GARBAGE COLLECTION ALGORITHM

Homepage: https://sites.google.com/site/doc4code/

Email: sj6219@hotmail.com

2011/10/21

요즈음 Java, C#과 같이 자동적으로 메모리를 관리해 주는 언어를 많이 사용하고 있다.

Object-C 의 경우 iphone 에서는 아직까지 Garbage Collection 을 제공하지 않지만, 매킨토시에서는 옵션으로 선택 가능하다. 이런 언어는 프로그래머가 메모리를 free 하지 않고 시스템에서 알아서 해주기 때문에, 개발하기가 쉽다는 게 장점이다.

Garbage Collection 은 이런 언어 환경에서 자동적으로 메모리를 free 해 주는 역할을 하고 있다.

Garbage Collection 을 사용하면 프로그램의 실행 성능이 떨어지는 것을 피할 수는 없다. 하지만 프로그램 개발을 쉽게 해 주기 때문에 잘 사용한다면 득이 될 수 있다.

GARBAGE COLLECTION 의 종류.

Garbage Collection 의 알고리즘에도 여러 가지 종류가 있다. Reference counting, mark-sweep, mark-compact, copying 등으로 분류할 수 있다. 각 알고리즘에는 장단점이 있어서 상황에 따라 각 알고리즘을 사용하거나, 여러 알고리즘을 조합하여 사용하곤 한다.

일반적인 알고리즘은 stop-the-world collector 방식이라 하는데, garbage collection 을 하는 도중에 다른 작업을 잠시 중단하고, garbage collection 이 끝난 후 기존 작업을 다시 시작한다.

이런 방식의 경우, 프로그램이 가끔씩 멈추게 되는데, 이런 단점을 줄여주는 알고리즘도 개발되어 있다.

그 중에 하나가 incremental 방식의 알고리즘인데, collection 작업을 한꺼번에 하는 것이 아니라, 여러 번에 나누어서 조금씩 한다. 한번에 하는 collection 작업량이 줄어들기 때문에, 멈추는 시간도 줄어든다.

그리고, Concurrent 방식의 알고리즘도 있는데, 이 방식에서는 프로그램의 작업을 그대로 실행하면서 동시에 별도의 전용 thread 에서 garbage collection 작업을 한다.

MARK-SWEEP COLLECTOR

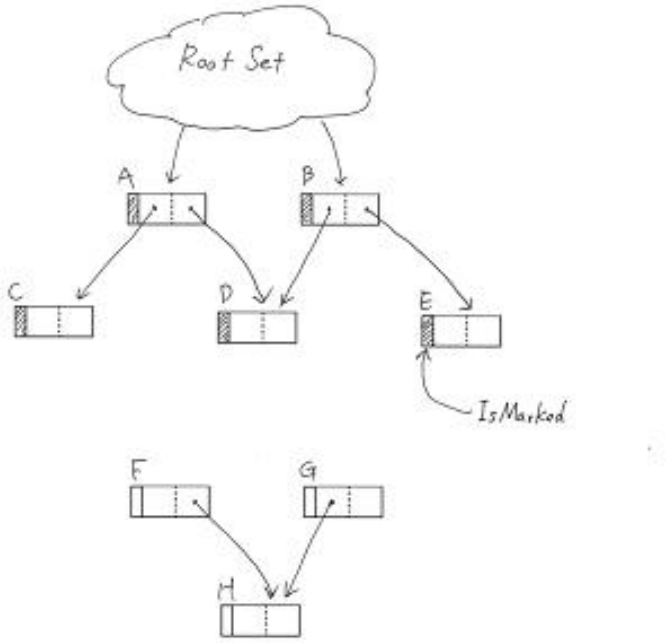

그림 1.

Mark-Sweep 은 가장 간단한 방식이어서 많이 사용되고 있다.

이 방식은 두 단계로 이루어 진다.

첫번째 단계인 Mark 단계에서는 현재 사용되는 Node 들을 찾아서 표시한다.

두번째 단계인 Sweep 단계에서는 첫 번째 단계에서 표시되지 않는 Garbage Node 들만 찾아서 메모리 반환(free)한다.

그림 1 에서보면 F, G, H 노드가 garbage 이기 때문에 반환하게 된다.

```
struct Node
{
        Node *   Child[...];
        bool     IsMarked;
};

Node *New()
{
        if free memory is empty
                Collect()
        Node *Newcell = Allocate();
        return Newcell()
}
```

New 함수에서 새로운 Node 를 생성해준다. 메모리가 충분하면 Allocate 함수를 이용해 새로운 메모리를 할당 받는다. 그러다가 Heap 의 메모리가 부족해지면 garbage collection 을 실행해서 메모리를 충분히 늘린 후에 할당한다.

```
void     Collect()
{
        Mark()
        Sweep()
        if free memory is empty
                Abort("memory exhausted")
}
```

Garbage Collection 은 Mark()와 Sweep()로 이루어 진다. RootSet 는 Global 변수나 Thread Stack 에 있는 변수에서 가리키는 노드 주소의 집합이다. Multi Thread 환경에서는 모든 쓰레드를 중단시키고(Suspend), 그 상태에서 글로벌 변수 및 각 쓰레드의 스택(stack)를 조사해서 로컬 변수들이 가리키는 메모리를 검사해야 한다.

```
stack<Node *> GCStack;

void     Mark()
{
```

```
        for Node **R in RootSet
                MarkNode(*R)
        while ! GCStack.empty()
                Node *top = GCStack.top();
                GCStack.pop()
                for Node **C in top->Child
                        MarkNode(*C)
}

void    MarkNode(Node *N)
{
        if N != NULL && ! N->IsMarked
                N->IsMarked = true
                GCStack.push(N)
}
```

MarkNode() 함수는 자식 노드를 검사해서, 아직 처리하지 않았다면 표시를 하고, 그 자식 노드를 스택에 추가한다.

```
#define INT_PTR(p) (int) p
char *Heap;

void    Sweep()
{
        Node *N = Heap;
        while N < Heap + Size(Heap)
                Node *Next = INT_PTR(N) + Size(N)
                if   ! N->IsMarked
                        Free(N)
                else
                        N->IsMarked = False
                N = NEXT
}
```

Sweep 단계에서는 Heap 를 앞에서부터 검사하면서 표시되지 않은 노드를 메모리에 반환(free)한다.

그림 1 의 경우에 위 알고리즘을 적용하면 우선 Rootset 에서 가리키는 노드 A, D 를 MarkNode()한다. 그 다음 노드 D 가 가리키는 B, C 노드를 MarkNode()한다. 다음으로, 노드 D 의 자식 노드 E 를 마크 처리하게 된다.

Mark-Sweep Collector 는 구현이 간단하지만, 메모리 fragment 가 발생하기 때문에 성능상 문제가 있다. 그래서 Mark-Sweep 보다는 Mark-Compact 나 Copying Collector 를 알고리즘을 사용하는 경우가 많다.

COPYING COLLECTOR

Copying Collector 는 새로운 방식을 사용한다. Heap 는 같은 크기의 from-space 와 to-space 의 두 블록으로 나뉘어 진다.

프로그램에서는 한 블록만 사용하다가, Copying Collector 에서 한 블록(from-space)에서 다른 블록(to-space)으로 Node 의 내용을 복사하면서, 압축(Compact)작업도 동시에 한다.

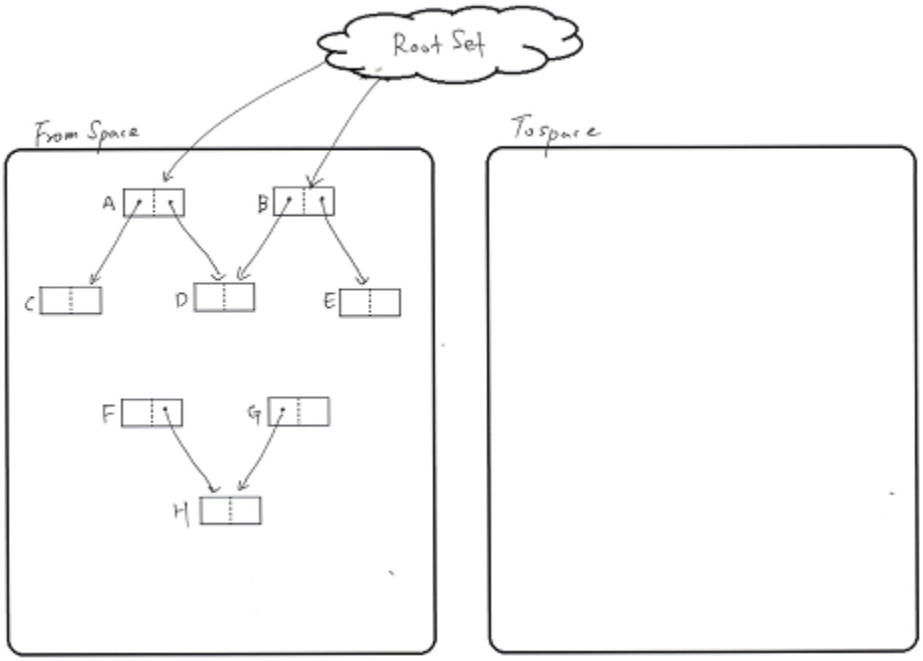

그림 2.

그림 2 는 Garbage Collector 를 하기 전의 모습이다.

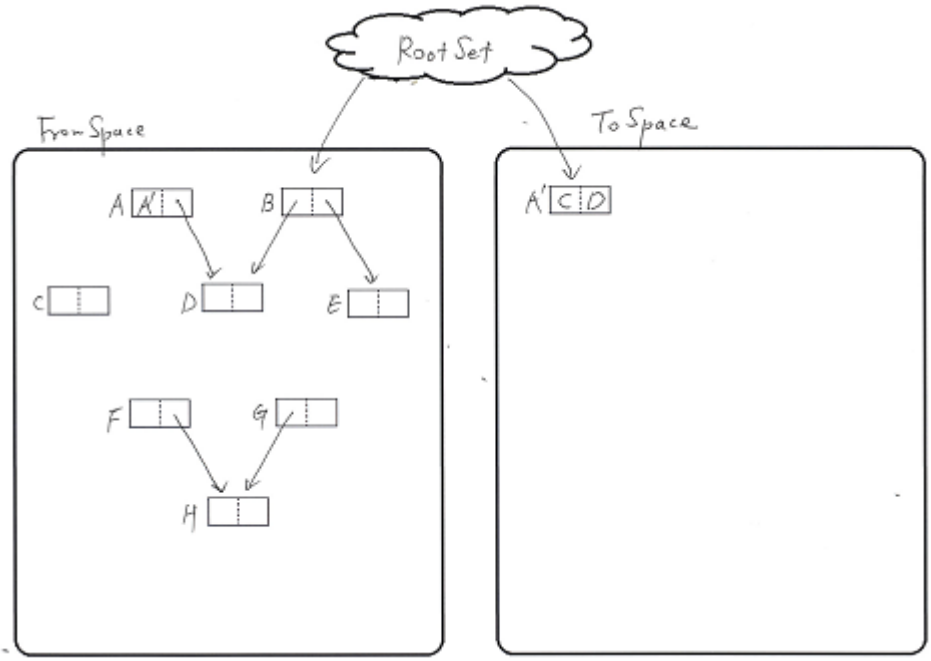

그림 3.

그림 3 은 root set 에서 가리키고 있는 Node A 를 처리하고 난 후의 모습이다. from-space 의 Node A 를 to-space 의 Node A'로 복사한다. 그리고, 기존 Node A 에 Node A'의 주소를 저장한다. rootset 에서 가리키고 있는 주소도 A'로 변경한다.

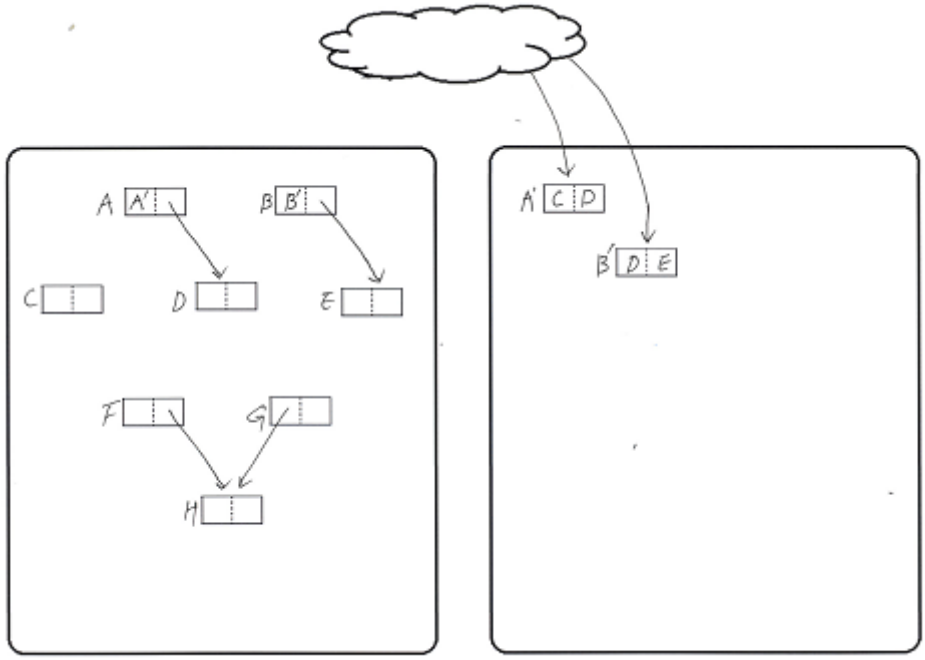

그림 4.

그림 4 는 rootset 에서 가리키는 Node B 를 처리하고 난 후의 모습이다.

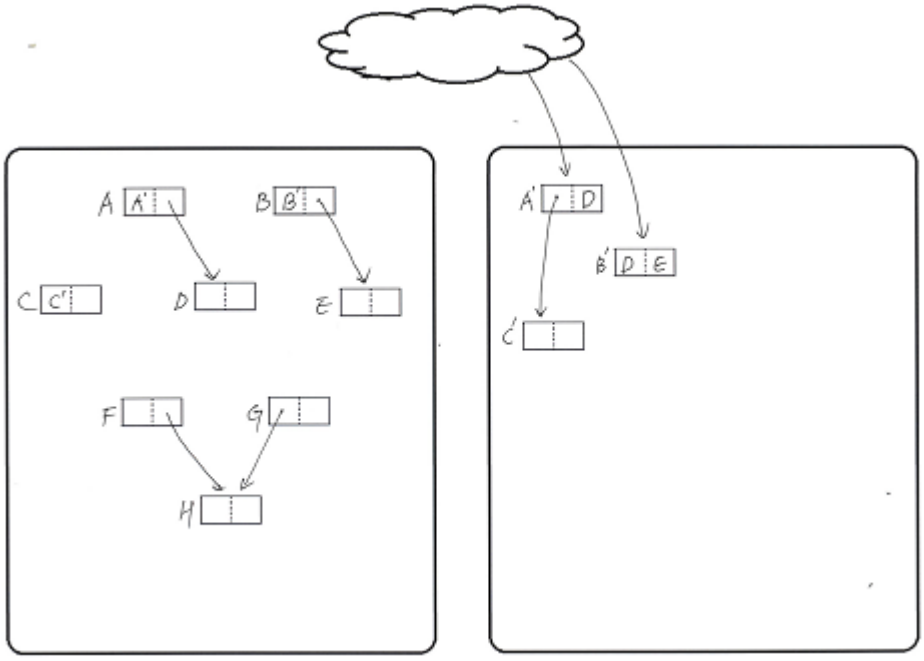

그림 5.

그림 5 에서는 노드 A'의 첫 번째 자식 노드 C 를 복사한다.

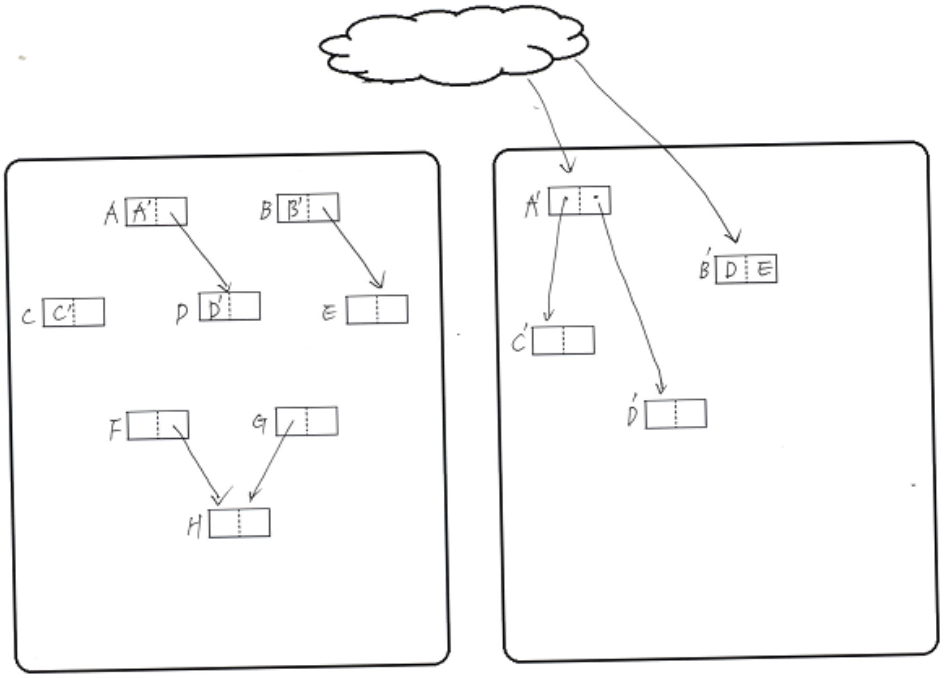

그림 6.

그림 6 에서는 노드 A'의 두 번째 자식 노드 D 를 복사한다.

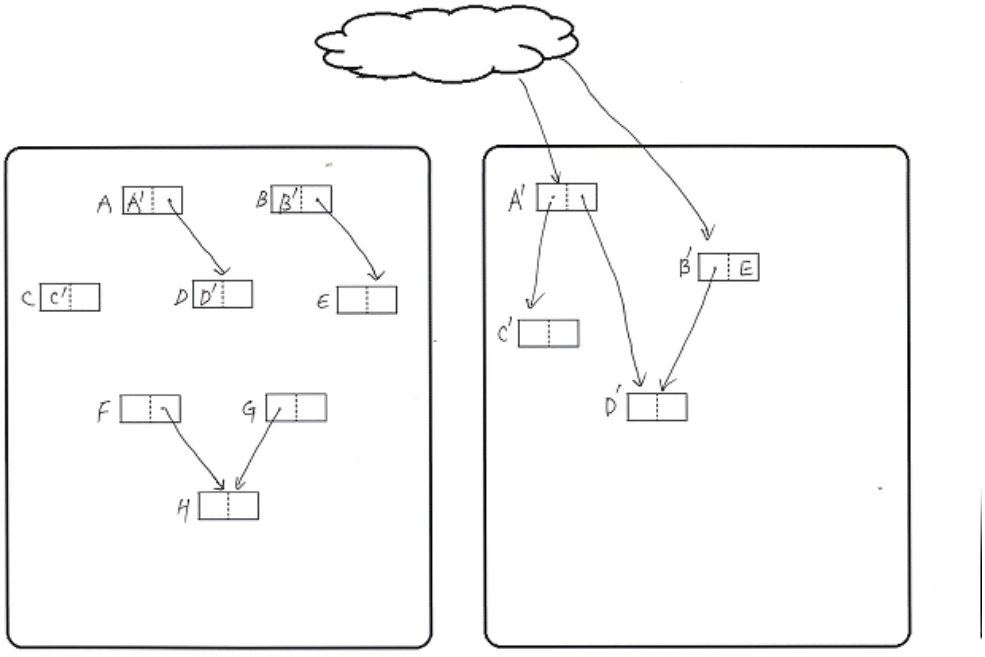

그림 7.

그 다음 노드 B'의 첫 번째 자식인 노드 D 를 다시 처리한다. 이 때, 노드 D 는 이미 복사되었고, 이동된 주소를 from-space 의 노드 D 에서 구할 수 있다.

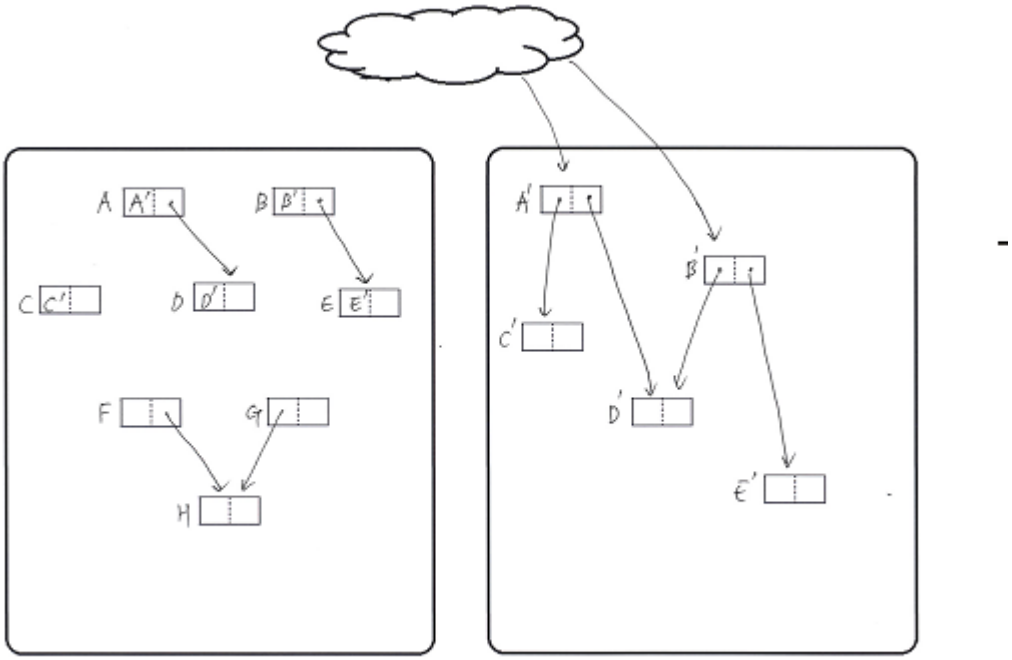

그림 8.

그림 7 에서 노드 B'의 두 번째 자식 E 를 복사한다. 그 후에, 노드 C', D', E'는 모두 자식이 없으므로 garbage collection 이 완료된다. Collect 단계가 끝나면 from-space 의 모든 노드들은 지워진다.

알고리즘은 다음과 같다.

```
#define INT_PTR(p) (int) p

char Heap[HEAP_SIZE]; // HEAP_SIZE is multiple of 2
char *ToSpace;
char *FromSpace;
char *Free;

void Initialize()
{
        ToSpace = Heap
```

```
        FromSpace = Heap + HEAP_SIZE /2
        Free = ToSpace
}

Node *New(int size)
{
        if Free + size >= ToSpace + HEAP_SIZE/2
                Flip()
        if Free + size >= ToSpace + HEAP_SIZE/2
                Abort("Memory Exhausted")
        Node *Newcell = Free
        Free = Free + size
        return newcell
}
```

프로그램 시작할 때 Initialize()을 호출하면 ToSpace, FromSpace, Free 변수들을 초기화한다. New 함수에서 메모리가 부족하면 Flip()을 호출해서 garbage collection 을 한다. 메모리는 to-space 에서 할당한다.

그리고, 메모리 할당하는 부분이 아주 간단하다. Free 가 가리키는 포인터 값을 증가시키기만 하면 된다. copying collector 를 사용하면 메모리 할당하는 부분의 성능이 좋아지게 된다.

```
void QueueInit();
bool QueueEmpty();
void QueuePush(Node *N);
Node *QueuePop();

void Collect()
{
        Flip()
        QueueInit()
        for Node **R in RootSet
                *R = CopyNode(*R)
        while ! QueueEmpty()
                Node *N = QueuePop();
                for Node **C in N->Child
                        *C = CopyNode(*C)
}

void Flip()
{
```

```
        char *Temp = FromSpace;
        FromSpace = ToSpace
        ToSpace = Temp

        Free = ToSpace
}

Node *&ForwardAddress(Node *N) { return N->Child[0] }

bool Forwarded(Node *N)
{
        return (ToSpace <= INT_PTR(N)) && (INT_PTR(N) < ToSpace + HEAP_SIZE/2)
}

Node *CopyNode(Node *N)
{
        if N == NULL
                return NULL
        if ! Forwarded(ForwardAddress(N))
                memcpy(Free, N, Size(N)) // Copy from N to Free
                FowardAddress(N) = Free
                QueuePush(Free)
                Free = Free + Size(N)
        return ForwardAddress(N)
}
```

Flip 함수에서는 우선 FromSpace 와 ToSpace 의 값을 서로 교환한다. 그 다음 RootSet 에서 가리키는 Node 와 그 자식 노드들을 from-space 에서 to-space 로 복사한다. 그러면서 from-space 에 있는 기존 노드에 새로운 노드의 주소를 저장한다. 이때, 메모리를 절약하기 위해 저장할 장소를 Child[0]와 함께 공유한다.

```
Node *QueueBegin;

void QueueInit()
{
        QueueBegin = Free
}

bool QueueEmpty()
{
```

```
        return QueueBegin != Free
}

void QueuePush(Node *N)
{
}

Node *QueuePop()
{
        Node *N = QueueBegin;
        QueueBegin = INT_PTR(QueueBegin) + Size(QueueBegin)
        return N
}
```

LIFO(Last In First Out) stack 구조체 방식으로 구현해도 되지만, 위와 같이 FIFO(Last In First Out) queue 방식으로 구현할 수도 있다. 이 방식은 별도의 메모리를 필요로 하지 않고 heap 의 메모리를 이용해 queue 를 구현하는 장점이 있다. 하지만, 자식 노드와 부모 노드 간의 메모리 위치가 멀어져서, 캐시 성능이 나빠지는 단점이 있다.

큐 방식보다는 스택 방식에서 부모 노드와 자식 노드가 가까운 위치에 배치되는 경향이 있다. 두 노드를 액세스할 때, 두 노드가 떨어져 있을 때 보다는 가까울 때 컴퓨터 하드웨어의 캐시 효율이 좋아진다.

Copying collector 는 장점은 copy-sweep 방식에 비해서 New()의 성능이 좋다는 것이다. 그리고, 메모리 fragment 가 줄어서 메모리도 효율적으로 사용할 수 있고 캐시 성능도 좋아진다. 그렇지만, copy sweep 방식에 비해서 2 배의 heap 메모리가 필요하다는 단점이 있다.

GENERATIONAL COLLECTION

메모리의 생성, 반환을 관찰해보면 98%정도의 메모리가 생성되지 얼마 안 가서 곧 반환된다고 한다. 즉, 최근에 생성된 메모리 오브젝트일수록 garbage 일 가능성이 높다. 그리고, 어느 정도 시간이 지나도록 살아있었다면(live), 그 이후에도 반환(free)되지 않고 살아있을 가능성이 점점 높아진다.

그래서, 오브젝트를 생성 나이(age)에 따라 분류하고, 나이가 젊은 오브젝트를 자주 garbage collect 하고 오래된(old) 오브젝트는 가끔씩 collect 하면, 전체적인 성능이 좋아진다.

generational collector 방식에서는 메모리 heap 을 여러 개의 generation 블록으로 나눈다. 새로 생성되는 오브젝트는 youngest generation 에서만 할당(allocate)된다. young generation 에 있는 오브젝트가 일정 시간이 지나도록 살아있다면 그 다음 (next older) generation 으로 이동한다 (promote). young generation 에는 생성된 지 얼마 안 된 오브젝트가 모여 있고, old generation 에는 오래된 오브젝트가 모이게 된다.

보통 young generation 에서는 copying collector 를 사용한다. young generation 에서 collect 하면서, 동시에 오래된 오브젝트를 그 다음 단계의 generation 으로 복사된다. 그리고, oldest generation 에서는 Mark-Sweep 이나 Mark-Compact 알고리즘 등이 주로 사용된다.

MINOR COLLECTION

youngest collection 만 collect 하는 것을 minor collection 이라고 부른다. minor collection 을 한 다음에도 메모리가 부족하다면 그 다음 old generation collect 하게 된다. Old generation 까지 collect 하는 것을 major collection 이라고 부른다. Garbage collection 의 성능을 높이기 위해서, minor collection 은 자주하고, major collection 은 드물게 한다

young generation 의 heap 크기는 old generation 의 heap 에 비해 크기가 작고, 대부분 오브젝트가 반환(free)되기 때문에, minor collection 의 처리시간은 짧다.

INTERGENERATION REFERENCE & REMEMBERED SET

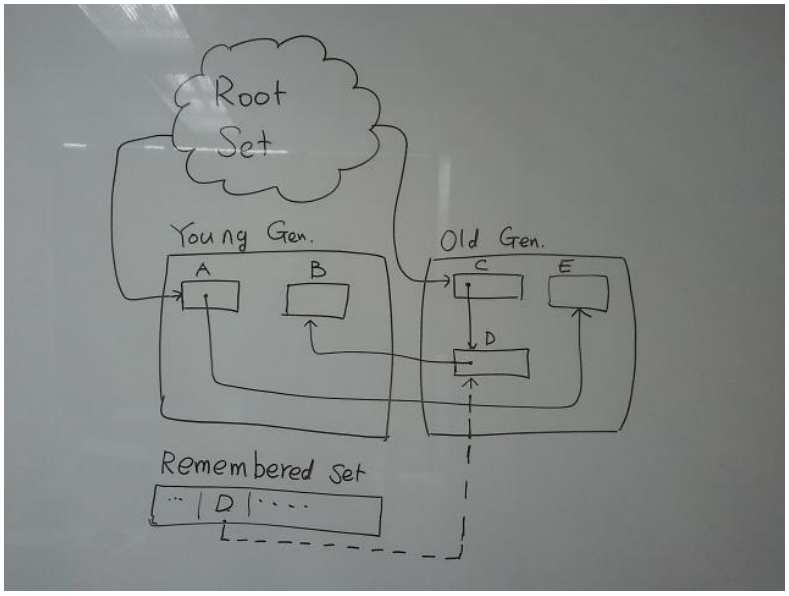

그림 8.

그림 8 에서 young generation 만 collect 한다고 해보자. 이 때 노드 A 는 rootset 에서 가리키고 있으므로 garbage 가 아닌 게 확실하다. 하지만 노드 B 는 rootset 에서 가리키지 않고 있기 때문에 garbage 로 잘못 처리 될 수 있다. 노드 B 를 제대로 처리하기 위해서는 old generation 에 속해있는 노드 D 가 노드 B 를 가리키고 있다는 정보가 필요하다. 이를 위해서, old generation 에 속한 노드에서 young generation 에 속한 노드로의 포인터의 집합을 remembered set 이라는 구조체에서 관리하게 된다.

그림에서 노드 E 에 대한 remembered set 은 필요 없다. old generation 을 collect 하기 전에 young generation 을 먼저 collect 하기 때문이다. young generation 을 collect 하는 과정에서 old generation 으로의 reference 를 구할 수 있다.

일반적으로, 같은 generation 에 속해있는 노드 간의 reference 에 비해서, 다른 generation 있는 노드의 reference 는 별로 없다. 생성시기가 비슷한 노드 간의 reference 가 대부분이기 때문이다. 또, young generation 에서 old generation 으로의 reference 에 비하여, old generation 에서 young generation 으로의 reference 가 더 적다.

intergenerational reference 가 생기는 것은 두 가지 경우에서 발생한다.

첫 번째는 프로그램에서 old generation 의 노드의 값이 young generation 의 노드를 가리키도록 값이 변경되는 것이다. 예를 들면 노드 D 가 노드 E 를 가리키고 있다가 노드 B 로 포인터의 값이 바뀌는 경우이다.

두 번째 경우는 minor collect 를 하는 과정에서 promote 되면서 발생하는 경우가 있다. 노드 C, D 가 원래 young generation 에 속해 있었는데 old generation 으로 promote 되었다면, 노드 D 에서 노드 B 로의 intergenerational reference 가 발생한다.

두 번째 경우는 collection 을 할 때, intergeneration reference 가 발생하는지 검사하면 되기 때문에 쉽게 처리할 수 있다. 하지만, 첫 번째 경우는 프로그램 실행 도중에 발생하기 때문에 처리하기 까다롭다. 다음 부분에서 첫 번째 경우를 어떻게 처리하는지 알아본다.

CARD MARKING

Intergenerational Reference 를 처리하기 위해서는 여러 가지 기법이 있다.

한가지 방법은 하드웨어의 virtual memory protection 기법을 이용하는 것이다. old generation heap 의 데이터가 바뀌는 것을 중간에서 가로채서 그 바뀐 page 를 기록하는 것이다. 더 효율적인 방법은 OS 에서 Page 의 Dirty Bit 를 바꾸는 부분을 수정해서 바뀐 페이지를 기억하는 것이다.

또 다른 방법은 compiler 를 수정해서 노드의 주소를 write 하는 코드를 변경하는 것이다. 이렇게 write 부분을 가로채어 추가 작업을 한다. 이렇게 추가된 코드를 write-barrier 라고 부른다.

Sun JDK 에서는 card marking 이라는 기법을 사용하는데, write-barrier 에서 generation heap 의 어떤 부분이 변경되는지 여부를 기록한다. Generation heap 를 일정크기(예를 들어, 128 바이트)의 card 로 나눈다. 그 다음 어떤 카드가 바뀌었는지를 기록하는 것이다. Card map 이라고 하는 byte map(또는 bit map)이 별도로 있어서 어떤 카드가 수정되었는지 관리한다.

```
char *ByteMap;

void WriteBarrier(Node *Parent, int Index, Node *NewChild)  // Parent->Child[Index] = NewChild
{
        char *Temp;

        MOVE NewChild, [Parent + Index]  // *(Parent + Index) = NewChild
        ADD Parent, Index, Temp          // Temp = Parent + Index
        SHIFT_RIGHT Temp, 7, Temp        // Temp = Temp >> 7
        CLEAR_BYTE [ByteMap + Temp]      // *(ByteMap + Temp) = 0
```

}

이것은 write-barrier 의 예이다. 이렇게 노드의 내용이 바뀔 때마다 bytemap 에 어떤 위치에서 바뀌었는지 저장한다. 나중에 garbage collection 을 할 때 전체 old generation heap 를 검사하지 않아도 card map 만 검사하면 intergenerational reference 를 찾아낼 수 있다. 카드크기가 128 바이트라면 heap 를 검사할 때보다 처리할 메모리가 1/128 로 줄어든다.

모든 write 코드에 write-barrier 를 추가할 필요는 없다. 컴파일러가 코드를 분석해서 write-barrier 의 양을 줄일 수 있다.

변수가 rootset 에 포함된 경우에는 write-barrier 가 필요 없다. 스택에 위치한 로컬변수에 저장하는 경우가 여기에 해당한다. 이런 경우가 많기 때문에 write-barrier 를 실제 추가하는 경우가 줄어든다.

THREAD SUSPENDSION

Garbage collect 도중에 heap 를 변경시키려면 다른 thread 를 안전한 상태에서 중지시켜야 한다. Managed object 를 사용하고 있는 도중에, 그 object 가 다른 주소로 이동하게 되면 안되기 때문이다. 따라서, managed object 를 사용하고 있는 도중에 중지시키지 않아야 한다.

여기서는 .Net 환경에서 thread 를 중지시키는 기법에 대해서 알아보겠다.

- 중지된 thread 가 native (unmanaged) code 를 실행하고 있는 경우

  Native code 에서 접근할 수 있는 managed object 는 모두 pin 되어 있기 때문에, native code 를 실행하는 도중에 managed object 의 주소가 바뀌지 않는다. Managed object 를 pin 해 놓으면, garbage collection 에서 그 메모리를 다른 주소로 이동하지 않는다. 그러므로 thread 가 계속 실행되도록 내버려 둬도 괜찮다. 그렇지만 native code 가 managed code 로 변경되는 순간에 thread 를 suspend 시켜야 한다. 스택 프래임을 조사하면 return address 의 위치를 찾아낼 수 있다. 이 return address 를 변경해서, 미리 준비된 코드를 실행시킨다. 이렇게, managed 코드로 바뀌는 순간을 가로채서(hijack) thread 가 중단되도록 한다.

- managed code 를 중단하는 경우

1. Fully Interruptible Code

JIT(Just-In-Time) 변환기가 중간 코드를 가지고 있기 때문에 현재 실행되고 있는 코드를 완전히 분석하게 만들 수 있다. 레지스터에 어떤 값이 들어있는지 추적할 수 있어서, 어떤 레지스터에 managed object 의 주소를 가지고 있는지 파악한다. 필요하다면 레지스터의 값을 변경한다.

하지만, 이렇게 코드를 분석하려면 미리 정보를 준비해야 하므로 코드가 커지게 된다. 그래서 많이 사용되지 않는다.

2. Return Address Hijackinig

Unmanaged code 에서 사용하는 방법과 마찬가지로 스택의 return address 부분을 가로채서 Thread 를 중단시킨다.

3. GC Safe Point

JIT 변환기가 프로그램 코드 중간에 garbage collection 를 해도 되는지 검사하는 코드를 삽입한다.

Garbage collection 을 해도 되는 안전한 위치를 GC Safe Point 라고 부르는데, GC Safe Point 에 검사하는 코드를 삽입해 thread 를 중단시킨다.

결론

Garbage collection 알고리즘에 대하여 공부하면서 놀란 것은, 예상한 것보다 좋은 알고리즘이 많이 개발되어 있다는 것이다. 성능에 크게 영향을 미치지 않으면서 쉽게 개발할 수 있는 방법으로 이용하면 좋겠다. 하지만, 높은 성능이 필요한 게임이나 서버 프로그램에는 적당하지 않다.

안드로이드 OS 에서는 Java 를 사용하고 있는데, garbage collection 을 하면 더 빨리 전원을 소모할 것 같다. 그래서, 배터리 용량과 메모리가 적은 모바일 환경에서는 불리할 것 같다.

참고문서

[1] The Garbage Collection Handbook: The Art of Automatic Memory Management by Richard Jones, Antony Hosking and Eliot Moss

Garbage collection 알고리즘에 대해서 자세히 설명한 책이다. 실제 사용되는 다양한 알고리즘을 공부하기 위해서는 꼭 읽어보기 바란다.

[2] Java theory and practice: Garbage collection in the HotSpot JVM

자바 GC 에 대해 설명이 간략하게 나와있다.

[3] Notes on the CLR Garbage Collector

.Net GC 에 대한 설명이 간략하게 나와있다.

[4] Incremental Collection of Mature Objects, Richard L. Hudson and J. Eliot B. Moss

Tree Algorithm 에 대한 논문이다.

[5] Fundamentals of Garbage Collection

MSDN 에서 있는 GC 정보