

#9

20174627 김혜진

**다음에서 가비지 수집기 알고리즘을 기술하는 2개의 논문을 읽고서 각각의 방법을 기술하고, 이들을 비교하라.**

### 각각의 논문을 비교하기에 앞서 가비지 수집기(Garbage Collection)가 무엇인가에 대해서 짚고 넘어가겠습니다. 가비지 수집기(Garbage Collection)는 동적으로 할당된 메모리 영역 가운데에서 더는 사용할 수 없게 된 영역을 탐지하여 자동으로 해제하는 기법으로 말 그대로 쓰레기 수집기 입니다. 사용할 수 없게 된 영역이라 함은 어떠한 변수도 가리키지 않게 된 영역을 의미합니다. 주로 Heap 메모리에서 처리 및 사용 여부를 식별하고 사용하지 않는 객체는 제거합니다.

### 

### 간략히 코드로 설명해 보겠습니다. 위 사진의 코드에서 for문이 반복될수록 실행문 안의 객체는 새롭게 생성되고 이전의 객체는 가비지가 되기 때문에 메모리 소모가 늘어나게 됩니다. 특정 몇몇 언어들은 이러한 메모리 부하 방지를 위해 자동으로 가비지 수집기(Garbage Collection) 기능이 함께 제공되기도 합니다. 결과적으로 동적 할당 메모리 영역 전체를 완벽하게 관리할 필요가 없게 되며, 버그를 줄이거나 막을 수 있게 됩니다.

### 위와 같은 장점이 있으면 단점도 있기 마련입니다. 객체가 필요 없어지는 시점을 알고 있더라도 가비지 수집기(Garbage Collection) 알고리즘에서 메모리 해제 시점을 추적해야 하므로 오버헤드가 되기 때문에 해제할 메모리를 결정할 때 비용이 발생하게 됩니다. 또한, 프로그램이 예측 불가능하게 일시적으로 정지할 수 있기에 실시간 시스템에 적합하지 않게 됩니다. 따라서 가비지 수집기(Garbage Collection)가 발생하는 타이밍 및 점유시간 예측이 어렵습니다.

|  |
| --- |
| H. SCHORR and W. M. WAITE,  “**An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures”** |

본 논문에서는 속도와 스토리지 활용의 상당한 이점을 제공하는 새로운 알고리즘을 보여줍니다. 해당 알고리즘을 통해 사용되는 언어는 리스트로 작성될 수 있으므로 어느 정도의 독립성을 갖춘 시스템을 보장합니다.

리스트 언어의 가장 유용한 기능 중 하나는 객체 프로그램을 실행하는 동안 동적으로 스토리지를 할당하는 기능입니다. 이는 사용 가능한 공간 목록을 통해 수행되며 사용되지 않는 레지스터를 포함합니다.

처음에 사용 가능한 리스트에는 프로그램이 차지하지 않는 모든 스토리지가 포함되어 있습니다. 또한 등록자가 분리되어 프로그램이 실행될 때 리스트 구조로 형성되는데, 이는 프로그램의 실행으로 인해 일반적으로 사용 가능 목록이 소진될 때 일시 중단되어야 하며, 더는 필요하지 않은 목록 구조의 해당 부분이 있는 경우 문제가 발생하게 됩니다.

이러한 문제를 해결하기 위해 참조 카운터가 소진될 때까지 레지스터가 사용 가능 목록으로 반환되지 않도록 합니다. 그다음 가비지 수집을 시작하여 전체 목록의 구조를 추적하고, 일부 목록에 첨부된 레지스터를 표시하도록 합니다. 결과적으로 더는 필요하지 않은 레지스터는 목록에 첨부되지 않으므로 표시되지 않게 됩니다. 모든 리스트의 추적이 끝나면 표시되지 않은 모든 레지스터는 사용 가능한 리스트로 새롭게 형성되어 집니다.

하지만 위의 해결 방안에는 두 가지 문제가 있습니다. 가장 기본적인 문제는 목록 추적입니다. 일반적으로 목록은 각 분기로 나누어져 있으며, 나누어진 모든 분기는 추적할 수 있어야 합니다. 하지만 위 방안의 경우 가비지 수집기가 발생한 분기 지점을 기억해야 하므로 목록에 여러 번 접근해야 한다는 문제가 있습니다.

다음으로 데이터가 부호 있는 숫자로 구성된 단어일 때 문제가 발생하게 됩니다. 가비지 수집 시 목록에 첨부된 레지스터들은 – (음수)로 설정이 되지만, 반면 첨부되지 않은 레지스터들은 + (양수)로 유지됩니다. 그런 다음 새로운 사용 가능 목록이 형성되면 연결된 레지스터의 부호가 + (양수)로 재설정 됩니다. 그렇게 되면 목록 처리 언어의 동적 스토리지 할당 원칙을 위반하게 되고, 비트 테이블에서 사용하는 저장소를 가비지 수집기로 제거해야 하므로 상당한 번거로움을 겪게 됩니다.

이 두 가지 문제를 보완하려면 목록을 단방향으로 하여 가비지 수집을 해주어야 합니다. 동시에 모든 목록 또한 추적할 수 있어야 합니다. 가비지 수집을 시작하여 전체 목록의 구조를 추적하고, 가비지 수집이 발생한 분기 지점을 모두 기억해야 하는 상황에서 제한된 양의 임시 저장소를 사용하는 것은 사실상 불가능 하므로 하위 목록을 무시하고 포인터(시스템에서 목록 요소의 주소 필드가 다음 레지스터의 위치를 포함하는 것)를 반대로 되돌려야 합니다. 공유된 리스트를 갖는 단방향 구조 목록에서 포인터를 되돌리게 되면 가비지 수집 리스트의 헤드부분으로 값이 리턴하게 됩니다.

이 첫 번째 패스 동안 각 레지스터의 부호는 음수가 됩니다. 첫 번째 패스의 끝은 목록의 끝에 도달하거나, 부호가 마이너스인 목록의 레지스터가 발견될 때 도달하게 됩니다. 리스트의 단방향 스캔 중에는 각 리스트 레지스터 헤드의 내용이 무시됩니다. 단방향 스캔이 종료될 때 입력되는 역방향 스캔은 원래 포인터를 복원 한 목록으로 다시 이동하게 됩니다. 또한 ID에 추가하여 각 레지스터의 헤드를 검사한 후, 하위 목록에 대한 참조가 포함되어 있는지 확인합니다. 이러한 레지스터가 발견되면 레지스터가 분기점임을 나타내도록 접두어가 변경됩니다. 그런 다음 스캔이 취소되고 리스트의 단방향 스캔이 수행됩니다. 하위 목록의 역방향 스캔은 해당 하위 목록의 모든 하위 목록을 선택하여 추적하고, 추적 중인 기본 목록의 첫 번째 레지스터로 되돌아가 다음 목록이 표시될 수 있도록 합니다. 모든 목록이 추적되고 나면 순차적으로 메모리 스캔이 시작됩니다. 부호가 양수인 레지스터는 사용 가능 목록에 표시되며, 음수 레지스터는 여전히 사용중인 목록에 표시되어야 하므로 부호가 양수로 재설정 됩니다. 이처럼 새로운 사용 가능 리스트 구성이 완료되면 프로그램이 재실행 되게 됩니다.

이러한 알고리즘을 바탕으로 가비지 수집 절차는 가능한 많은 임시 저장소를 사용하여 마지막 루틴을 사용하고 이 저장소가 가득 차면 현재 분기의 나머지를 추적하게 됩니다. 이런 식으로 사용 가능한 임시 메모리의 양에 대해 가장 효율적인 수집을 실현할 수 있으며, 어떤 구조에서도 오류가 발생하지 않게 됩니다.

|  |
| --- |
| L. Peter Deutsch and Daniel G. Bobrow, “**An Efficient, Incremental, Automatic Garbage Collector”** |

본 논문에서는 힙에서 자동으로 스토리지를 할당하고 프로그래머가 더는 특정 항목이 유용하거나 액세스할 수 없다는 표시를 하지 않아도 되는 Lisp와 같은 시스템의 스토리지 회수 문제를 해결하는 새로운 방법을 설명합니다.

스토리지 관리 기능은 일반적으로 모든 스토리지를 빠른 랜덤 액세스 메모리에 보관할 수 있는 환경에서 가장 잘 작동하도록 설계되었습니다. 지금까지 구현된 방식은 실행 중 오버헤드가 있으며 이는 사용 중인 스토리지 양에 비례하게 됩니다.

해당 알고리즘의 경우 여러 레벨의 스토리지와 큰 가상 주소 공간을 사용하는 시스템에서 잘 작동하도록 설계되었으며, 대부분의 셀이 정확히 한 번만 참조되고 저장 횟수가 확보될 때만 참조 횟수가 정확한 값이라는 기준을 정해 두었습니다. 트랜잭션 파일은 참조 카운트에 대한 변경사항을 저장하고, 다중 참조 테이블은 두 번 이상 참조되는 항목의 카운트를 저장합니다. 이를 기반으로 더는 사용하지 않는 스토리지의 자동 교정은 다음 두 가지 기술로 수행됩니다.

첫 번째, 가비지 수집입니다. 스토리지가 가득 차면 시스템은 프로그래머가 직접 액세스 할 수 있는 모든 데이터 구조에서 시작하여 이들 및 모든 간접 액세스가 가능한 구조를 표시해주어야 합니다. 반면, 표시되지 않은 구조의 경우 모든 공간이 회수됩니다.

이러한 가비지 수집의 단점은 사용하지 않는 공간을 수집하는 데 걸리는 시간이 사용 중인 공간의 양에 비례한다는 점입니다. 이는 실제 메모리 시스템에서 보조 저장장치에 존재하는 많은 페이지에 액세스 하여 모든 공간을 추적해야 하기 때문입니다.

두 번째, 참조 횟수입니다. 할당된 각 데이텀은 이를 참조하는 다른 데이터 수를 추적합니다. 이로 인해 변숫값과 할당된 다른 데이터의 포인터가 표시되지 않으며, 카운트가 0이 되면 데이텀을 회수할 수 있습니다. 이 기술의 경우 외부에서 액세스 할 수 없는 자기 참조 구조를 감지할 수 있는 능력이 없을뿐더러, 참조 카운트는 혹이 몇 비트에 불과하므로 최댓값에 도달하게 되면 절대 감소할 수 없게 됩니다.

이러한 속성에는 참조 횟수 외에 가비지 수집기가 필요합니다. 액세스 할 수 없는 대부분의 스토리지는 참조 횟수가 0이 되어 선택되기 때문에 가비지 수집기는 훨씬 덜 빈번해지며, 스토리지의 최대 활용을 보장할 수 있게 됩니다.

참조 카운트 체계의 주요 장점은 트랜잭션 지향적이라는 것입니다. 즉, 적절한 스토리지 교정에 필요한 카운트의 증가 및 감소가 데이터의 액세스 가능성에 영향을 미치는 트랜잭션 수에 비례한다는 뜻입니다. 이러한 작업은 일반적으로 계산에 균일하게 분산되기 때문에 저장소 유지 관리에 대한 오버헤드는 총 계산에 비례하게 됩니다. 이러한 참조 카운트 체계에는 두 가지 중요한 요소가 있습니다.

첫 번째, 할당된 모든 데이텀 입니다. 즉, 목록의 셀과 레코드를 뜻합니다. 패킹 제약조건은 기준점 자체 내에 기준 카운트 저장을 종종 어색하게 하며 이는 추가적인 계산 오버헤드를 발생시키게 됩니다. 또한, 카운트가 최댓값에 도달함에 따라 가비지 수집이 더 빈번하게 발생하여 집니다.

두 번째, 데이텀의 접근성에 영향을 줄 수 있는 모든 거래는 참조 횟수를 변경해야 한다는 것입니다. 여기에는 새로운 데이터 생성이 포함되지만 포인터 변수 재설정, 함수 바인딩 및 바인딩 해제 그리고 기존의 할당된 데이터로의 모든 저장소 또한 포함됩니다. 많은 시스템에서 변수에 대한 모든 할당의 카운트를 변경하는 데 필요한 시간은 많지 않습니다. 참조 변경이 있을 때 참조된 데이터가 보조 메모리에 있을 수 있는 시스템 에서는 특수 하드웨어가 코어 내 참조 변경 위해 간단한 코드를 허용하지만 카운트가 갱신되기까지의 시간을 기다릴 수는 없기 때문입니다.

모든 참조 횟수 체계에는 이러한 문제점들을 처리하기 위해 독립적으로 추적이 가능한 가비지 수집기 저장소가 필요합니다. 이 독립적인 가비지 수집기는 버려진 데이터의 동적 재활용과는 반대로 더 많은 선형 목록을 만들기 위한 저장과 재구성의 결합이라고 보여집니다.

결과적으로 본 논문의 알고리즘은 선형화를 위한 추적 기술을 사용하고 새롭게 선형화 된 공간에 유지 될 구조를 복사하며, 복사된 모든 공간이 블록으로 사용 가능한 스토리지로 리턴 됨을 보이고 있습니다. 또한, 교정 프로세스는 거의 전적으로 프로세서 및 주 계산에 사용된 메모리와 로컬 메모리 분리율에 의해 수행될 수 있는 것으로 보여집니다. 메인 프로세서가 편리한 지점에 도달하게 되면 현재 스택을 보조 프로세서로 전송할 수 있으며, 보조 프로세서는 포인터가 도착할 때 포인터를 찾아보고 스택의 마지막 조각을 수신하자마자 회수 가능한 셀 목록을 얻을 수 있습니다.

**[ 논문 비교 ]**

첫 번째 논문(①)의 알고리즘은 속도와 스토리지 활용의 상당한 이점을 제공할 수 있도록 목록을 단방향으로 하여 가비지 수집을 해주었고, 동시에 모든 목록을 추적할 수 있었습니다.

두 번째 논문(②)의 알고리즘은 시스템의 스토리지 회수 문제를 해결하기 위해 가비지 수집 선형화를 위한 추적 기술을 사용하였습니다.

가비지 수집을 시작하여 전체 목록의 구조를 추적하고 가비지 수집이 발생한 분기 지점을 모두 기억해야 하는 ①의 경우, 제한된 임시 저장소 문제로 포인터를 되돌리게 되면 가비지 수집 리스트의 헤드부분으로 값이 반환되어 집니다. 반면, 선형화를 적용 시킨 ②의 경우 새롭게 선형화 되어 생선 된 구조의 모든 공간이 블록으로 사용 가능한 스토리지로 리턴 됨을 보아 ②는 ①과 다르게 저장소에 대한 규격이 정해져 있지 않음을 알 수 있었습니다.

또한, ②의 경우 대부분의 셀이 정확히 한 번만 참조되고 저장 횟수가 확보될 때만 참조 횟수가 정확한 값이라는 기준을 정해 두었기 때문에 카운트에 대한 기준이 명확함을 알 수 있습니다. 이를 기반으로 더는 사용하지 않는 스토리지의 자동 교정을 수월하게 할 수 있는 반면, ①은 참조 카운터가 소진될 때까지 레지스터가 사용 가능 목록으로 반환되지 않도록 하였습니다. 그 다음 가비지 수집을 시작하여 전체 목록의 구조를 추적하고, 일부 목록에 첨부된 레지스터를 표시하도록 하여 더는 필요하지 않은 레지스터는 목록에 첨부되지 않게 하여 표시에서 제외 시켰습니다. 이렇게 모든 리스트의 추적이 끝나면 표시되지 않은 모든 레지스터는 사용 가능한 리스트로 새롭게 형성되게 하였으나, 목록 추적과 데이터의 부호에 따른 문제로 인해 ②에 비해 현저히 낮은 선형성을 보입니다.

마지막으로 ②의 가비지 수집 에서는 스토리지가 가득 차면 모든 간접 액세스가 가능한 구조를 표시해주는데, 여기서 표시되지 않은 구조의 경우 모든 공간이 자동으로 회수가 됩니다. 하지만 ①의 경우 각 방향의 포인터에 대한 스캔 및 레지스터 헤더에 대한 검사 절차를 거쳐야 하며 이를 통해 임시 저장소에 대한 메모리를 확보해야 한다는 번거로움이 있습니다.

**Java의 가비지 수집기에 대해서 조사하여 기술하고, 앞의 두 논문과의 차이점을 기술하라.**

|  |
| --- |
| **Java Garbage Collection** |

자바 애플리케이션에서는 C/C++과 같이 개발자가 직접 프로그램 코드에서 메모리를 명시적으로 해제하는 행위를 하지 않습니다. 대신 JVM의 가비지 수집기가 더이상 사용되지 않는 인스턴스를 찾아 메모리에서 삭제하는 행위를 하게 됩니다. 가비지 수집기는 힙 영역의 메모리를 JVM이 판단해 더이상 사용되지 않는 인스턴스는 자동으로 할당 된 메모리를 삭제 하도록 하는 행위 입니다.

이렇게 메모리를 복사하고 해제 하는 행위를 실행 하기 위해서는 자바 어플리케이션은 가비지 수집기를 실행하기 위한 스레드를 제외하여야 하며, 이외의 모든 스레드는 멈추고 가비지 수집기가 완료된 이후에나 다시 스레드가 실행 상태로 돌아가게 됩니다.

이와 같이 이외의 모든 스레드의 작업이 멈추는 상태를 Stop The World라고 하고 어떠한 가비지 수집기 알고리즘을 사용 하더라도 Stop The World 상태에 부딪히게 됩니다. 일반적으로 자바 어플리케이션의 경우 오래 유지되는 객체에 비해, 생성된 이후에 짧은 시간 동안에 필요없게 되는 짧은 생명 주기의 객체가 더욱 많습니다.

각각의 객체는 Old Generation과  Young Generation으로 나뉘며 Sun JVM의 힙에 속해있습니다. Young Generation은 다시 Eden과 두개의 Survivor Space로 구성되어 집니다. Old Generation의 경우 생명 주기가 긴 오래된 객체를 가비지 수집기를 대상으로 하는 영역입니다. 해당 영역에서 발생되는 GC를 Major GC 라고 하며 Minor GC에 비해 속도가 느립니다. Young Generation의 경우 생명 주기가 짧은 젊은 객체를 가비지 수집기를 대상으로 하는 영역입니다. 해당 영역에서 발생되는 GC를 Minor GC 라고 하며 Major GC에 비해 속도가 빠릅니다. Sun JVM의 힙 영역의 가비지 수집기 동작 과정은 이러합니다.

1. 최초의 객체는 Young Generations의 Eden에서 생성된다.

2. 이후 Eden이 Full이 가비지 수집기가 한 번 발생한 후 살아남은 객체는 Survivor 영역 중 From Space로 이동한다.

3. 다시 Eden이 가비지 수집기가 발생하면 From Space의 객체는 To Space로 이동하게 된다.

4. 마지막으로 이 과정이 반복되면서 객체가 여전히 살아있는 상태라면 Old Generationd 영역으로 이동하게 된다.

객체는 최초에 Eden에서 생성이 되고 Eden의 메모리 공간이 Full이 되면 메모리 복사가 일어나게 됩니다. 이러한 작업을 Minor 가비지 수집기 라고 하는데, Minor 가비지 수집기는Eden의 객체를 검사하여 사용되고 있지 않거나 참조하고 있지 않는 객체는 삭제하고 나머지 살아있는 객체들은 From space가 꽉 찰 때까지 Eden에서 From Space로 이동하게 됩니다. 이러한 작업은 짧은 생명 주기를 가질 수 밖에 없는 객체는 최대한 짧게 가지고 있겠다는 Sun JVM의 특징을 그대로 보이고 있습니다.

Eden이 다시 Full이 되면 From Space의 Lived 객체는 To Space로 이동합니다. 그러나 실제로 JVM은 각 Minor 가비지 수집기가 발생할 때 마다 Survivor Space의 포인터를 유지하고 있다가 단순히 From Space에서 To Space로 변경해 주는 것입니다. Eden이 다시 Full이 되면 To Space에 있는 Lived 객체는 Old Generation 으로 복사됩니다.

Minor 가비지 수집기는 매우 빠르고 효율적 입니다. 소요시간은 Young Generation의 크기에 따라 다르지만 1초 미만으로 알려져 있습니다. 또한 JVM Thread Processing를 멈추게 하는 등의 부작용도 발생하지 않습니다. Young Generation이 모두 꽉차게 되면 가비지 수집기는 가용 메모리를 확보하기 위해 Major 가비지 수집기를 수행합니다.

Major 가비지 수집기는 객체들이 실행 상태로 있는지 여부를 파악하기 위해 모든 스레드의 실행을 잠시 멈추고 살아있는 객체를 표시하고 사용하지 않는 객체는 제거하여 힙을 정리 합니다. 이 Major 가비지 수집기는 스레드를 잠시 멈추게 되고 Mark and Sweep 작업을 위해 CPU에 부하를 가하게 되며 이러한 작업은 보통 Minor 가비지 수집기에 비해 10배 이상의 시간을 사용하기 때문에 성능에 악영향을 주게 됩니다.

결과적으로 생명 주기가 짧은 젊은 객체는 Old Generation으로 올라가기 전에 Young Generation에서 제거 되게끔 하고 오래된 객체의 경우 Old Generation에 상주시켜 상대적으로 아주 저렴한 Minor 가비지 수집기만으로 힙의 유지가 가능하게 유도하는 것이 좋다고 생각합니다.

**[ Java의 가비지 수집기와 논문 비교 ]**

②에서 스토리지가 가득 차면 모든 간접 액세스가 가능한 구조를 표시해주고, 여기서 표시되지 않은 구조의 경우 모든 공간이 자동으로 회수되는 경우가 있었다. 마찬가지로 자바의 가비지 수집기 또한 더이상 사용되지 않는 인스턴스는 자동으로 할당된 메모리를 삭제해주는 기능을 가지고 있음을 알 수 있었습니다. 이러한 부분으로 봤을 때 ②와 자바의 가비지 수집기는 ①보다 효율성이 높음을 알 수 있었습니다.

또한, ①의 가비지 수집 절차는 가능한 많은 임시 저장소를 사용하여 마지막 루틴을 사용하고 이 저장소가 가득 차면 현재 분기의 나머지를 추적하게 되도록 구성되어 있었습니다. 이런 식으로 사용 가능한 임시 메모리의 양에 대해 가장 효율적인 수집을 실현할 수 있었습니다. 여기서 자바의 가비지 수집기와 유사한 부분을 찾을 수 있었는데, 자바의 가비지 수집기 에서 Young Generation이 모두 꽉차게 되면 가비지 수집기는 가용 메모리를 확보하기 위해 Major 가비지 수집기를 수행하는 부분입니다. 양측 모두 메모리를 확보하기 위해 움직이는 행위임을 알 수 있습니다.

최종적으로 ①과 ② 그리고 자바의 가비지 수집기의 궁극적 목표는 동적으로 할당된 메모리 영역 가운데에서 더는 사용할 수 없게 된 영역을 탐지하여 해제하는 역할을 수행한다는 것 입니다.

다만 그 역할을 속도나 스토리지 그리고 이를 처리하는 과정에 있어 어떤 알고리즘을 사용하여 보다 신속하고 효율적이게 처리하냐에 따른 장단점이 나뉘고 있습니다.

결국 원활한 서비스를 위해서 가비지 수집기를 어떻게 관리하느냐가 시스템의 안정성에 큰 변수로 작용하게 될 것입니다. 하지만 주의해야 할 점은 어떤 서비스에서 A라는 가비지 수집 옵션을 적용해서 잘 동작한다 하더라도, 그 가비지 수집 옵션이 다른 서비스에서도 훌륭하게 적용되어 최적의 효과를 볼 수 있다고 생각해서는 안 된다고 느꼈습니다. 각 서비스에서 생성하는 객체의 크기나 생존 주기가 모두 다르며, 장비의 종류도 다양하기 때문에 가비지 수집기의 옵션은 지속적인 관리와 모니터링이 필요하며, 해당 서비스에 가장 적합한 값을 도출해 낼 수 있도록 하여야 합니다.