KIPS Tr. Comp. and Comm. Sys.
Vol.4, No.9 pp.327~336 pISSN: 2287-5891

http://dx.doi.org/10.3745/KTCCS.2015.4.9.327

Fixed Size Memory Pool Management Method for Mobile Game Servers

Seyoung Park† · Jongsun Choi†† · Jaeyoung Choi††† · Eunhoe Kim††††

ABSTRACT

Mobile game servers usually execute frequent dynamic memory allocation for generating the buffers that deal with clients requests. It causes to deteriorate the performance of game servers since it increases system workload and memory fragmentation. In this paper, we propose fixed-sized memory pool management method. Memory pool for the proposed method has a sequential memory structure based on circular linked list data structure. It solves memory fragmentation problem and saves time for searching the memory blocks which are required for memory allocation and deallocation. We showed the efficiency of the proposed method by evaluating the performance of dynamic memory allocation, through the proposed method and the memory pool management method based on boost open source library.

Keywords : Circular Linked List, Fixed Size Memory Pool, Mobile Game Server, Sequential Memory Allocation

모바일 게임 서버를 위한 고정크기 메모리 풀 관리 방법

박 세 영† · 최 종 선†† · 최 재 영††† · 김 은 회††††

요 약

모바일 환경에서의 게임 서버는 클라이언트의 요청을 처리하는 버퍼를 생성하기 위해 일반적으로 동적 메모리 할당을 빈번하게 수행한다. 이는 시스템에 부하를 가중시키고 메모리 단편화를 발생시키게 되어 게임 서버의 성능을 저하시킨다. 본 논문에서는 이러한 문제를 해결하기 위해 고정크기 메모리 풀 관리 방법을 제안한다. 제안하는 방법에서의 메모리 풀은 원형 연결 리스트 형태의 순차적 메모리 구조를 가지며, 이를 통해 게임 서버에서의 메모리 단편화 문제를 해결하고, 메모리 할당과 해제를 위해 필요한 메모리 블록의 탐색 시간 비용을 줄일 수 있다. 실험에서는 제안하는 방법과 잘 알려진 오픈소스 메모리 풀 라이브러리(boost) 기반의 메모리 풀 관리방법을 이용하여, 동적 할당을 수행할 때의 성능평가를 통해 해당 기법의 효율성을 보이도록 한다.

키워드 : 원형 연결 리스트, 고정크기 메모리 풀, 모바일 게임 서버, 메모리 동적 할당

1. 서 론

모바일 환경에서의 게임 서버는 다수의 클라이언트들을 제어하고 서비스의 가용성, 보안성 및 속도를 보장해야 한다. 이를 위해 다수의 서버 시스템을 두고 역할을 분담하여 처리하도록 한다. 사용자 수의 증가와 사용자 간의 잦은 상호작용들은 게임 서버의 부하를 증가시키기 때문에, 분산 게임 서버 기술은 서버시스템의 부담을 줄이기 위해 더욱 중요하다. 분산 게임 서버 제작에 자주 사용되는 기술은 기능, 데이터, 공간에 따른 분산처리가 있다. 기능에 따른 서버의 분산은 게임 서버에서 클라이언트의 요청을 처리하는 데 효율성과 확장성을 높이기 위해서 기능을 중심으로 분산서버를 구성하는 방식이다[1].

데이터에 따른 분산 처리는 사용자가 증가함과 동시에 늘어나는 작업부하가 많아지는 서버를 분산 처리하기 위한 방식이다. 늘어나는 작업부하는 사용자 수에 비례하기 때문에 데이터 분산 기술을 사용하여 게임 사용자들을 효과적으로 분산해야 한다. 데이터 분산의 대표적인 예로는 영역 분할 서버 모델(Zone-based Server Model)과 심리스 서버 모델(Seamless Server Model)이 있다[2]. 공간에 따른 서버의 분산은 서비스가 가능한 게임서버를 다수 설치하여 수용능력을 높이고 사용자의 밀집현상을 제거할 수 있다. 이는 온라인 게임에서 다수의 게임 채널 또는 다수의 서버를 제공하는 방법으로 활용된다[1]. 앞서 설명한 분산서버 구성 방식 중에서 Fig. 1과 같이 기능에 따른 서버분산 방법을 사용하여 게임 서버를 구성할 수 있다[1].

※ 본 논문은 한국 산업통산자원부의 로봇산업융합핵심기술사업 프로그램 (No. 10048474)의 지원으로 수행되었음.
† 준 회 원 : 숭실대학교 정보보안학과 석사과정
†† 정 회 원 : 숭실대학교 컴퓨터학부 조교수
††† 종신회원 : 숭실대학교 컴퓨터학부 교수
†††† 정 회 원 : 서일대학교 인터넷정보과 조교수
Manuscript Received : August 3, 2015
First Revision : September 1, 2015
Accepted : September 1, 2015
* Corresponding Author : Jaeyoung Choi(choi@ssu.ac.kr)

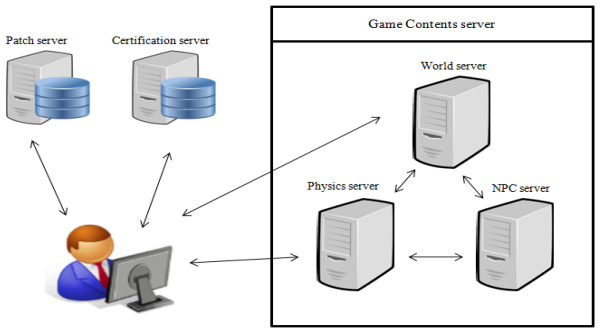


Fig. 1. Server Configuration by Server Function

이는 보편적으로 많이 사용하는 분산게임서버의 구조 중 하나이며, 패치, 인증, 게임 콘텐츠별로 기능에 따라 분산한 것이다. 만약 분산서버 구조가 바뀐다 하더라도, 각 서버는 변함없이 자신의 고유 역할을 수행한다. 이렇게 분산게임서버를 구축하는 것 이외에도 게임서버의 성능을 향상시키기 위해 다양한 연구들이 진행되었다. 그 예로 다양한 타입의 게임서버를 수용하기 위한 다목적 서버의 설계[10], 서버 간 전송되는 데이터의 효율적인 처리를 위한 서버 설계[11], 서버 푸시를 위한 이벤트 기반 서버 간 메시지 교환 아키텍처의 설계 및 구현[14], 가상화 기법을 사용한 경계 없는 캐주얼 게임서버 설계 및 구현[15] 등의 연구가 진행되었다. 이처럼 게임서버의 성능을 향상시키기 위해서 고려할 수 있는 요인은 다양하게 존재한다.

본 논문에서는 서버엔진을 설계할 때 생기는 성능 저하 요인을 개선하여 게임서버의 성능을 향상시키려 한다. 서버엔진 설계 시 성능 향상을 위해 고려할 사항으로는 서버와 클라이언트 간의 통신을 위한 소켓 이벤트 핸들링, 서버에서 데이터 처리를 위한 메모리 할당 문제, 데이터베이스 접속 방식 문제, IOCP[12] 방식을 사용할 때 패킷처리 문제 등이 있다[3]. 그중 메모리 할당에 대한 문제는 자주 메모리 할당, 해제를 수행하는 게임서버에서 성능을 위해 반드시 고려되어야 할 사항이다. 그 이유는 다음과 같다.

게임서버는 클라이언트와의 통신을 위해 소켓, 송수신되는 사용자의 데이터를 저장할 버퍼를 필요로 한다. 인증서버 같은 경우에 Fig. 2와 같이 클라이언트의 접속 요청마다 소켓과 버퍼를 생성한다. 그리고 동적 메모리 할당을 이용하여 확보한 버퍼를 이용해 사용자의 요청을 처리한다. 동적 메모리 할당은 시스템 호출을 동반하는 작업이다[3]. 따라서 동적 메모리 할당과 해제가 자주 발생하게 되는 인증서버에서는 매번 시스템 호출이 일어나게 된다. 그에 따라 시스템 부하는 물론, 메모리 단편화도 발생하여 게임서버의 성능 저하를 초래한다[2].

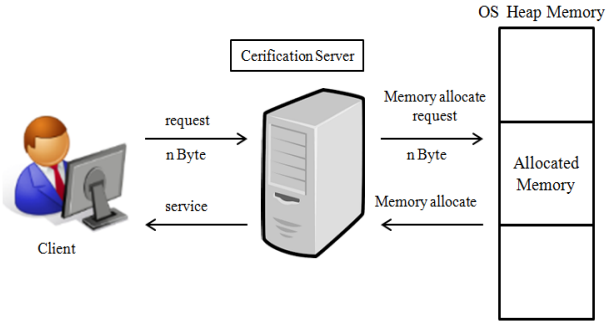


Fig. 2. Dynamic Allocation of Buffer

메모리 단편화는 메모리를 할당하고 해제하는 과정이 반복되면서 사용하지 않고 남은 빈 공간 조각들이 생기는 것을 의미한다. 빈 메모리 공간이 계속 생겨나면서 메모리 단편화가 발생하고, 결국 메모리를 할당받을 수 없거나 원치 않는 메모리 낭비 현상이 발생한다. 이러한 문제점을 해결하기 위해 페이징, 세그멘테이션, 메모리 풀, 메모리 압축 등 다양한 기법들이 존재한다[5]. 따라서 본 논문에서는 메모리 풀을 개선하여 앞서 제시한 문제들을 해결할 것이다. 문제 해결을 위해 본 논문에서는 원형 연결 리스트 형태로 메모리를 관리하는 메모리 풀 관리 방법을 제안한다.

본 논문은 다음과 같이 구성되었다. 2절에서는 배경 지식 및 관련 연구로서 일반적인 메모리 풀과 오픈소스로 잘 알려진 Boost 라이브러리 기반의 메모리 풀 관리 방법, PJLIB 라이브러리를 이용한 메모리 풀(Fast Memory Pool), 그리고 Loki 메모리 할당자 등을 소개 및 분석한다. 3절에서는 본 논문에서 제안하는 메모리 풀 관리 방법에 대하여 언급한다. 4절에서는 성능을 측정한 실험 결과를 보이고, 5절에서는 결론 및 향후 연구에 대해 기술한다.

2. 관련 연구

2.1 메모리 풀

메모리 풀은 Fig. 3과 같이 같은 크기의 메모리 블록들을 필요한 만큼 미리 할당해놓고 나중에 필요에 따라 꺼내어 사용하는 방법을 말한다[6].

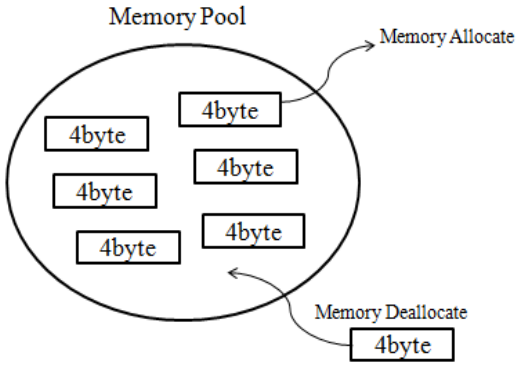


Fig. 3. Memory Pool

메모리 풀을 활용하여 얻을 수 있는 이점은 다음과 같다. 첫째, C/C++ 언어를 바탕으로 개발되는 애플리케이션들은 동적 메모리 할당을 위해 시스템 호출을 사용해야만 한다. 잦은 시스템 호출은 해당 애플리케이션에 큰 부하를 주게

되어 전체적인 성능을 저하시킨다[4]. 하지만 메모리 풀을 이용하면 미리 할당해놓은 메모리를 가져와 사용하고 사용이 끝난 후에 반납하기만 하면 되기 때문에 시스템 호출에 대한 부하가 생기지 않아 충분한 성능을 보장받을 수 있다[16]. 둘째, 메모리 동적 할당과 해제가 잦을수록 단편화 현상이 발생할 확률이 높다. 단편화가 발생하면 메모리가 여러 곳에 흩어져 있게 된다. 따라서 메모리를 블록 단위로 나누어서 처리하는 작업이 많아지게 되고 결국 시스템 부하가 증가하게 된다. 하지만 메모리 풀을 이용하게 되면 처음에 할당받은 큰 메모리 블록 내에서 메모리를 할당, 해제하기 때문에 메모리 단편화를 최소화할 수 있다[6]. 셋째, 실제로 메모리를 할당할 때에는 요구한 크기보다 더 큰 크기로 메모리를 할당하게 된다. 왜냐하면 할당한 메모리에 대한 정보들을 함께 보관해야 하기 때문이다. 이는 메모리를 요구한 크기보다 더 많이 사용하게 되고, 예상보다 빠르게 메모리 고갈을 초래할 수 있다. 나아가 운영체제의 페이징, 프레임 기법에 영향을 미칠 수도 있다[7]. 메모리 풀은 앞서 설명한 것과 같이 처음에 할당받은 큰 메모리 블록 내에서 할당과 해제를 반복하기 때문에 해당 문제를 최소화할 수 있다.

2.2 Boost 메모리 풀

Boost 라이브러리[8]는 C++ 개발자들이 개발하고 있는 오픈소스 라이브러리의 집합이다. 특히 C++언어를 기반으로 제작되는 애플리케이션들을 위해 멀티 스레딩(Multi Threading), 정규 표현식(Regular Expression), 의사 난수 발생(Random Number Generation), 선형대수(Linear Algebra) 등의 다양한 라이브러리를 제공한다. Boost 라이브러리에서는 메모리 풀 라이브러리도 제공한다. 이 메모리 풀은 메모리의 할당과 해제를 효율적으로 처리하기 위해 개발되고 있다. Boost 메모리 풀은 단순분리 저장소(Simple Segregated Storage)라는 메모리 블록 형태를 이용한다. 단순분리 저장소는 할당받은 메모리 블록을 청크(Chunk)로 분할한 메모리 블록의 형태이다. 여기서 청크(Chunk)는 메모리 블록을 고정크기로 나누었을 때, 나누어진 한 부분을 의미한다. 이러한 단순분리 저장소의 구조는 메모리의 할당과 해제가 빠르게 일어나도록 한다.

Fig. 4(a)는 Boost 메모리 풀의 단순분리 저장소의 구조를 보여준다. 단순분리 저장소의 청크들은 항상 같은 크기를 가져야만 하는 제약사항을 가진다. 단순분리 저장소는 Fig. 4(b)와 같이 각 청크들을 연결 리스트를 이용하여 관리한다.

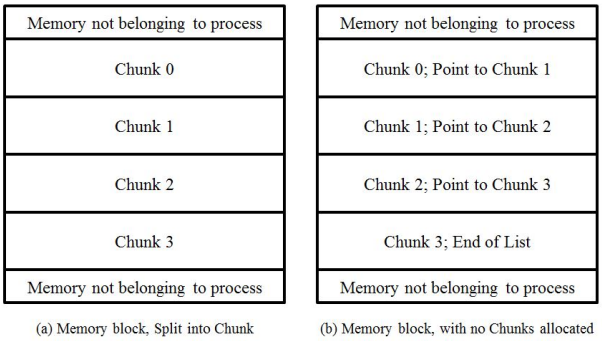


Fig. 4. Simple Segregated Storage Boost in Memory Pool

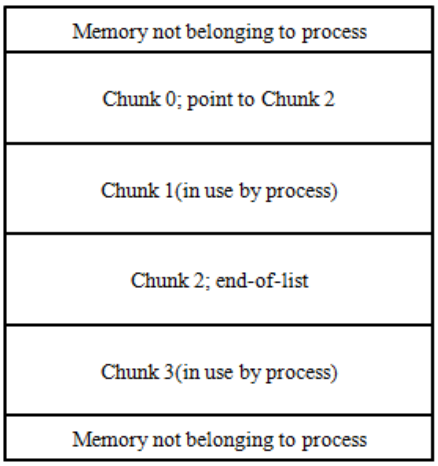


Fig. 5. Memory Allocation in Boost Memory Pool

Fig. 5는 네 개의 청크 중에 할당된 청크가 2개 있을 경우의 단순분리 저장소를 보여준다. 할당된 청크는 연결 리스트에서 제거되고 할당이 가능한 청크들끼리 연결 리스트로 연결되어있다. Boost 메모리 풀에서 메모리의 할당은 사용 가능한 청크들의 연결 리스트로부터 첫 번째 청크를 제거하는 것이다. 이 방법은 O(1)에 메모리를 할당할 수 있게 한다. 만약 연결 리스트가 비어있다면 다른 메모리 블록이 요구되는데, 다른 메모리 블록이 생성되는 것은 O(1)의 시간이 소요된다. 메모리 해제는 연결 리스트에서 제거한 청크를 리스트의 가장 앞부분에 추가하면 된다. 이 방법은 메모리 해제를 O(1)에 처리할 수 있도록 한다. 하지만 연속된 메모리의 할당과 같은 복잡한 단순분리 저장소의 사용은 정렬된 청크 리스트를 요구한다.

연속된 메모리의 할당은 Fig. 6에서처럼 7 Byte의 메모리 할당 요청이 들어왔을 때 연속된 두 개의 청크를 할당해주면 된다. 이때는 각 청크들의 연결 리스트가 연속되는 메모

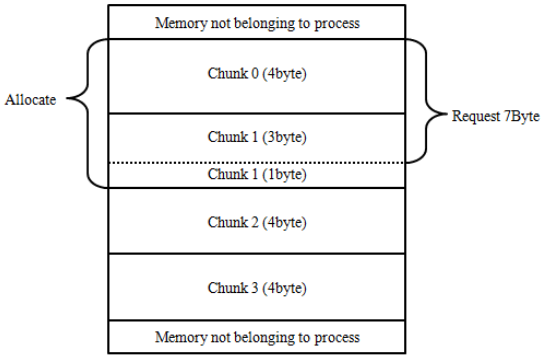


Fig. 6. Sequential Memory Allocation Request

리 순서대로 정렬되어있기 때문에 메모리 할당이 가능한 것이다. 만약 청크들의 연결 리스트가 Chunk 1, Chunk 3, Chunk 2, Chunk 4의 순서대로 정렬되어있는 상태라고 가정하면, 이 연결 리스트는 연속된 메모리 순서대로 청크들이 정렬된 것이 아니기 때문에 메모리를 할당할 수 없게 된다. 그러므로 메모리를 해제할 때는 매번 리스트의 정렬을 수행해야 하는 단점이 있다. 이때는 O(N)에 메모리를 해제하게 된다. 사실 Boost 메모리 풀은 단순분리 저장소 안에 청크들만을 가지는 것이 아니라 메모리 풀의 관리를 위해 추가적으로 메모리를 더 확보한다. 해당 메모리 공간은 Fig. 7에서 보듯이 다음 메모리 블록을 가리키기 위한 포인터와 다음 블록의 크기를 알기 위한 변수를 저장하는 데 사용한다.

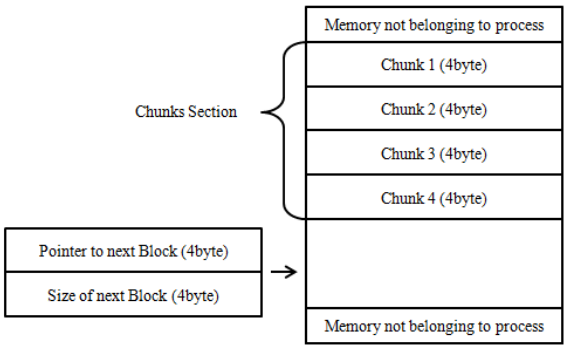


Fig. 7. Memory Block Structure in Boost Memory Pool

2.3 PJLIB 라이브러리를 이용한 메모리 풀(Fast Memory Pool, 이하 패스트 메모리 풀)

PJLIB[9]는 확장 가능한 응용 프로그램을 만들기 위한 C 언어로 작성된 오픈소스 라이브러리이다. PJLIB에서는 패스트 메모리 풀이라는 커스텀 메모리 할당자를 제공한다. 패스트 메모리 풀은 malloc() 또는 new 연산자와 비슷하며 동적 메모리를 할당할 수 있게 도와준다. PJLIB 역시 기본 할당자를 이용했을 때 발생하는 성능 저하, 단편화 현상 등을 해결하였다.

패스트 메모리 풀은 Fig. 8에서 보듯이 풀 팩토리(Pool Factory), 메모리 풀, 메모리 청크를 생성하여 메모리를 관리한다. 이런 메모리 관리 방법은 메모리 단편화 문제를 해결한다. 최초에 할당된 연속된 메모리 블록은 풀 팩토리가 관리하게 된다. 패스트 메모리 풀은 다양한 크기의 메모리를 할당할 수 있게 하는 장점을 가지고 있다. 이는 다른 메모리 풀 방식과 비교하여, 메모리를 할당하는 데 있어 유연함을 가진다. 그리고 패스트 메모리 풀에서 일반적인 메모리 할당은 O(1)의 시간 복잡도를 가지는 동작이며, 몇 회의 포인터 연산만을 필요로 하는 단순한 동작이다. 할당되었던 메모리들은 풀 자체가 제거될 때 모두 반환이 되는 형태로, malloc(), new 연산자를 이용한 할당보다 높은 성능을 나타낸다. 하지만 패스트 메모리 풀의 방식은 메모리 풀을 생성할 때 크기를 미리 지정해야 한다는 단점이 있다. PJLIB는 풀이 생성되는 순간에 지정한 크기로 시스템에서 메모리를 할당하기 때문이다.

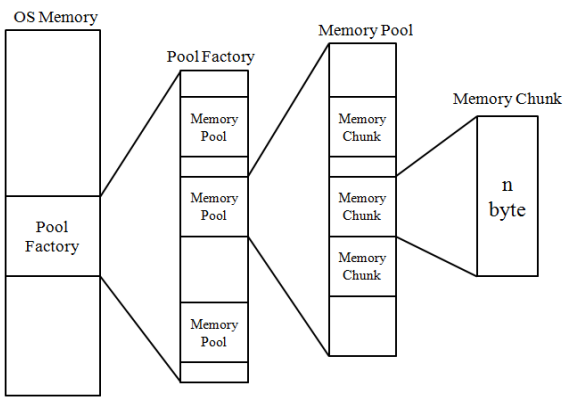


Fig. 8. PJLIB Fast Memory Pool Structure

2.4 Loki 메모리 할당자

Loki 메모리 할당자[7]는 Alexxandrescu가 C++언어를 이용하여 구현한 메모리 풀 방식의 메모리 할당자이다. 작은 크기의 데이터를 자주 다루는 프로그램에서 작은 크기의 객체(object)를 활용하여 데이터를 다루게 되면 프로그램 성능을 향상시킬 수 있다는 사실을 바탕으로 설계되었다.

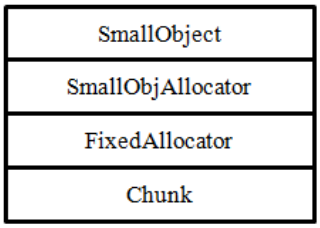


Fig. 9. Layer Structure of Loki SmallObject Class

Loki 메모리 할당자는 Fig. 9와 같이 4계층으로 구성된다. 먼저 1계층인 청크(Chunk)는 메모리 블록 할당 및 반환을 실질적으로 수행하는 클래스(Class)이다. 청크 객체를 생성할 때 메모리 블록의 크기와 개수를 설정할 수 있다. 그리고 생성된 메모리 블록을 실질적으로 관리한다. 청크는 다음 메모리 블록을 가리키기 위한 포인터 연산 대신 1바이트를 이용하여 인덱스를 생성한다. 그리고 이를 통하여 메모리 블록의 정렬을 수행한다. FixedAllocator는 청크를 이용하여 메모리의 할당 및 반환 요청을 수행한다. 먼저 생성된 청크의 메모리 블록이 모두 할당되었을 때 청크를 동적으로 추가 생성한다. FixedAllocator는 고정크기의 메모리 블록을 할당 및 반환하는데, 해당 객체를 생성할 때 메모리 블록 크기를 Chunk를 통해 결정할 수 있어 다양한 크기의 메모리를 할당 및 해제할 수 있는 객체들을 확보할 수 있다. 그리고 메모리 할당 및 해제 시 요구되는 메모리 크기에 알맞은 객체를 이용하여 메모리 할당과 해제를 수행한다. SmallObjAllocator는 각각 다른 크기의 메모리를 할당하는 FixedAllocator 객체들을 가진다. SmallObject는 캡슐화

(encapsulated)된 포장 클래스이다. new와 delete 연산자를 오버로딩 하였고 SmallObjAllocator에 이를 전달하여 메모리 할당 및 해제를 수행한다. 사용자는 실질적으로 해당 클래스를 통해 메모리 할당 및 해제를 요청하게 된다.

FixedAllocator에서는 요청된 메모리 블록 크기를 할당 및 반환해줄 청크를 찾아내는 탐색시간을 줄이기 위해 양방향 선형 탐색을 수행한다. 그리고 탐색 시간에 따른 부담을 줄이기 위해 마지막으로 할당 및 반환이 이루어진 청크의 정보를 캐시(cache) 한다. 그러나 Boost 메모리 풀과 성능을 비교하였을 때는 비교적 낮은 성능을 보인다[13].

3. 개선된 고정크기 메모리 풀 방식

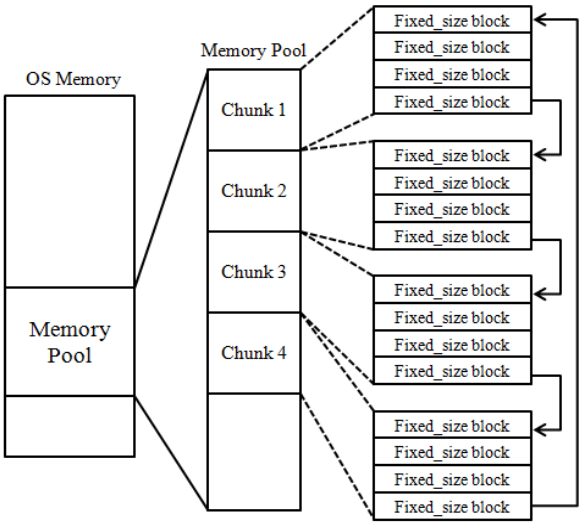


Fig. 10. Memory Pool Structure Based on Circular Linked List

Fig. 10은 본 연구에서 제안하는 메모리 풀의 전체적인 구조를 보여준다. 먼저 시스템으로부터 메모리 풀 역할을 할 큰 메모리 블록을 할당받는다. 그리고 해당 메모리 풀을 같은 크기를 갖는 청크들로 분할한다. 다시 각 청크들은 내부적으로 자신의 메모리 블록을 일정한 크기로 나눈다. 이처럼 작은 크기의 메모리를 자주 다루게 되는 메모리 풀은 작은 크기의 데이터를 다루는 데 있어 높은 성능 향상을 기대할 수 있다[7]. 이러한 메모리 풀 구성 방식은 메모리 풀을 힙 메모리로부터 미리 한 번에 크게 할당하고 이를 나누어 사용하기 때문에 메모리의 단편화가 발생하지 않아 시스템 성능 저하를 최소화할 수 있다. 그리고 메모리 풀 역시 같은 크기의 청크들로 분할하고 사용하기 때문에, 메모리 풀 내에서의 단편화 또한 발생하지 않는다. 그러므로 메모리 풀 자체를 효율적으로 사용할 수 있게 된다.

각 청크들이 갖는 자신의 고정크기 메모리 블록들은 Fig. 10에서 보듯이 원형 연결 리스트로 관리된다. 원형 연결 리스트는 청크들을 순서대로 이동하며 메모리를 할당하고, 이를 순환시키기 위해 논리적으로 구성한 자료 구조의 형태이다.

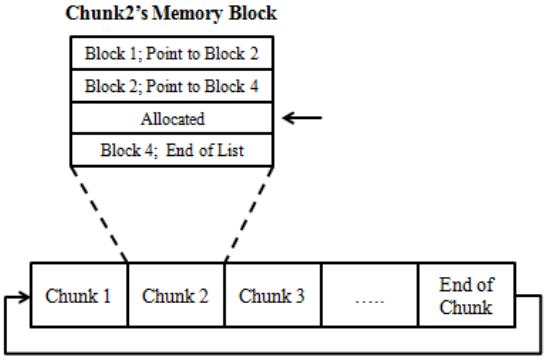


Fig. 11. Logical Circular Linked List Structure Using Chunk Circular Array

각 청크는 자신이 보유한 고정크기 메모리 블록들을 연결 리스트를 이용해 관리한다. 이 연결 리스트들은 서로 다른 청크들에 대해 독립적이다. 여기서 각 청크들은 원형 배열(Circular Array)로 관리되는데, 이 구조를 통해서 해당 메모리 풀이 논리적인 원형 연결 리스트로 구성된다. Fig. 11은 앞서 설명한 메모리 풀의 실제 자료 구조를 보여준다. 따라서 해당 자료 구조를 통해 청크의 배열을 순차적으로 순환하면서 각 청크가 보유한 고정크기 메모리 블록을 할당할 수 있다. 전체 청크를 배열로 관리하지만, 순차적으로 각 청크에 접근하는 것을 원칙으로 한다.

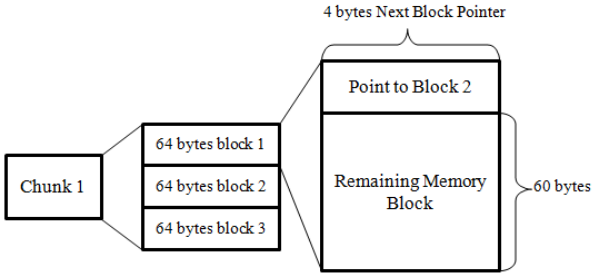


Fig. 12. Memory Block Structure in Chunk

Fig. 12에서 보는 바와 같이, 청크가 소유한 메모리 블록들을 연결 리스트로 관리하기 위해 각 메모리 블록의 상위 4바이트를 이용하여 다음 블록을 가리키는 포인터를 저장한다. 따라서 할당받은 메모리 블록은 반드시 NULL로 초기화하여 사용해야 한다. 메모리 풀의 생성 초기에 청크가 소유한 메모리 블록들은 모두 연결 리스트에 포함되어있다. 그러나 메모리 할당 및 해제가 진행된 후에는 할당이 가능한 메모리 블록만을 대상으로 연결 리스트가 구성된다. 그리고 연속된 공간에 대한 메모리 할당 요청 문제를 해결하기 위해 물리적인 메모리 위치 순서를 기준으로 연결 리스트가 정렬된다.

메모리 할당 순서는 Fig. 13에서 보듯이 첫 번째 청크가 가진 메모리 블록부터 시작되며 (a)~(c)처럼 순차적으로 메모리를 할당한다. 해당 메모리 블록의 끝에 도달하게 되면

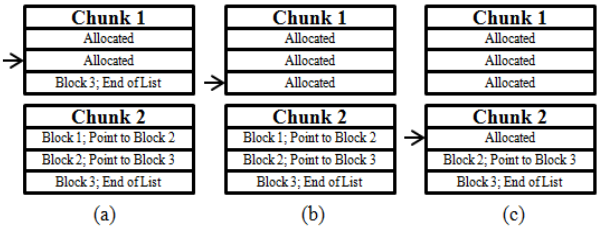


Fig. 13. Sequential Memory Allocation

자신의 청크가 보유한 메모리 블록을 탐색하여 할당 가능한 메모리 블록을 찾는 것이 아니라 다음 청크의 메모리 블록으로 넘어가서 할당한다. 마지막 청크의 모든 메모리 블록까지 할당이 완료되면 다시 첫 번째 청크로 돌아가 새롭게 구성된 연결 리스트를 따라 순차적으로 메모리를 할당한다.

```
class Chunk
{
Public :
   Chunk(std::size_t block_size, unsigned int blocks)
   void *alloc(std::size_t block_size)
   void dealloc(void *dealloc_target, std::size_t block_size)
   …
Private :
   …
   unsigned int block_count; // usable memory block count
}
```

Fig. 14. Setting a Number of Memory Block

각 청크는 Fig. 14에서처럼 할당 가능한 자신의 메모리 블록 숫자를 저장하는 변수를 가진다. 해당 값이 0인 청크는 현재 할당 가능한 메모리 블록이 없기 때문에 다음 청크로 바로 이동하여 메모리를 할당하게 한다. 이 방법은 메모리 할당이 불가능한 청크의 메모리 블록을 불필요하게 탐색하지 않도록 하기 때문에 메모리 할당의 성능을 향상시킨다.

앞서 소개한 해당 메모리 풀의 구조에 따라 각 청크는 메모리 할당 후 다시 자신의 차례가 되면 새롭게 구성된 메모리 블록 연결 리스트를 바탕으로 메모리 할당을 수행한다. 그리고 해당 연결 리스트는 메모리 할당이 가능한 메모리 블록만을 포함하고 있는 연결 리스트이기 때문에, 메모리를 순차적으로 할당받으면 되므로 언제나 O(1)의 시간 복잡도를 갖는다. 그리고 단순히 연결 리스트의 포인터를 이동시키는 연산만이 필요하다.

제안하는 메모리 풀 관리 방법은 메모리 해제과정을 통해 메모리 할당의 성능을 향상시킬 수 있다. 상세한 내용은 Fig. 15와 같다. 메모리 해제는 청크가 가진 메모리 블록 연결 리스트를 구성할 때 연속된 공간에 대한 메모리 할당 요청을 처리하기 위해서 항상 연결 리스트를 정렬한다. 연결 리스트를 정렬하여 새롭게 구성하는 과정은 Fig. 15에서 보는 것과 같이, 현재 구성된 연결 리스트의 각 메모리 블록 주소 값들을 순서대로 비교하여 연결 리스트를 구성한다.

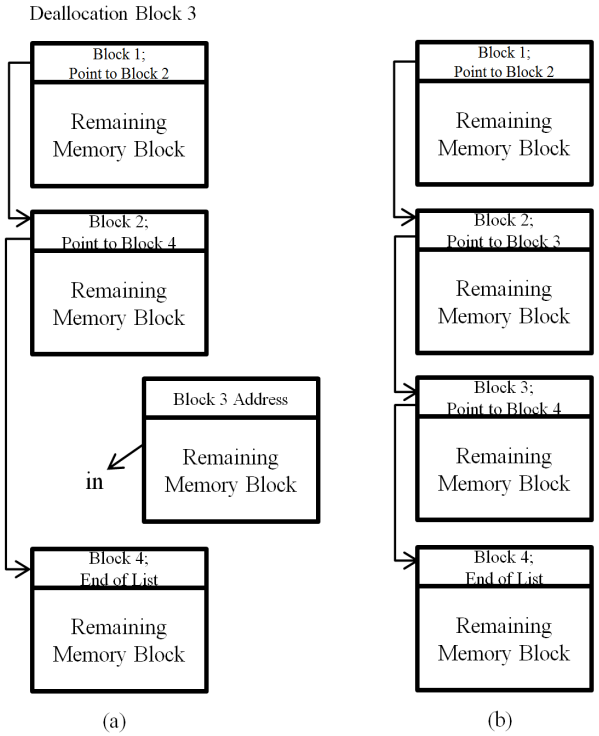


Fig. 15. Memory Deallocation Process

이때 연결 리스트 탐색에 O(N)의 시간 복잡도가 요구되고, 메모리 블록 숫자가 많을수록 메모리를 해제하는 비용은 늘어난다. 따라서 메모리 해제 비용을 줄이기 위해서는 처음 메모리 풀을 생성할 때 청크 내 메모리 블록 수를 청크의 숫자보다 더 적게 생성해야 한다. 예를 들어, 1개의 청크를 100개의 고정크기 메모리 블록으로 나누어 사용하는 것보다, 25개의 청크를 각각 4개의 고정크기 메모리 블록으로 나누어 사용하는 것이 메모리 해제에 필요한 연결 리스트 탐색 시간을 줄어들게 한다. 그 이유는 전자의 경우 연결 리스트의 정렬을 위해 최대 100개의 리스트를 탐색해야 하지만, 후자의 경우 각 청크별로 메모리 블록 연결 리스트가 존재하기 때문에 각각 최대 4개의 리스트만을 탐색하면 되기 때문이다. 후자를 선택했을 때 발생하는 부하는 청크 배열 증가에 대한 추가적인 공간만 필요하고, 배열 특성상 내부 원소에 접근하는 데는 시간 복잡도 O(1) 정도의 비용만이 필요하다.

메모리 할당과 해제에 앞서 메모리 풀을 초기화하기 위해 블록들을 초기화해야 한다.

Fig. 16에서 보는 바와 같이, 초기화 과정은 메모리 풀로 사용할 메모리 블록의 할당(a), 메모리 풀 생성(b), 메모리 풀을 청크로 분할(c), 그리고 마지막으로 각 청크들을 고정크기의 메모리 블록으로 분할(d)하는 순서대로 진행된다.

메모리 할당 및 해제를 위해 초기화 과정이 완료되면, Fig. 17에서 보는 바와 같이 메모리 할당 및 해제를 수행하게 된다. (a)~(d)는 첫 번째 청크가 가진 메모리 블록의 연결 리스트를 순서대로 접근하여 메모리를 할당 및 해제하는 것을 나타낸다. 이때 (c)는 메모리 해제가 되면 연결 리스트

가 다시 재구성되는 것을 의미한다. (e)~(g)는 첫 번째 청크가 가진 마지막 메모리 블록까지 할당이 끝난 후 두 번째 청크의 메모리 블록으로 할당 순서가 넘어가는 과정을 나타낸다. 이후 두 번째 청크의 마지막 메모리 블록까지 할당이 완료되면, 논리적인 원형 연결 리스트 구조이므로 다시 첫 번째 청크의 메모리 블록으로 할당 순서가 넘어간다.

4. 실험 및 성능 측정

4.1 실험 환경

본 실험에서 사용한 메모리 풀은 Visual C++ 12.0을 사용하여 컴파일 하였다. 실험이 진행된 시스템은 64비트 Windows8 OS, Intel(R) Core(TM) i7-4720HQ CPU @ 2.30GHz 프로세서, 16GB 메모리를 탑재하고 있다. 본 연구에서 제안한 방법으로 구성한 메모리 풀을 이용해 고정크기 할당과 해제를 했을 때와 임의크기 할당과 해제를 했을 때 성능을 실험하였으며, 데이터 블록 크기를 64, 128, 256, 512 바이트로 변경하며 각각 진행하였다. 비교 대상은 Boost 라이브러리의 메모리 풀(boost pool)과 C++에서 제공하는 기본 메모리 할당자(default)이다. Boost 라이브러리의 메모리 풀은 다른 오픈소스 라이브러리에서 제공하는 메모리 풀들과 비교하여도 평균적으로 우수한 성능을 갖기 때문에 다른 메모리 풀들을 대표하여 비교대상으로 선택하였다[13]. 실험은 단일 쓰레드 상에서 각 항목별로 메모리의 할당과 해제를 10000번씩 10회 수행시키고 소요된 시간을 측정하였다. 측정 시간 단위는 밀리세컨드(ms) 단위로 설정하였으며 정확한 코드 수행 시간을 측정하기 위해 QueryPerformanceCounter[17]를 이용하였다. 그리고 본 논문에서 제안한 메모리 풀을 의미하는 용어는 fixed size pool로 나타낸다.

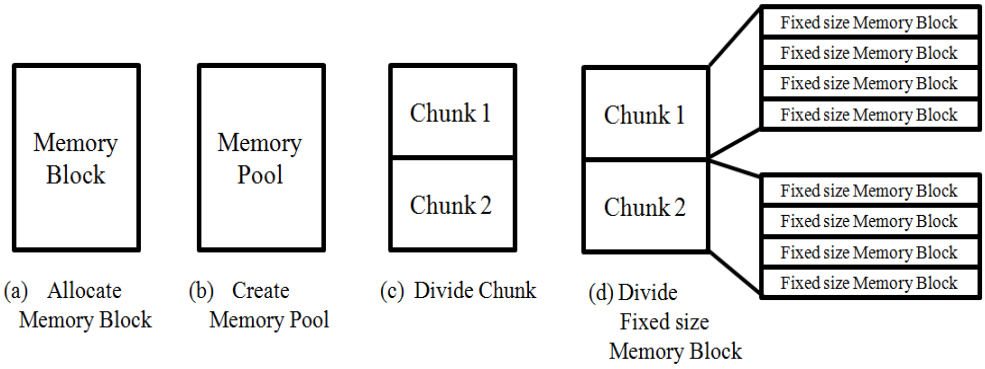


Fig. 16. Step-by-Step Example of Initialization Memory Block

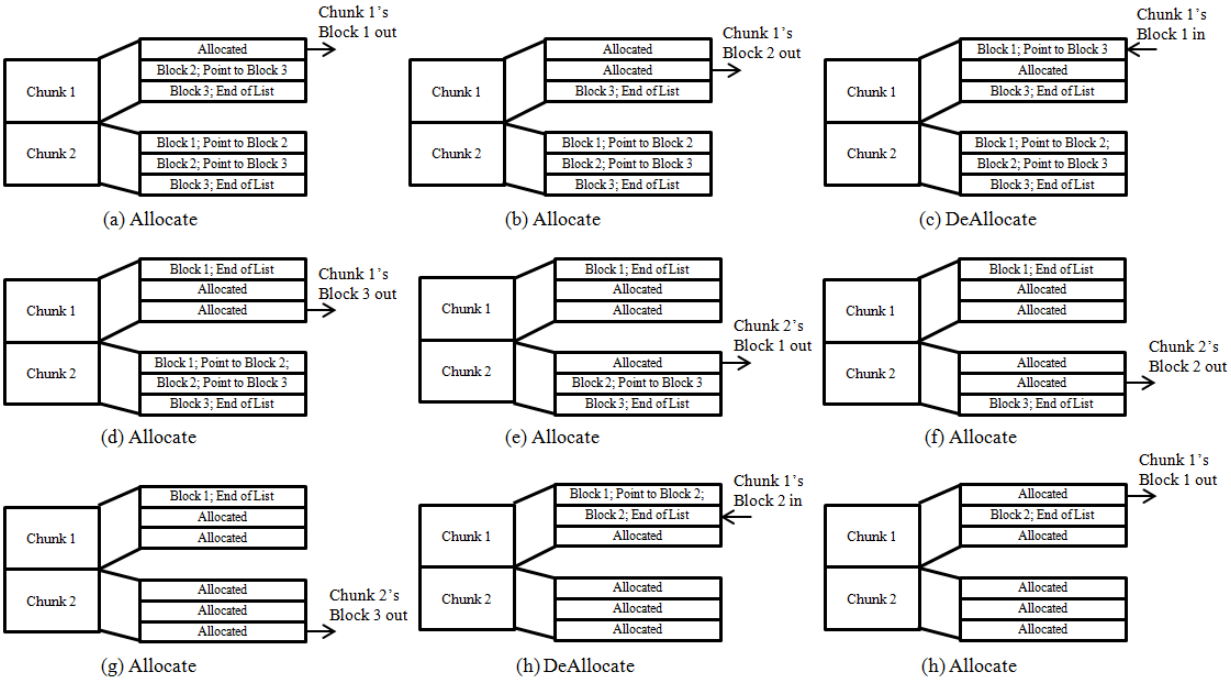


Fig. 17. Step-by-Step Example of Memory Allocation and Deallocation

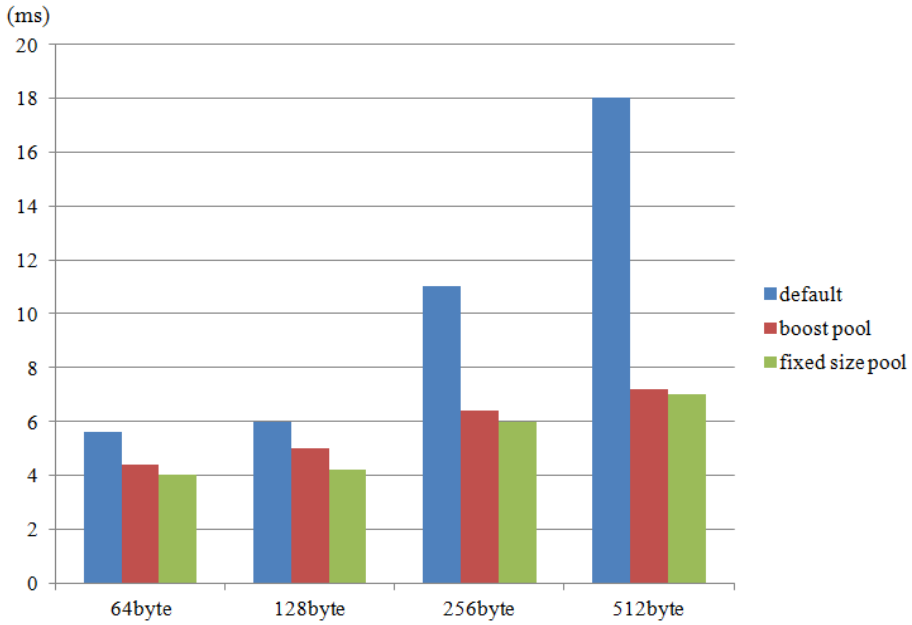


Fig. 18. Performance of Memory Pool in Sequential Memory Allocation and Deallocation

4.2 고정크기 메모리 할당과 해제

고정크기 메모리를 연속해서 10000번 할당한 후에 할당된 메모리를 연속해서 해제한다. 연속적으로 메모리를 할당할 때 소요된 시간과 연속적으로 메모리를 해제할 때 소요된 시간을 더하여 기록하였다. 순차적인 메모리 할당과 해제의 경우, 본 논문에서 제안한 방법으로 구성된 메모리 풀은 작은 크기의 메모리 블록을 할당 및 해제할 때, 다른 메모리 풀을 이용한 할당 방식보다 빠른 성능을 보여준다. 그 이유는 항상 청크 내에 할당 가능한 메모리 블록들을 연결 리스트로 관리하고 있고, 메모리 할당에 간단한 포인터에 대한 연산만이 사용되기 때문이다. 그러나 Fig. 17에서 보듯이 메모리 블록의 크기가 커질수록 성능은 조금씩 떨어지는 모습을 보이지만, 일반적으로 모바일 게임서버에서 교환되는 패킷의 크기가 비교적 작으므로 비교대상인 나머지 메모리 풀보다는 높은 성능을 보인다 할 수 있다. Boost 라이브러리의 메모리 풀도 높은 성능을 보이며, 메모리 블록 크기를 변경하여도 거의 일정한 성능을 보인다.

4.3 임의크기 메모리 할당과 해제

임의크기 메모리 할당과 해제는 임의크기의 메모리를

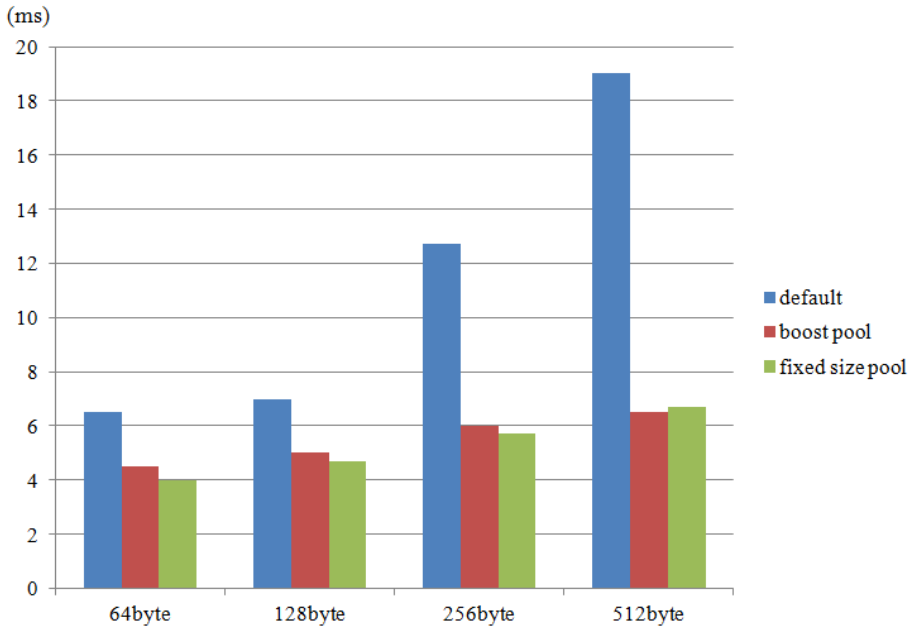


Fig. 19. Performance of Memory Pool in Random Size Memory Allocation and Deallocation

10000번 할당하였을 때 소요된 시간과 할당된 메모리들을 연속적으로 모두 해제하였을 때 소요된 시간을 측정하였다. Fig. 18에서 x축의 값은 임의로 메모리를 할당할 때 최대 메모리 크기를 나타낸다. 메모리 할당과 해제 성능은 Fig. 18과 같다.

제안한 방법으로 구성한 메모리 풀이 boost 메모리 풀과 함께 전반적으로 기본 메모리 할당자를 이용한 메모리 할당 및 해제보다 우수한 성능을 보인다. 그리고 평균적으로 제안한 방법에서의 메모리 풀이 더 우수한 성능을 보이지만 최대 512 byte 크기의 메모리 할당 및 해제에서는 약간 부족한 성능을 보였다. 이는 메모리 할당 상황에서 연속된 메모리 블록이 할당되어야 하는 경우가 빈번히 발생하면서 연속된 메모리 블록에 대한 탐색 소요시간과 더불어 연산량도 증가하기 때문에 성능이 조금씩 저하되는 것을 확인할 수 있다. 이 문제는 제안하는 방법에서의 메모리 풀의 청크의 크기와 개수, 청크 내 메모리 블록들의 크기와 개수를 어떻게 지정해주는가에 따라 달라질 수도 있다. 예를 들어, 청크 내 메모리 블록들의 크기를 512 byte로 설정하였다면 0~512 byte 사이에 어떠한 메모리 할당 요청이 발생하더라도 메모리 블록을 하나만 할당해주면 되기 때문에 연속된 메모리 블록을 찾는 시간이 소요되지 않는다. 하지만 이렇게 한다면 앞서 실험한 4.2절의 고정크기의 메모리 할당과 다르지 않다. 그리고 10 byte처럼 작은 크기의 메모리 할당에도 512 byte를 사용하게 되는 상황이 발생하게 되는데, 이때는 나머지 502 byte가 낭비된다. 따라서 사용자는 메모리 블록의 효율성과 할당 및 해제 성능을 고려하여 제안하는 방법에서의 메모리 풀을 활용하는 것이 좋다.

5. 결론 및 향후 연구

본 논문에서 제안한 메모리 풀을 이용하여 메모리를 할당 및 해제할 때 다른 메모리 풀보다 우수한 성능을 나타냄을 알 수 있었다. 이는 작은 메모리 블록을 자주 다루게 되는 게임서버에서 사용할 메모리 풀의 설계 목적에 부합한다. 물론, 위의 실험 예제보다 더 큰 메모리 블록을 요구할수록 성능은 떨어지겠지만 이점은 다른 메모리 풀도 가지고 있는 공통된 문제점이다. 따라서 작은 메모리 블록이나 객체들을 다루는 게임서버에서는 해당 논문에서 제안하는 메모리 풀을 사용하게 되면 더 높은 성능 향상을 기대할 수 있다. 더불어 메모리 단편화도 발생하지 않아 이로 인한 게임서버의 성능 저하 문제도 해결된다. 그리고 해당 메모리 풀의 생성 시 청크의 개수와 크기, 청크 내 각 메모리 블록의 크기를 설정하는 것은 사용자가 이를 활용하고자 하는 시스템에 맞추어 올바르게 설정하는 것이 필요하다.

향후 연구에서는 해당 메모리 풀의 범용성을 좀 더 넓히기 위해 2048byte 이하의 메모리 블록들을 사용해도 성능 저하가 없도록 개선할 것이다. 그리고 메모리 해제 시 메모리 블록 연결 리스트의 재구성에 소요되는 탐색시간을 줄이는 연구와 함께 다중 쓰레드 환경에서의 메모리 풀의 안정성 보장을 위한 연구도 진행하려 한다.

References

[1] J. Lim, I. Park, J. Chung, and K. Shim, “Technical Trend of Distributed Game Server”, *Electronics and Telecommunications Trends*, Vol.20, No.4, pp.93-102, 2005.

[2] Thor Alexander, “Massively Multiplayer Game Development,” Charles River Media, pp.210-230, 2003.

[3] D. Hahn, “Programming Benchmark of Online Game Server,” Information Publishing Group, 2008.

[4] C dynamic memory allocation, [Internet] https://en.wikipedia.org/wiki/C_dynamic_memory_allocation.

[5] Abraham Silberschatz, et al., “Operating System Concepts,” 7th Edition, Hongrung Publishing Co, 2008.

[6] Memory Pool [Internet], https://en.wikipedia.org/wiki/Memory_pool.

[7] Andre Alexandrescu, “MODERN C++ Design: Generic Programming and Design Patterns Applied,” Addison-Wesley, 2001.

[8] C++ Boost Library [Internet], http://www.boost.org/doc/libs/1_58_0/libs/pool/doc/html/boost_pool/pool/pooling.html.

[9] PJLIB [Internet], http://www.pjsip.org/pjlib/docs/html/index.htm.

[10] Jonatan Pålsson, Richard Pannek, Niklas Landin, and Mattias Petterson, “A Generic Game Server,” University of Gothenburg, Department of Computer Science and Engineering, 2011.

[11] Vlad Mihai Alecu, “Developing a Client-Server Architecture and Minimizing Data Transfer for a Massively Multiplayer Online Game,” Utrecht University, Game and Media Technology, 2012.

[12] Microsoft IOCP [Internet], https://msdn.microsoft.com/ko-kr/library/windows/desktop/aa365198(v=vs.85).aspx.

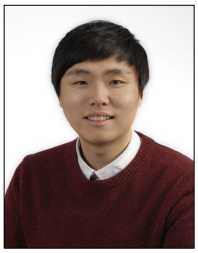
[13] C++ Memory Pool and Angra [Internet], http://celdee.tistory.com/638.

[14] D. I. Jo and S. Y. Ryu, “Design and Implementation of Event Based Message Exchange Architecture between Servers for Server Push,” *Korean Society For Internet Information*, Vol.12, No.4, Aug., 2011.

[15] S. B. Kim and J. D. Lee, “Design and Implementation of Borderless Casual Game Server using Virtualization,” *Korea Society Of Industrial Information Systems*, Vol.17, No.4, Aug., 2012.

[16] J. Deng, “Why to use memory pool and how to implement it,” [Internet], http://www.codeproject.com/Articles/27487/Why-to-use-memory-pool-and-how-to-implement-it, Jun., 2008.

[17] QueryPerformanceCounter [Internet], http://maytrees.tistory.com/81.



박 세 영

e-mail : qkrtpdud0304@gmail.com
2013년 한경대학교 컴퓨터공학과(학사)
2014년~현 재 숭실대학교 정보보안학과 석사과정
관심분야 : 운영체제, 네트워크 프로그래밍

최 재 영

e-mail : choi@ssu.ac.kr
1984년 서울대학교 제어계측공학과(학사)
1986년 미국 남가주대학교 전기·전자공학과 졸업(석사)
1991년 미국 코넬대학교 전기·전자공학부 졸업(박사)
1992년~1994년 미국 국립오크리지연구소 연구원
1994년~1995년 미국 테네시 주립대학교 연구교수
1995년~현 재 숭실대학교 컴퓨터학부 교수
관심분야 : 시스템소프트웨어, 병렬/분산처리, 고성능컴퓨팅(HPC)

최 종 선

e-mail : jongsun.choi@ssu.ac.kr
2000년 숭실대학교 컴퓨터학부(학사)
2002년 숭실대학교 컴퓨터학과(석사)
2008년~2010년 유한대학교 e-비즈니스과 전임교원
2010년 숭실대학교 컴퓨터학과(박사)
2011년~2012년 숭실대학교 지능형로봇연구소 연구원
2012년~2013년 서일대학교 인터넷정보과 전임교원
2013년~현 재 숭실대학교 컴퓨터학부 조교수
관심분야 : 시스템소프트웨어, 병렬/분산처리, 지능형 로봇

김 은 희

email : ehkim@seoil.ac.kr
1989년 숭실대학교 전자계산학과(학사)
1996년 숭실대학교 컴퓨터학과(석사)
2006년 숭실대학교 컴퓨터학과(박사)
2007년~2009년 숭실대학교 정보미디어 기술연구소 전임연구원
2010년~2012년 숭실대학교 지능형로봇연구소 전임연구원
2013년~현 재 서일대학교 인터넷정보과 조교수
관심분야 : 병렬/분산처리, 클라우드 컴퓨팅, 정보보호