자바가상기계 최적화를 위한 가비지 컬렉션 알고리즘과 힙 메모리 연구

임동기°, 배철성, 정민수
경남대학교 컴퓨터공학과

Study of Garbage Collection Algorithm and Heap Memory for Java Virtual Machine Optimization

DongKi Im, CheolSung Bae, MinSoo Jung
Dept. of Computer Engineering, Kyungnam Univ.
E-mail : toodulli@hawk.com.kyungnam.ac.kr

요 약

최근에 발표된 마이크로소프트의 C#이라는 언어에서도 채택했듯이 그 동안 자바 개발자의 특권으로만 여겨졌던 가비지 컬렉션(garbage collection)은 개발자로 하여금 메모리와 관련된 고민으로부터 해방시켰다. 단순히 메모리의 효율적 관리뿐 만 아니라 프로그램의 무결성을 높여 자바 보안정책에 중요한 부분을 제공한다.

본 논문에서는 자바 가상 머신의 최적화를 위해서 가비지 컬렉션을 처리에 효과적인 알고리즘과 힙 메모리를 설계하였다.

1. 서론

MS 의 차세대 플랫폼인 닷넷이 발표되면서, 개발자들은 도스에서 윈도우로 이동할 때만큼의 큰 변화를 맞고 있다. 단지 자바와 닷넷을 서로 경쟁적이고 배타적인 관점에서 비교하는 것이 아닌 상호 공존이라는 입장에서 생산적이고 발전적인 관점에서 접근이 필요하다.

근본적으로 자바 VM 이나 닷넷의 CLR(Common Language Runtime)은 플랫폼 위에 올라가는 가상기계(VM)에서 출발한 것이므로 크게 다를 바가 없다. 그 중 가비지 컬렉션을 사용한 메모리 처리 부분은 특히 그렇다.

그 동안 자바 개발자를 부러워 한 점 중에 하나인 가비지 컬렉션이 닷넷에서 추가 되었다는 점을 보아도 가비지 컬렉션의 역할을 짐작할 수 있을 것이다.

자바 가상 기계의 가비지 컬렉션과 닷넷의 CLR 은 기법과 알고리즘에서의 차이가 존재할 뿐 실제 가비지 컬렉션이 하는 역할은 자바 가상 기계이나 CLR 이 유사할 것이다.

그래서 본 논문과 관련된 자바 가상 기계의 관점에서 가비지 컬렉션 알고리즘과 힙 메모리

본 논문은 정보통신부 (2001-149-2)지원으로 수행되었음.

구조를 살펴 보면, 자바 가상 기계는 모든 객체를 자바 가상 기계의 힙 영역에 저장한다. 프로그래머는 'new' 명령을 통해서 객체를 생성할 수 있을 뿐 생성된 객체를 제거할 수 있는 명령이 없다. 따라서 메모리가 무한 하지 않는 이상 반드시 가비지 컬렉터가 필요하게 되는 것이다.

가비지 컬렉터의 두 가지 중요한 역할을 살펴보면, 첫째, 가비지 컬렉터가 있기 때문에 자바 플랫폼은 특유의 보안 특성을 유지 할 수 있다. 즉, 악의적인 코드 또는 프로그래머의 실수로 인해 적절하지 못한 자원을 해제할 수 있는 가능성을 막아준다. 둘째, 가비지 컬렉터로 인해 개발자는 좀 더 쉽게 원하는 로직을 구현할 수 있게 되고, 인류가 없는 프로그램을 작성하기가 수월해지며, 좋은 설계가 가능해진다.

정리 하면 자바 플랫폼에서 가비지 컬렉터가 수행하는 두 가지 역할은 생산성과 보안성인 것이다.

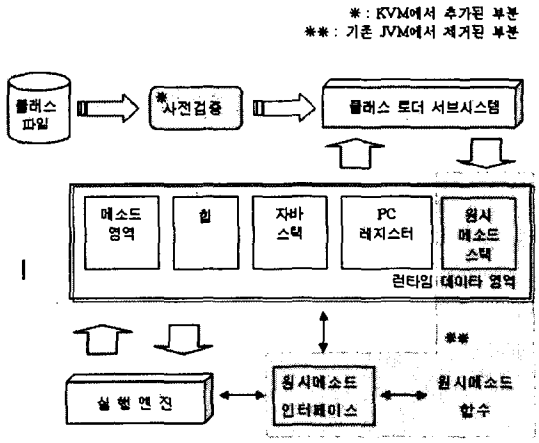
이런 중요한 역할의 가비지 컬렉션이 좀 더 효과적으로 실행되기 위해서 효율적인 알고리즘과 힙 메모리의 설계가 필요하다. 그래서, 본 논문 2 장에서 가비지 컬렉션을 알아보기 전에 관련연구로써 실제 가비지 컬렉션이 처리되어지는 자바 가상기계의 구조와 가비지 컬렉션 알고리즘을 살펴보고 3 장에서 퍼스널 자바에서의 가비지 컬렉션 알고리즘을 살펴보고 힙 메모리를 설계해서 보다 더 효율적인 알고리즘과 힙 메모리 구조를 제안해 본다. 마지막으로 4 장 결론 및 향후 연구 방향에 대해서 살펴 보겠다.

2. 관련연구

본 장에서는 관련연구로서 K-자바 가상 기계의 내부 구조에 대해서 알아보고, 실제 사용 되고 있는 K-자바 가상 기계나 퍼스널 자바 가상 기계 그리고 CLR에서의 가비지 컬렉션의 기법과 알고리즘을 살펴 본다.

2.1 K-자바 가상기계의 구조 분석

K-자바 가상 기계의 구조는 [그림 1]과 같이 클래스 로더 시스템(Class Loader System), 실행 엔진(Execution Engine), 런타임 데이터 영역(RunTime Data Areas)으로 구성되어 있다. 기본적으로 기존의 자바 가상 기계와 유사하다.



[그림 1] 자바 가상 기계와 K-자바 가상 기계의 내부구조

자바 가상 기계에서 프로그램을 실행시키는 과정을 살펴보면 다음과 같다.

① 적재

적재(Loading)란 애플리케이션에서 사용하려는 클래스파일을 가상 기계로 전달하는 과정을 말한다.

② 배치

배치(Linking)이란 로딩된 클래스파일, 즉 바이너리 코드가 가상 기계에 의해 수행될 수 있는 상태로 만드는 과정이다.

③ 검증

검증은 적재된 데이터의 형이 자바 언어의 의미(semantics)를 따르고 있는지 혹은 가상 기계의 내부 형식을 만족하는지를 검사하는 단계이다

④ 실행

실행은 실제 로드된 클래스 파일을 실행엔진으로 실행시키는 단계이다.

2.2 가비지 컬렉션 알고리즘

2.2.1 참조되지 않는 객체를 검출하기 위한 방법

① Reference Counting Collection 기법

레퍼런스 카운팅 기법은 객체에 대한 레퍼런스의 수를 세고 있다가 다른 변수가 그 객체를 참조하면 참조계수를 1증가시키고 참조가 범위를 벗어나거나 변수에 다른 값이 배정되면 참조계수를 1감소시킨다. 참조계수가 0이 되면 가비지 컬렉트 시킨다.

② Tracing Collection 기법

트레이싱 기법은 객체 레퍼런스에 대한 루터 노드에서 시작해서 객체 레퍼런스를 추적하며 마크(mark)시키거나 스윕(sweep)시킨다. 이 과정에 마크되지 않는 오브젝트를 가비지 컬렉트 시킨다.

③ Generational Collection 기법

객체들을 생명주기에 따라 세대별로 따로 처리하는 알고리즘이다. 예를 들면, 닷넷 CLR의 가비지 컬렉트는 G0, G1, G2로 구분되는 세가지 영역으로 객체를 관리하며 마크앤스윕과 컴팩팅 기법을 사용한다.

④ Adaptive Collection 기법

자바 핫스팟 가상기계의 가비지 컬렉션에서 사용하는 기법으로써 상황에 따라 적절한 컬렉팅 알고리즘을 사용하게끔 설계되어져 있다. 짧은 생명주기를 갖는 객체들을 위해서는 카핑(copying) 컬렉터를 사용하고 긴 생명주기를 갖는 객체들을 위해서는 Incremental Pauseless Collector 기법을 사용한다. Incremental Pauseless Collector는 트레인(train) 알고리즘을 사용하여 끊어짐 현상을 0.01초 이하의 작은 조각으로 나눠 사람이 느낄 수 있을 정도의 끊김 현상을 제거하였다.

2.2.2 메모리 단편화를 정리하기 위한 방법

① Compacting 기법

힙 단편화를 최소화 하기 위해서 힙 한쪽 끝으로 참조중인 객체를 이동시킨다. 그렇게 해서 한쪽 끝에 사용 가능한 공간을 만든다.

② Copying 기법

새로운 장소에 참조 되어지는 객체를 옮긴다. 옮겨진 객체가 연속적으로 배치되면서 옛 공간에 단편화된 공간을 확보할수 있다.

3. 퍼스널 자바의 가비지 컬렉션

3.1 퍼스널자바의 가비지 컬렉션 알고리즘

현재 퍼스널자바의 가비지 컬렉터는 컨저베이티브 마크-컴팩트(conservative mark-compact) 방식의 알고리즘을 따르고 있다. 즉, 힙 영역의 핸들 스페이스나 오브젝트 스페이스가 부족하여 메모리를 할당할 수 없을 경우 루트 셋을 스캔하여 살아있는 객체를 마크한다.

그리고 스윕(sweep) 작업을 통해 필요한 여유공간을 확보한다. 만일 스윕 작업을 통해서 충분한 여유공간을 확보하지 못한 경우에는 메모리 컴팩션(memory compaction)을 수행한다.

이 작업은 인접한 여러 개의 빈 공간들을 합쳐서 가능한 큰 덩어리의 블럭으로 만드는 작업이다. 그래도 메모리가 부족할 경우에는 시스템으로부터 추가의 메모리를 할당 받아 사용한다.

3.2 자바 힙 초기화 알고리즘

객체를 생성할 때마다 힙이 하나씩 생성이 된다. 가비지 컬렉션이 이 힙에서 이루어 지므로 자바 힙의 초기화를 통한 메모리 설계는 중요성을 가진다. InitializeAlloc() 메소드에서 자바 힙을 초기화 한다.

최고 요구 힙 메모리 값을 나타내는 max_request 값은 16777216을 할당하고, startup시 최소 힙 메모

리로 사용될 양인 min_request은 1048576로 설정한다. 전체 힙을 미리 할당할 경우 시작 메모리를 max_request값인 16777216을 사용한다.

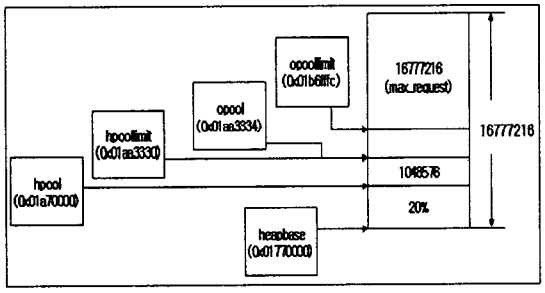
InitializeAlloc0() 메소드에서 역시 max_request값과 min_request값을 매개변수로 입력받아 실질적인 자바 힙 초기화를 실시하게 된다.

자바 힙을 초기화 시키는 부분이다.

```
heaptop = heapbase + max;
hpool = heapbase + (int) ((heaptop - heapbase) * 0.20);
hpoollimit = hpool + min;
```

[그림 2] 자바 힙 초기화 부분

heaptop, hpool, hpoollimit을 잡아 준 후에 InitializeGC()메소드에서 가비지 컬렉션 마크 비트 배열(mark bit array)을 초기화해서 할당한다. 즉, 마크 비트에 0을 할당한다.



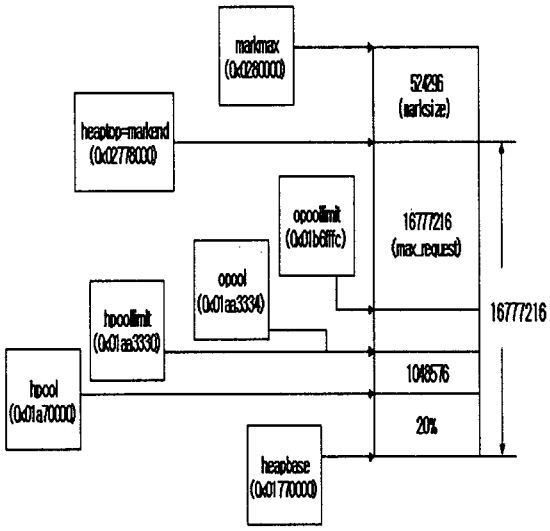
[그림 3] 초기화 후 힙 메모리 구조

```
opoollimit = hpoollimit - sizeof(hdr);
opool = hpoollimit;
marksize
= ((max/(OBJECTGRAIN*BITSPERMARK) + 1) * 2)
* sizeof(*markbits);
markbits
= (unsigned int *) sysMapMem(marksize,
&marksize);
markmax = markbits + marksize /
sizeof(*markbits);
markend = markbits;
```

[그림 4] 오브젝트 풀과 마크사이즈 설정 부분

루트 셋에서 스캔하여 마크를 실시 하다 공간이 부족할 경우 마크 비트 배열을 확장하기 위해서 expandMarkBits()메소드를 호출한다.

이렇게 되면 [그림 4]에서 보는 것과 같이 마크 비트가 증가 하게 된다. 즉, markend가 (markend + 32768/sizeof(*markbits))만큼 증가하게 되는 것이다. 이때 원래 만들어 졌던 힙에 꼭 연속적일 필요는 없다.



[그림 5] 초기화 후 힙 메모리 구조

자바 가상 기계에서 힙 메모리는 청크(Chunk)단위로 관리 된다. 이 청크를 사용해서 객체를 마크, 스윕, 컴팩션을 실행하게 된다. 핸들과 객체는 청크에 할당되고, 더 많은 메모리가 요구 될 때는 추가 청크를 할당하게 된다. 청크를 해제할 때는 청크를 운영체제에 돌려준다. 청크는 연속적일 필요가 없고 핸들 청크와 객체 청크로 분리되어져 이중 연결 리스트(doubly linked list)로 연결되고 관리되어진다.

청크의 구조를 살펴보면 다음과 같다.

[표 1] 청크의 자료 구조

void*	chunkHandle
struct CHUNK_BLK	*nextPtr
struct CHUNK_BLK	*prevPtr
Long	chunkFlags
Long	allocSize
Long	freeCnt
unsigned char	*startPtr
unsigned char	*endPtr
unsigned char	*freePtr
unsigned char	*physEndPtr
unsigned long	*markBits

3.3 Mark, Sweep, Compaction알고리즘

퍼스널 자바 가상 기계에서의 가비지 컬렉션에 두 가지 단계가 있다. 마크단계는 트리를 순회하며 만나는 객체를 마크하는 것이고, 스윕단계는 마크되지 않은 객체를 반환하고 결과 메모리를 재사용 가능하도록 만드는 단계다. 이 단계에는 객체의 파이널라이제이션(finalization)을 포함한다.

스윕작업을 통해서도 충분한 공간을 확보하지 못했을 경우에 메모리 컴팩션을 실행한다. 그래도 부족할 경우는 운영체제에 추가 메모리를 요구해서 할당받아 사용하게 된다.

① Mark

루트 로드에서 스캔을 하면서 각각의 객체를 마크시키는 부분이다. 스캔을 하면서 적재 되지 않은 클래스를 마크 비트 배열에 마크하게 된다. 이 과정에 프리미티브 클래스도 마크하게 되는데 로마이징(romizing)이 되었다면 프리미티브 클래스는 롬안에 들어가기 때문에 마크할 필요가 없게 된다.

```
markStickyClasses();
markPrimitiveClasses();
markROMClassStatics();
```

② Sweep

freeSweep(free_space_goal)에서 메모리의 추가여부를 알려주는 free_space_goal을 매개변수로 가져와서 마크되지 않고 남아있는 객체들을 찾기 위해 스캔하고 메모리 할당을 해제시킨다. 모든 객체는 finalization되기 위해서 마크 과정이 있어야 한다. 그리고, 그 결과로서 메모리를 프로그램이 다시 쓸 수 있도록 반환한다. 다음은 Do문을 돌면서 chunk별로 메모리를 해제시킨다. 이 부분에서 힙의 컴팩션 여부를 리턴해 준다.

```
chunk->freePtr = chunk->startPtr
chunk = chunk->nextPtr
```

[그림 5] 메모리 스윕 부분

③ Compaction

여유분의 블록 메모리의 양인 free_space_goal이 부족하면 힙을 컴팩션하게 된다.
compactHeap(&last_free, free_space_goal)에서 chunk의 모든 데이타를 compact한다.
매개변수로 넘겨진 destChunk에 chunk를 옮겨주면서 compact를 실시한다. 그런후 연속적이 빈 블록들을 합한다.

3.4 보다 효과적인 가비지 컬렉션 알고리즘

현재 사용되고 있는 컨저베이티브 마크 컴팩트(conservative mark-compact) 알고리즘은 'stop' 과 'go' 방식으로 동작하므로 가비지 컬렉션 작업에 소요되는 시간을 제한시킬 수 없다.

즉, 실시간 자바 응용 프로그램의 수행 도중 메모리가 부족할 경우, 응용 프로그램의 수행을 멈추고 가비지 컬렉션 작업을 수행하여 사용 가능한 메모리를 찾게 되는데, 이러한 가비지 컬렉션 작업에 소요되는 시간을 한계를 알 수 없어 실시간 응용 프로그램이 주어진 데드라인(deadline)을 만족함을 보장할 수 없게 된다.

따라서 가비지 컬렉션 작업의 수행 시간을 제한시킬 수 있도록 점진적으로 수행되는 가비지 컬렉션의 구현이 필요하다.

다음으로 현재의 가비지 컬렉터는 루터 셋(특히, 자바 프레임 내의 오프랜드 스택)을 스캔 할 때 컨저베이티브한 방식에 의해 객체를 마크한다. 따라서 메모리의 누수가 발생할 여지가 남아있고,

더욱이 실시간에 오랜 시간 동작하는 경우에는 메모리의 누수가 축적되어 실시간 시스템의 동작을 멈추게 할 수도 있다. 따라서 정확한 가비지 컬렉터의 구현이 필요하다.

또한 마크-스윕 방식의 알고리즘은 메모리 파편화 현상을 초래하기 때문에 컴팩션 작업으로 인한 오버헤드 등으로 인하여 메모리 할당에 많은 오버헤드가 발생한다. 그러나 카핑(copying) 알고리즘을 적용할 경우 메모리 파편화 현상을 근본적으로 없앰으로써 메모리 할당을 간단하게 만든다. 사실 카핑 알고리즘을 적용하기 위해서는 위에서 언급한 정확한 방식의 루터 섹 스캔이 가능해야 한다.

핫스팟 1.3.1 가상 기계와 CLR 이 동일한 제너레이셔널(generational), 마크 앤 컴팩트 방식을 지원하고 있는데 실시간 자바 응용프로그램의 실행을 위해서 보안해야 할 부분이 있다.

4. 결론 및 향후 연구 계획

본 논문에서는 퍼스널 자바 가상 기계의 최적화를 위해서 많은 알고리즘을 살펴보았다. 그 중에서 힙 메모리를 관리함에 있어서 객체 참조상에서 발생할 수 있는 문제점을 최소로 하는 트레인 알고리즘을 선택한다. 그리고 힙 할당, 해제 과정에서 발생하는 메모리 단편화 문제를 제거하기 위해서 일반적으로 많이 사용되고 있는 'stop'과 'copy' 방식에서 발생한 비효율성을 개선한 제너레이셔널(generational) 컬렉터 알고리즘을 트레인 알고리즘과 함께 자바 가상 기계의 최적화를 위한 방법으로 제안 한다. 물론 트레인 알고리즘을 사용해서 힙 메모리를 셋(set)과 블록(block)을 관리 함으로써 보다 효율적인 힙 메모리 관리 또한 가능하다.

다양한 알고리즘과 기법이 존재하기 때문에, 자바 가상 기계에서 가비지 컬렉션의 알고리즘과 구현방법에 따라 가상머신의 최적화 문제는 앞으로도 계속 연구 해야 할 분야이다.

향후 연구계획으로는 현재 이루어지고 있는 가비지 컬렉션 알고리즘이 느린 속도와 중앙처리장치의 자원소비문제를 가지고 있다. 그리고 실시간 자바 응용프로그램의 실행에서는 특히 문제점을 가질수 있다. 비록 핫스팟 1.3.1 가상 기계의 경우 CLR과 동일한 제너레이셔널, 마크 앤 컴팩트 방식을 지원하며 많은 발전이 있었지만 보다 효율적인 가비지 컬렉션 처리를 위해서 가비지 컬렉션의 알고리즘과 구현에 대한 연구가 요구 된다.

[참고문헌]

[1] Java 2 SDK Documentation 1.4
[2] Java Virtual Machine Specification
[3] Bill Venners, " Inside the Java Virtual Machine second edtion"
[4] http://java.sun.com/, Sun Microsystems, Java Home Page
[5] Engel저, 곽용재역, " 자바 가상 머신 프로그래밍"
[6] http://www.mobilejava.co.kr/