleter_thread`를 적용하여 성능을 검증하였다.

4.1. 성능 검증 방법

본 절의 성능평가에 이용되는 서버S/W의 주요 동작은 다음과 같다.

- ① 사용자의 명령이 있을 때 데이터베이스에서 데이터를 얻어 온다.
- ② 데이터베이스로부터 얻어온 데이터는 데이터를 적절히 담을 수 있도록 설계한 `Dataset` 객체에 저장되며, 이 객체를 스마트 포인터가 가리키게 된다.
- ③ `Dataset` 객체의 데이터를 패킷에 담을 수 있는 크기로 나누어서 전송한다.
- ④ `Dataset` 객체의 마지막 데이터를 패킷으로 가공 후 전송이 완료되면, `Dataset` 객체는 더 이상 필요가 없으므로 삭제 한다.

본 절의 성능검증 조건에서는 서버에서 필요한 데이터를 데이터베이스로부터 49,998개를 요구하여 객체에 담고, 이것을 1,023개의 패킷으로 나누어서 Gateway로 전송을 한다. 데이터베이스에 저장된 49,998개의 데이터 용량은 약 5.15MB이다.

그림 6. 멀티 쓰레드 서버 S/W에서의 성능 검증 과정

Fig. 6 Multi-threaded server S / W in the performance of the verification process

그리고 마지막 패킷을 가공하여 전송하고 객체를 삭제하여 하나의 시퀀스를 마무리하는 시간을 일반적인 스마트 포인터를 사용 할 때와 `deleter_thread`를 적용하였을 때로 나누어서 측정하여 비교해 본다. [그림 6]은 이러한 과정을 도식화한 그림이다.

4.2. 성능 검증 결과

성능 검증 결과는 [표 1], [표 2]와 같다. 최종 루틴 처리 시간은 마지막 패킷을 가공하여 전송하고 스마트 포인터가 가리키고 있는 데이터를 담고 있는 객체를 삭제하는 시간을 의미하고, 총 처리시간은 1,023개의 패킷을 전송하는 모든 시간을 의미한다.

표 1. 실험결과 : 일반스마트 포인터의 성능 검증 결과 (단위 ms)

Table. 1 Performance verification results of a general smart pointer

	일반 스마트 포인터	
	최종 루틴 처리 시간	총 처리 시간
1	300	46,886
2	325	46,839
3	292	47,021
4	307	45,346
5	297	47,926
평균	304.2	46,803.6

표 2. 실험결과 : `deleter_thread`를 적용 했을 때의 성능 검증 결과 (단위 ms)

Table. 2 Performance verification results when applying the `deleter_thread`

	<code>deleter_thread</code> 적용	
	최종 루틴 처리 시간	총 처리 시간
1	23	45,039
2	24	45,103
3	25	45,731
4	23	45,051
5	24	45,662
평균	23.8	45,317

[표 1]와 [표 2]를 비교한 결과로 볼 때 `deleter_thread`를 적용하였을 때와 일반 스마트 포인터의 최종 루틴 처리 시간의 차이가 스마트 포인터의 삭제시의 처리 시간으로 볼 수 있고, 평균 270ms의 시간이 소요되는 것으로 나타났다.

결론적으로 본 성능검증에 이용되는 서버 S/W에서 `deleter_thread`방식을 이용하게 되면 해당 루틴에서 평균 270ms의 시간동안의 응답 딜레이를 방지 할 수 있으며, 더 큰 대용량의 데이터를 다루며 빈번하게 이러한 루틴을 이용하는 시스템에서는 그에 비례하는 성능향상을 이룰 수 있다고 볼 수 있다.

V. 결 론

본 논문에서는 참조 카운팅 방식 스마트 포인터가 가리키는 대용량 객체에 대한 삭제 작업을 수행 시 삭제 작업을 위한 쓰레드를 생성하여 성능을 개선하는 방법을 제안하였고, 또한 이를 일반화 시켜서 모든 자료 형에서 동작시키기 편리한 방법을 제시하였다.

본 논문에서 제안한 방법을 이용하면, 스마트 포인터를 더욱 유용하게 사용할 수 있을 것이라 기대된다. 특히 대용량의 데이터베이스와 연동되어 동작하는 다중쓰레드 환경의 서버 시스템 등에서 성능상의 장점을 줄 수 있을 것이다.

참고문헌

- [1] 유서희, 강동현, 이권용, 박성용, “학습기반의 동적 쓰레드 풀 기법을 적용한 웹 서버의 설계 및 구현”, 한국 정보과학회 논문지, 제16권, 제1호, pp.23-34, 2010.1.
- [2] 임우섭, 한환수, 이상원, “동적 순환 메모리 할당 기법을 이용한 메모리 누수 검출”, 정보과학회논문지 : 소프트웨어 및 응용, 제37권, 제10호, pp.760-767, 2010.10
- [3] ANDERI ALEXANDRESCU, “Modern C++ Design”, INFOBOOK, pp.262-282, 2003.

- [4] Scott Meyers, “Effective C++”, PEARSON, pp.121, 2006
- [5] A Proposal to Add General Purpose Smart Pointers to the Library Technical Report, <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1450.html>.
- [6] 윤성우, “C++ 프로그래밍”, FREELEC, pp.451, 2007.
- [7] 공동환, “뇌를 자극하는 C++ STL”, 한빛미디어, pp.91-123, 2012.
- [8] 윤성우, “열혈 TCP/IP 소켓 프로그래밍”, ORANGE MEDIA, pp.392-395, 2011.

저자소개

박경준(Kyungjoon Park)

2011 한밭대학교
전과공학과 졸업 (학사)
2011~현재 한밭대학교
정보통신전문대학원
전과공학과 재학 (석사과정)

※관심분야: 무선 통신 소프트웨어, 실내 위치기반 서비스

서민석(Minseok Seo)

2005 한밭대학교
정보통신전문대학원
전과공학과 졸업 (석사)
2012 한밭대학교
정보통신전문대학원
전과공학과 재학 (박사과정)

※관심분야: 무선 통신 소프트웨어, 임베디드 소프트웨어

박현주(Hyunju Park)

1990년 서울 시립대학교
전산통계학과 학사
1992년 서울대학교 대학원
전산과학과 석사

1997년 서울대학교 대학원 전산과학과 박사

1997년~현재 한밭대학교 전파공학과 교수

※ 관심분야: 데이터베이스, 무선 통신 소프트웨어