# ADVANCED GARBAGE COLLECTION ALGORITHM

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

Email: sj6219@hotmail.com

2011/10/21

이전 문서에서는 Garbage Collection 의 기초적인 것에 대해서 다루었다. 이번 문서에서는 좀 더 복잡한 알고리즘에 대해서 다뤄보겠다.

## MARK-COMPACT COLLECTOR

Copying-Collector 의 경우 Heap 의 메모리가 커지는 단점이 있다. 메모리를 절약하면서도 fragment 를 줄이기 위해서 Mark-Compact 알고리즘을 사용한다. 여기서는 Mark-Compact 중 가장 간단한 Lisp2 알고리즘에 대해서 알아보겠다.

### LISP2 ALGORITHM

```
struct Node
{
        Node *   Child[...];
        Node*    ForwardAddress;
```

```
};
stack<Node *> GCStack;

void    Collect()
{
        Mark()
        ComputeAddress()
        UpdateAddress()
        Relocate()
}

void    Mark()
{
        for Node **R in RootSet
                MarkNode(*R)
        while ! GCStack.empty()
                Node *top = GCStack.top();
                GCStack.pop()
                MarkNode(top)
}

void    MarkNode(Node *N)
{
        if N != 0 && N->ForwardAddress == 0
                N->ForwardAddress = 1
                for Node **C in N-> Child
                        GCStack.push(*C)

}
```

Mark-Compact 의 단점은 여러 단계가 필요하다는 점이다. Lisp2 알고리즘도 4 개의 단계로 이루어져 있다

첫 번째 단계인 Mark()는 Mark-Sweep 알고리즘의 Mark()와 거의 비슷하다.

메모리를 절약하기 위해 IsMarked 대신에 ForwardAddress 필드를 공유해서 사용한다는 점이 다르다. ForwardAddress 에 0 이 아닌 값을 가지면 Mark 된 것으로 간주하면 된다.

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

```
char *Heap;
char *Free;

void    ComputeAddress()
{
        Free = Heap
        Node *N = Heap;
        while N < Heap + Size(Heap)
                if (N->ForwadAddress != 0)
                        N->ForwardAddress = Free
                        Free = Free + Size(N)
                N = INT_PTR(N) + Size(N)
}
```

두 번째 단계인 ComputeAddress()은 ForwardAddress 필드에 새로 이동할 주소를 계산해서 저장한다.

```
void    UpdateAddress()
{
        for Node **R in RootSet
                if *R != 0
                        *R = (*R)->ForwardAddress
        Node *N = Heap;
        while N < Heap + Size(Heap)
                if N->FowardAddress != 0
                        for Node **C in N->Child
                                if *C != 0
                                        *C = (*C)->ForwardAddress
                N = INT_PTR(N) + Size(N)
}
```

UpdateAddress 에서는 Node 를 가리키고 있는 변수나 Node 의 필드를 기존 주소 값에서 새 주소로 바꾸는 기능을 한다.

```
void    Relocate()
{
```

```
        Node *N = Heap;
        while N < Heap + Size(Heap)
                Node *Next = INT_PTR(N) + Size(N)
                if N->ForwardAddress != 0
                        Node *Dest = N->ForwardAddress
                        N->ForwardAddress = 0;
                        memmove (Dest, N, Size(N)) // Copy from N to Dest
                N = Next
}
```

마지막으로 Relocate() 단계에서 각 노드를 새 주소로 이동시킨다.

Mark-Compact 는 각 Node 마다 ForwardAddress 필드가 추가로 필요하다는 단점이 있다. 그리고, 여러 단계를 거쳐야 하는 단점이 있다. 그렇지만, 처리과정에서 메모리를 순차적으로(sequential) 접근하기 때문에 효율적이다. Virtual Memory System 에서 메모리를 랜덤(random)하게 접근하면 Page fault 가 많이 발생하지만 순차적으로 접근하면 Page Fault 가 줄어든다.

또 생성된 순서를 그대로 유지되면서 Compact 하기 때문에, 실제 프로그램에서도 비슷한 위치의 메모리에 몰려서 접근할 가능성이 높다. 그래서, Page Fault 의 확률을 줄여주는 장점이 있다.

## INCREMENTAL COLLECTOR

Generational Collector 를 사용하면 전체적인 성능은 높아지지만 최악의 경우 최대 처리 시간을 줄이지는 못한다. Oldest Collection 을 할 때 전체 Heap 를 Full Garbage Collect 해야 하기 때문에 프로그램이 오랫동안 멈추게 된다. 멈추는 시간을 줄이려면 처리해야 할 작업을 여러 번으로 나누는 게 효과적이다. 이렇게 여러 단계에 걸쳐서 처리하는 방식을 Increment collection 이라고 부른다.

### TRAIN ALGORITHM

Train Algorithm 은 JDK 1.2 에서부터 사용되었던 Incremental collection 알고리즘이다. 공부해보면 굉장히 잘 만들어져 있고, 재미있는 알고리즘이다. Java 5 부터는 train 알고리즘 대신에 concurrent collector 를 사용하고 있다.

Train 알고리즘은 전체 Heap 을 여러 개의 블록으로 나눈다.

일단 전체 Heap 를 여러 개의 Train 으로 나눈다. 또한 Train 은 여러 개의 Car 로 나뉜다. 각 Car 는 generational 알고리즘에서 하나의 generation 과 비슷하게 remembered set 을 가진다.

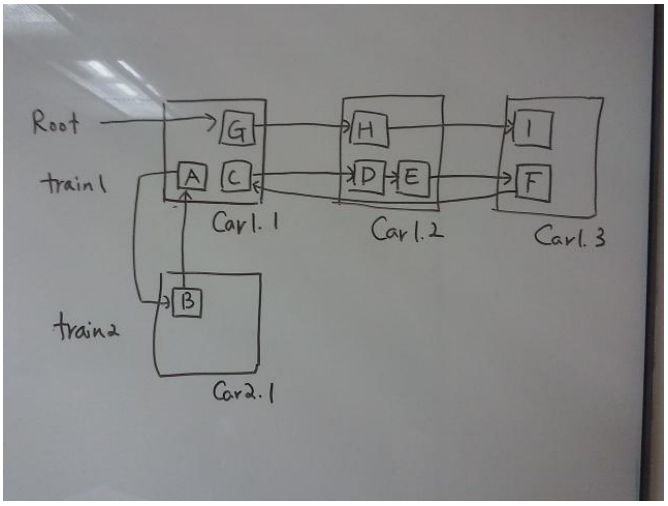


그림 1.

그림 1 에서 Train 1 는 Car 1.1, Car 1.2, Car 1.3 으로 이루어지고, Train 2 는 Car 2.1 로 이루어져 있다. Car 1.1 에는 노드 A, C, G 가 들어있다.

Train 알고리즘에서는 전체 Heap 를 한꺼번에 Collect 하지 않고, 가장 오래된 Car 한 개씩만 Collect 한다. 예제에서 첫 번째 Garbage Collection 을 하면 Car 1.1 만 처리해서 반환한다.

첫 번째 단계에서는 외부 Train 에서 현재 Car 가 속한 Train 내부로 가리키는 reference 가 없는지 검사한다. Rootset 에서부터 현재 train 으로 reference 가 없고, 현재 train 의 remembered set 이 비어 있다면, train 전체를 반환한다.

두 번째 단계에서는 rootset 에서 가리키는 노드를 외부의 최신 train 으로 이동시킨다. 구현하는 방법에 따라서 신규 train 을 만들어서 이동시킬 수도 있고, 기존 train 들 중에 최신 train 으로 이동할 수도 있다. 그림 1 에서 rootset 이 가리키는 노드 G 를 외부 train 2 로 이동시킨다.

세 번째 단계에서는 위에서 이동된 노드의 자식 노드도 부모 노드들을 따라 이동시킨다. 만약 이동해야 할 car 가 꽉 차 있다면, 해당 train 끝에 새로 car 를 만들어 추가한다.

네 번째 단계에서는 young generation 에서 promote 된 노드들을 해당 train 으로 이동시킨다. 보통, Generational Collector 환경에서 oldest generation 의 알고리즘으로 train 알고리즘을 많이 사용된다. Young generation 에서 노드들이 promote 되면, 그 노드를 가리키는 train 으로 이동시킨다.

다섯 번째 단계에서는 다른 외부 train 에서 가리키는 노드를 그 외부 train 으로 이동시킨다.

여섯 번째 단계에서는 현재 train 의 다른 car 에서 가리키는 노드를 train 의 마지막 car 로 이동한다. 마지막 car 가 꽉 차 있으면, train 끝에 새로 car 를 만들어 추가한다. 그림 1 에서 노드 C 는 Car 1.3 으로 이동한다.

이제 Car 1.1 에는 외부에서 가리키는 노드가 남아있지 않으므로 모두 반환하면 된다.

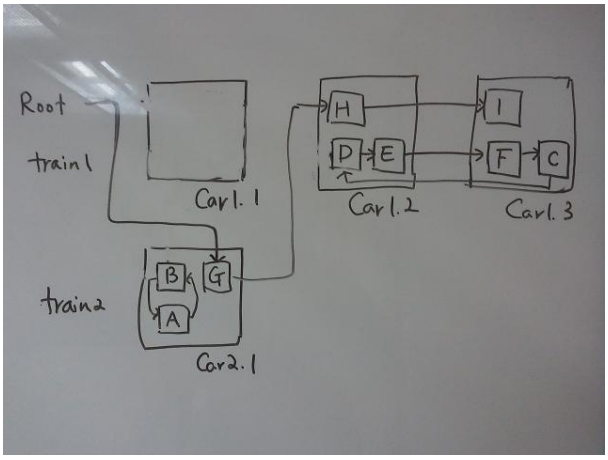


**그림 2.**

그림 2 는 Car 1.2 를 collect 하기 직전의 모습이다.

외부 Train 2 에서 노드 H 를 가리키고 있으므로, 노드 H 를 Train 2 로 이동시킨다. 이때, Car 2.1 은 꽉 차 있어서 더 이상 노드를 받아들일 수 없으므로, 새로운 Car 2.2 를 만들어서 Car 2.2 로 노드 H 를 이동시킨다.

또, 노드 D 를 Car 1.3 에서 가리키고 있다. Car 1.3 도 꽉 차 있으므로, Car 1.4 를 만든 후 이동시킨다. 노드 D 의 자식 노드 E 도 Car 1.4 로 이동시킨다.

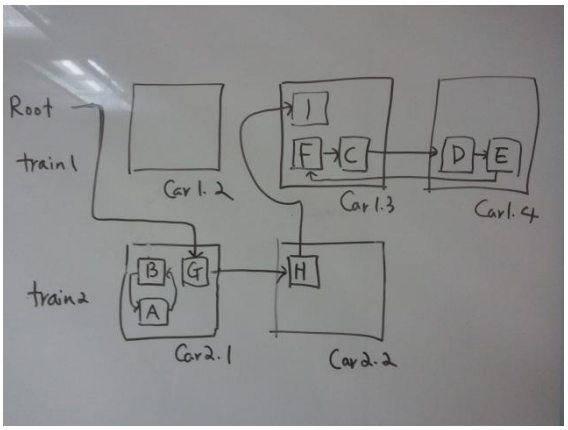


**그림 3.**

그림 3 에서는 Car 1.3 을 collect 한다. 우선 외부 Train 2 에서 가리키는 노드 I 를 이동시킨다. 그 다음 Car 1.4 에서 가리키는 노드 F 를 Car 1.4 로 이동시킨다. 노드 F 의 자식 노드 C 도 이동시킨다. 이번에는 Car 1.4 가 꽉 차 있으므로 Car 1.5 를 만들고 이동시킨다.

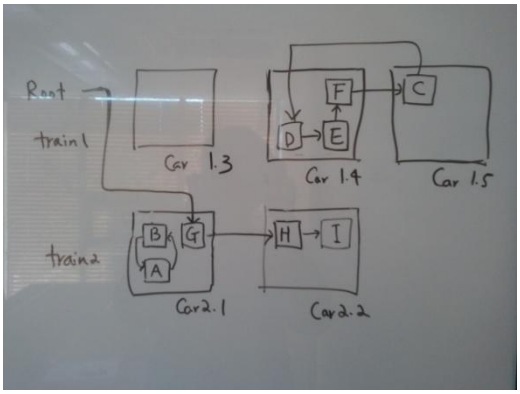


**그림 4.**

이제 Car 1.4 를 처리할 차례이다. 그런데 Train 1 의 외부에서 가리키는 reference 가 하나도 없으므로 train 전체를 반환한다. 즉 Car 1.4 와 Car 1.5 를 통째로 반환한다.

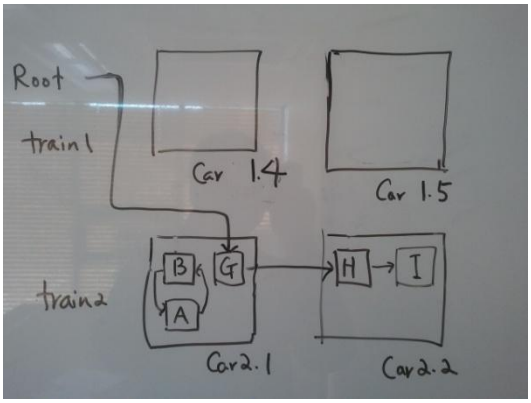


**그림 5.**

그림 5 는 Car 2.1 을 처리한다. 우선 Root 에서 가리키는 노드 G 를 이동시켜야 한다. 이 때 Train 2 가 아닌 다른 외부 train 으로 이동해야 하므로, 새로운 Train 3 을 만들어서 이동한다. 노드 A,B 는 반환된다.

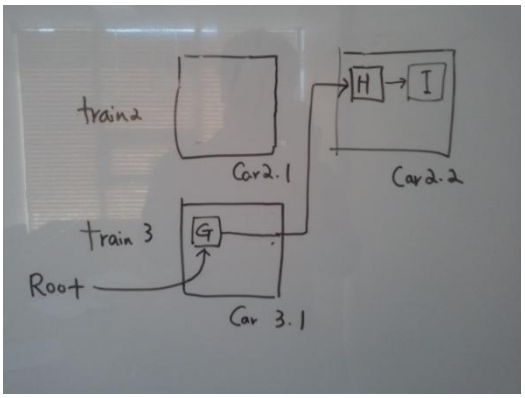


**그림 6.**

그림 6 에서 Car 2.2 를 처리한다. Car 2.2 의 노드 H 는 외부 Train3 에서 가리키므로 Car 3.1 로 이동한다. 노드 H 의 자식 노드 I 도 Car 3.1 로 이동한다.

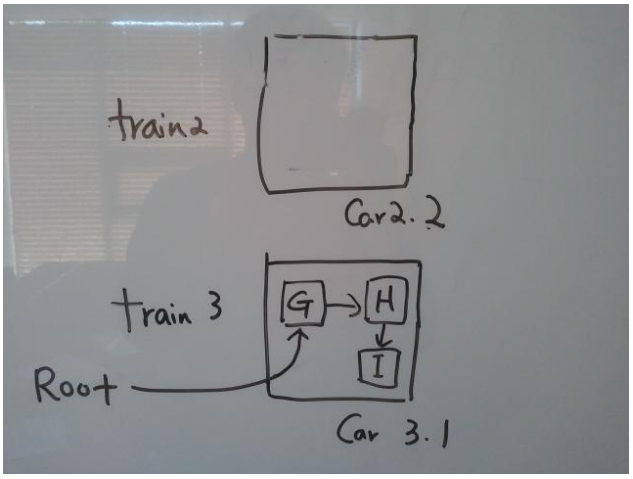


그림 7.

그림 7 은 collect 가 완전히 끝난 후의 모습이다. 이 예제에서는 garbage collect 를 6 번에 나누어 처리한 것을 볼 수 있다. Train 알고리즘에서는 이렇게 나누어 처리함으로써 프로그램이 중단되는 시간을 줄인다.

## FINALIZATION

Java 와 .Net 환경에서는 Finalize 라는 메소드(Method)를 제공한다. 만약, rootset 에서 오브젝트로의 직간접적인 참조가 없으면 Finalize Method 를 호출한다.

Collect 과정에서 오브젝트를 반환해야 할 때, 그 오브젝트가 finalize 메소드를 가지고 있는지 검사한다. 만약, Finalize 메소드를 가지고 있다면, 반환하지 않고 finalize queue 에 그 오브젝트를 추가한다.

그리고, 별도의 finalize 전용 thread 에서 finalize 메소드를 실행한다. Collect 도중에 실행하지 않고 별도의 쓰레드에서 실행하는 이유는 collect 가 프로그램 실행 도중 여러 곳에서 호출될 수 있기 때문이다. lock 을 가지고 있거나, 데이터가 불완전한 상태에서 finalize 메소드가 호출되는 경우를 막기 위함이다.

Finalize 메소드 실행 도중에 새로 참조가 생기는 경우가 있는데 이를 resurrection 이라고 부른다. 예를 들면, 글로벌 변수에서 새로 오브젝트를 참조하는 수가 있다. 오브젝트가 resurrect 되지

않았다면, 다음 collect 할 때 그 오브젝트의 메모리를 반환한다. 그리고, 오브젝트의 finalize 메소드는 한번만 호출된다. 만약, resurrect 된 이후 다시 참조가 없어지면, collect 할 때 finalize 메소드를 호출하지 않고 직접 반환한다.

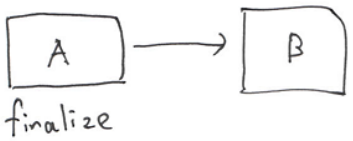


**그림 1.**

그림 1 에서, 오브젝트 A 의 Finalize 메소드가 실행될 때, 오브젝트 B 를 정상적으로 참조하려면, 오브젝트 B 가 반환되지 않았어야 한다. 그래서 collect 에서 finalize queue 에 오브젝트를 추가할 때, 그 오브젝트가 참조하는 오브젝트들도 찾아서 Mark 해야 한다.

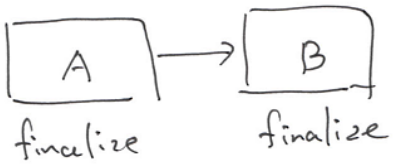


**그림 2.**

만약, 오브젝트 B 도 Finalize 메소드를 가지고 있다면, 좀 더 복잡해진다. 이 경우에는 A 의 Finalize 가 먼저 실행되고, B 의 Finalize 는 나중에 실행되어야 한다. 그래야만, 오브젝트 A 가 오브젝트 B 를 포함하고 있을 때, 정상적으로 A 의 Finalize 를 한다.

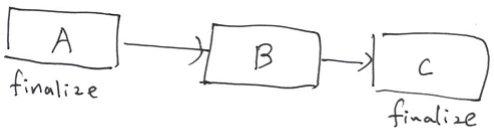


**그림 3.**

마찬가지로 그림 3 에서 오브젝트 A 와 오브젝트 C 가 Finalize 메소드를 가지고 있다면, 오브젝트 A 의 Finalize 가 오브젝트 C 보다 먼저 실행되어야 한다.

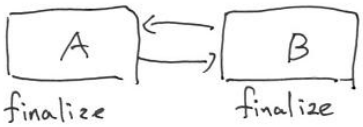


그림 4.

그림 4 처럼 서로 참조하는 경우에는 어떤 순서로 finalize 할 지 정해지지 않는다. 이 때는 어떤 순서로 실행되어도 무방하다.

## CONCURRENT MARK-SWEEP COLLECTOR

Garbage collection 의 인한 프로그램 멈춤을 줄이는 또 하나의 방법은 concurrent 알고리즘을 사용하는 것이다.

Concurrent 알고리즘에서는 응용 프로그램을 그대로 실행하면서 별도의 thread 에서 동시에 garbage collect 를 하는 것이다. 이때, garbage collection 을 하는 쓰레드를 collect 쓰레드라고 하고, 응용 프로그램의 원래 기능을 수행하는 쓰레드를 mutator 쓰레드라고 부른다. Concurrent 방식에서는 mutator 쓰레드와 collect 쓰레드가 동시에 실행된다. 하지만, 이 알고리즘에는 단점이 있어서, 전체적인 성능이 오히려 떨어질 수도 있다. 동시에 garbage collect 를 하려면, 복잡도가 높아지고, 그것 때문에 비효율적인 면이 있기 때문이다.

Concurrent 알고리즘에서는 메모리 노드의 주소를 읽거나 쓸 때 별도의 작업을 해야 한다. Concurrent mark-sweep collector 에서는 write-barrier 라는 것을 이용해 주소 값을 변경(write)할 때마다 별도의 추가 작업을 해야 한다. Concurrent copying collector 알고리즘도 있는데, 이 알고리즘은 read-barrier 를 사용해서 주소 값을 읽을 때도 추가작업을 한다. Mark-sweep 방식에서는 노드의 주소가 바뀌지 않으므로 concurrent mark-sweep collector 가 concurrent copy collector 보다 더 간단하다.

여기에서는 concurrent mark-sweep collector 에 대해서 알아보겠다.

### LOST OBJECT PROBLEM

Garbage Collection 에서 각 노드는 세가지 색깔의 상태로 분류할 수 있다.

- White
  노드의 IsMarked 값이 false 인 노드이다. Collector 가 rootset 으로부터 노드까지의 경로(path)를 찾지 못한 상태이다. Garbage collect 를 시작할 때, 모든 노드는 white 상태에서부터 시작한다. Garbage collect 를 완료했을 때, white 상태의 노드는 garbage 이다.
- Grey
  Collector 가 rootset 으로부터 노드까지의 경로(path)를 찾은 상태이지만, 아직 자식 노드를 처리하지 않고, 스택(GCStack)에서 기다리고 있는 상태이다.
- Black
  자식 노드까지 완료된 상태이다. Black 노드의 자식 노드는 grey 이거나 black 이어야 한다.

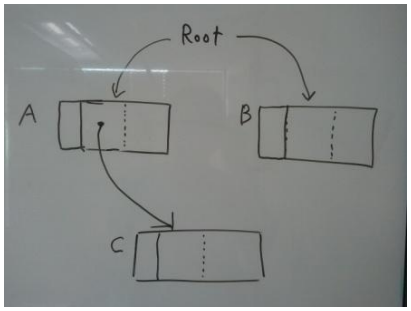


그림 8.

원래 모습이 그림 8 처럼 되어 있었다고 가정하자.

Step1 을 실행하면 그림 9 처럼 바뀌게 된다.

**(step 1)**
**B->Child[0] = A->Child[0]**
**A->Child[0] = NULL**

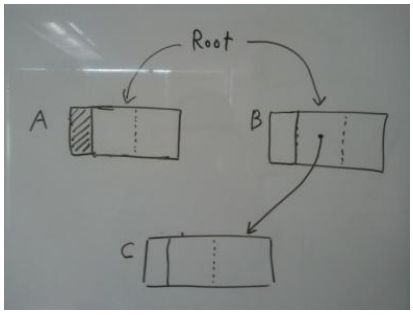


그림 9

또, 이 상태에서 스텝 2 를 실행하면 그림 10 처럼 바뀐다.

**(step 2)**
**A->Child[0] = B->Child[0]**
**B->Child[0] = NULLL**

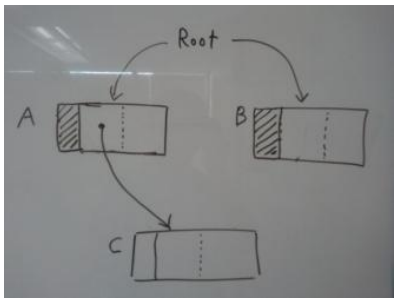


그림 10

Concurrent garbage collector 에서는 mutator 쓰레드와 collect 쓰레드가 동시에 실행되므로, garbage 가 아닌 노드를 free 하지 않도록 특별히 주의해야 한다.

그림 8 을 다시 보자. 이 상태에서는 garbage collector 가 시작되었다고 하자. 노드 A 와 노드 B 둘 다 grey 상태이다. 그 후, mutator 쓰레드가 (step 1)을 실행하고, collector 쓰레드가 노드 A 의 MarkNode()를 실행하였다고 하자.

이제, 그림 9 의 상태가 되었다. 노드 A 의 상태는 black 이고, 노드 B 는 아직 grey 이다. 다음에 mutator 쓰레드에서 (step 2)가 실행된 후, collector 쓰레드에서 노드 B 의 MarkNode()를 실행하였다고 하자.

이제, 그림 10 처럼 된다. Garbage collector 는 모두 완료되었다. 노드 A 와 노드 B 는 black 이지만, 노드 C 는 white 이다. 그래서, 노드 C 가 garbage 인 것으로 잘못 처리되었다. 이렇게, garbage 가 아닌데, garbage 로 잘못 처리되는 문제를 lost object problem 이라고 부른다. Concurrent 알고리즘에서는 collect 쓰레드의 Mark 과정과 mutator 쓰레드의 write 문이 동시에 실행돼서 발생하는 lost object problem 을 피할 수 있게 만들어야 한다.

## WRITE-BARRIER

앞에서 설명한 것처럼 복잡한 lost object problem 를 해결하기 위해서는 write-barrier 를 사용해야 한다. Concurrent collector 에서는 어떤 노드가 garbage 인지 여부가 불확실한 경우가 있다. 이런 경우에는 보수적으로(conservatively) garbage 가 아닌 것으로 간주해야 한다.

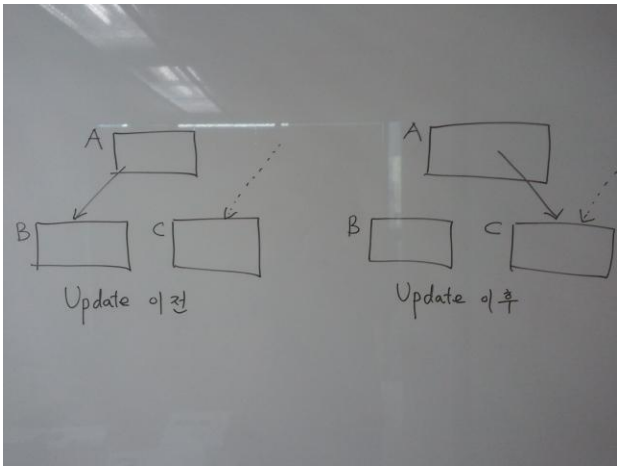


**그림 10.**

A->Child[0] = C 를 실행하기 전과 후의 모습이다. 변수 값을 바뀌기 전에는 B 가 A 의 자식 노드이지만, write 문을 실행하면 자식 노드가 C 로 바뀐다. 이 경우 write-barrier 에서 A(부모 노드), B(신규 자식 노드), C(기존 자식 노드)를 grey 상태로 바꾸면(GCStack 에 추가하면) lost

object problem 을 방지할 수 있다. 사실 셋 중 하나만 해도 된다. 이것에 대해서 자세하게 알아보자.

lost object problem 이 발생하려면 다음 두 가지 조건을 만족해야 한다.

1. white 노드가 black 노드의 자식 노드로 될 때.
   즉, write 문의 부모 노드가 black 상태이고, 신규 자식 노드가 white 상태일 때.
2. write 문이 실행되는 순간에는 어떤 grey 노드에서 white 노드(신규 자식 노드)로의 경로(path)가 존재했다가, 나중에 그 경로가 없어질 때.

두 번째 조건에 대해 좀 더 설명하면, write 문을 실행되는 순간 신규 자식 노드는 살아있는 노드(live node: garbage 가 아닌 노드)여야 한다. 신규 자식 노드가 white 노드라면, 어떤 grey 노드에서 신규 자식 노드로의 path 가 존재해야 한다. 그리고, collection 이 끝났을 때 신규 자식 노드가 white 상태(garbage 노드)로 남으려면 그 path 가 중간에 없어져야만 한다.

그런데, write 문을 실행할 때마다 부모 노드를 grey 상태로 바꾸면 첫 번째 조건을 만족할 수가 없게 된다. 또, write 문을 실행할 때마다 신규 자식 노드를 grey 상태로 바꿔도 첫 번째 조건을 만족할 수 없다.

write 문을 실행할 때마다 기존 자식 노드를 grey 상태로 바꾸면 어떤 grey 노드에서 white 노드로의 경로가 없어지는 것을 방지한다. 그러므로, 두 번째 조건을 만족할 수 없게 만든다. 약간 복잡하니까, 잘 생각해보자. 이것은 이전에 실행된 다른 write 문의 신규 자식 노드가 garbage 로 간주되는 것을 방지한다. 결국, 현재 write 문의 신규 자식 노드가 garbage 로 간주되는 것은 이후에 실행되는 다른 write 문의 기존 자식 노드를 grey 상태로 바꿈으로써 방지된다. 이런 방식을 snapshot-at-the-beginning 방식이라고 한다. 왜냐하면, garbage collection 이 시작되는 시점에 live node 들은 (그 이후에 dead node 가 되더라도) 반환(free)되지 않는 것이 보장되기 때문이다.

결론적으로, write 문의 부모 노드, 신규 자식 노드, 기존 자식 노드 중 하나를 grey 상태로 바꾸면(GCStack 에 추가하면), lost object problem 을 방지할 수 있다. 세가지 방법 모두 garbage collection 의 mark 과정이 종료되는 시점에 live node 들이 반환되지 않는 것이 보장된다.

## STEELE ALGORITHM

write 문의 어떤 노드를 grey 상태로 바꾸느냐에 따라 세 가지 알고리즘으로 분류한다. 여기서는 그 중 부모 노드를 grey 상태로 바꾸는 Steele 알고리즘에 대해 더 자세히 알아보자.

좀 복잡해서 자세히 파악하기가 힘들겠지만, **WriteBarrier**()함수에서 어떤 일을 하는지 대충 알면 Concurrent Garbage Collector 가 어떻게 동작하는지 이해할 수 있을 것이다.

```
enum { MARK_PHASE, SWEEP_PHASE };
enum { WHITE, GREY, BLACK };
struct Node
{
        Node *   Child[...];
        int      Color;
};

stack<Node *> GCStack;
int Phase;
Node *Sweep;


void WriteBarrier(Node *Parent, int Index, Node* NewChild) // Parent->Child[Index]=NewChild
{
        atomic
                Parent->Child[Index] = NewChild
                if Phase == MARK_PHASE
                        if Parent->Color == BLACK &&   NewChild->Color == WHITE
                                Parent->Color = GREY
                                GCStack.push(Parent)
}
```

Steele 알고리즘에서 사용하는 Write-barrier 이다. Garbage 인 노드가 살아있는 것으로 간주되는 경우를 줄여서 free 되는 비율을 높이기 위해, 가능한 한 꼭 필요할 때만 노드 Parent 를 grey 상태로 바꾼다.

다음은 collect 쓰레드에서 실행되는 Mark()와 Sweep() 함수이다. 각각의 Mutator 쓰레드마다 ThreadColor 라는 값를 가지고 있어서, 쓰레드 자체도 GREY 와 BLACK 의 색깔 상태를 가진다.

```
int      ThreadColor[MAX_MUTATOR];

void Mark()
```

```
{
        atomic
                Phase = MARK_PHASE
                for (int T = 0; T <MAX_MUTATOR; ++T)
                        ThreadColor[T] = GREY
        for Node **R in RootSet which is not in thread stack
                atomic
                        MarkNode(*R)
                ProcessStack()
        for (int T = 0; T <MAX_MUTATOR; ++T)
                < suspend thread T >
                atomic
                        ThreadColor[T] = BLACK
                        for Node **S in RootSet which is in thread T's stack
                                MarkNode(*S)
                < resume thread T >
                ProcessStack()
}

void    MarkNode(Node *N)
{
        if N != NULL && N->Color == WHITE
                N->Color = GREY
                GCStack.push(N)
}

void ProcessStack()
{
        Node *N;
        loop
                atomic
                        if GCStack.empty()
                                break
                        Node *N = GCStack.top()
                        GCStack.pop()

                        for Node **C in N->Child
                                MarkNode(*C)
                        N->Color = BLACK
}
```

```
void Sweep()
{
	atomic
		Phase = SWEEP_PHASE
		Sweep = Heap
	loop
		atomic
			if Sweep >= Heap+Size(Heap)
				break
			Node *Next = INT_PTR(Sweep) + Size(Sweep);
			if  Sweep->Color == WHITE
				Free(Sweep);
			else
				Sweep->Color = WHITE
			Sweep = Next
}
```

Mark()함수를 보면, RootSet 를 여러 번 나누어 처리한다. 그 이유는 white 노드를 가능한 한 많이 남겨서, 노드가 free 되는 확률을 높이기 위함이다. 앞에서 설명한 것처럼, 대부분의 노드들은 생성된 지 얼마 안가 garbage 가 되므로, 나중에 검사할수록 garbage 로 처리될 확률이 높다. 특히, 쓰레드 스택에 위치하고 있는 로컬 변수들을 나중에 처리하면 효율적이다.

그리고, 전체 mutator 쓰레드를 동시에 정지시키고 쓰레드 스택의 RootSet 을 처리할 필요가 없다. 로컬 변수의 내용이 바뀌는 경우에 사용하는 특별한 WriteBarrier 가 있기 때문에, 한 쓰레드씩 스택을 처리해도 된다.

다음은 노드의 주소를 로컬 변수에 저장하는 경우에 쓰는 WriteBarrier 이다.

```
int GetCurrentThreadID();

void	WriteBarrierForLocalVar(Node **Var, Node* N)		// (*Var) = N
{
	atomic
		(*Var) = N
		if Phase == MARK_PHASE
			if ThreadColor[GetCurrentThreadID()] == BLACK && N->Color = WHITE
				N->Color = GREY
				GCStack.push(N)
}
```

GetCurrentThreadID()는 현재 Mutator 쓰레드의 번호를 구해준다.

현재 쓰레드의 Color 가 BLACK 이 아니면 collect 쓰레드에서 로컬 변수에 대한 처리가 이루어지지 않은 상태이므로, 나중에 collect 쓰레드에서 노드 N 의 처리를 하도록 놔두는 게 낫다.
그때쯤이면, 노드 N 이 garbage 가 되어 있을 수 있다.

다음은 새 노드를 생성해서 로컬변수에 저장하는 함수이다

```
void NewAndWrite(Node **Var, int ChildSize, Node **Child)  // (*Var) = new Node(ChildSize, Child)
{
        Node *N = Allocate(ChildSize);
        atomic
                if Phase == SWEEP_PHASE
                        N->Color = (Sweep <= N) ? BLACK : WHITE
                else
                        N->Color = ThreadColor[GetCurrentThreadID()] == BLACK
                        ? BLACK : WHITE
                for (int i = 0; i < ChildSize; ++i)
                        N->Child[i] = Child[i]
                        if Phase == MARK_PHASE && Child[i]->Color == WHITE
                                N->Color = WHITE
                WriteBarrierForLocalVar (Var, N)
}
```

NewAndWrite()에서는 가능한 한 white 상태로 만들어서 나중에 free 될 가능성을 높인다.

```
                        N->Color = ThreadColor[GetCurrentThreadID()] == BLACK
                        ? BLACK : WHITE
```

중간에 이 코드는 이해하기 힘들 것이다. Mark()가 거의 끝나가면(Mutator 쓰레드의 Color 가 BLACK 이면) GCStack 에 새로운 노드를 추가하는 것보다는 그냥 black 상태로 만들어서 collect 쓰레드가 Mark()함수를 빨리 끝마치도록 돕는다.

Concurrent 알고리즘이 garbage collection 시에 멈추는 시간을 줄여주기는 하지만, 복잡한 write-barrier(또는 read-barrier) 코드 때문에 전체적인 성능이 떨어지고 실행 파일 크기도 커진다.

위에서는 사용하지 않았지만, card marking 기법을 사용하면 더 효율적으로 구현할 수 있다. Write barrier 에서 card map 에 어떤 노드가 변경되었는지 기록만하고, 나중에 collect 쓰레드가 card

map 을 읽어서 해당 노드를 GCStack 에 추가하는 것이다. 이렇게 하면 write barrier 에서 하는 일이 줄어들어 효과적이다.

또 한가지 개선할 점은 로컬 변수에 노드의 주소를 저장할 때에 write-barrier 를 사용하지 않는 것이다. 로컬 변수의 경우 자주 사용하므로 write-barrier 를 사용하지 않는다면 더 효율적일 수 있다. 그런데 이렇게 하면 쓰레드 스택 검사를 언제 끝내야 하는지 좀 복잡해진다. 이 때는 모든 mutator 쓰레드를 순차적으로 검사해서 그동안 GCStack 에 노드를 추가한 경우가 하나도 없을 때 끝내도록 하면 된다. 이 때는 grey 노드가 하나도 존재하지 않고 신규 노드는 모두 black 상태이다.

```
void    WriteBarrierForLocalVar(Node **Var, Node* N)        // (*Var) = N
{
        (*Var) = N
}

void Mark()
{
        for Node **R in RootSet which is not in thread stack
                atomic
                        MarkNode(*R)
                ProcessStack()
        loop
                bool changed = false;
                for (int T = 0; T <MAX_MUTATOR; ++T)
                        < suspend thread T >
                        atomic
                                ThreadColor[T] = BLACK
                                for Node **S in RootSet which is in thread T's stack
                                        MarkNode(*S)
                        < resume thread T >
                        atomic
                                if GCStack.empty()
                                        continue
                        changed = true
                        ProcessStack()
                if ! changed
                        break
}
```

# CONCURRENT COPYING COLLECTOR

이제 concurrent copying collector 에 대해서 알아보자. Copying collector 에선 노드의 주소가 바뀔 수 있으므로 ReadBarrier()가 필요하다. 실제 프로그램에서 read 가 write 보다 빈번하게 사용되므로, ReadBarrier()를 사용하면 성능상 불리한 점은 있다. 여기에선 간단한 Baker 알고리즘에 대해 알아보겠다.

## BAKER ALGORITHM

Baker 알고리즘은 의외로 간단하다. 가장 중요한 ReadBarrier 부터 살펴보자.

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

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

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 *ReadBarrier(Node *N, Int Index)       // N->Child[Index]
{
        atomic
                Node *C = N->Child[Index]
```

```
        if C != NULL && !Forwarded(C)
                C = CopyNode(C)
        return C
}
```

ReadBarrier() 함수에서는 노드의 주소를 to-space 영역의 주소로 변환한다. 그러므로, mutator 쓰레드에선 항상 to-space 영역의 주소만 존재한다. 즉, mutator 쓰레드의 입장에서 보면, black 또는 grey 상태의 노드들만 볼 수 있다..

다음에 나오는 Collect 함수는 이전에 배웠던 non-concurrent copying collector 와 거의 비슷하다.

```
void Collect()
{
    atomic
        Flip()
        QueueInit()
        for Node **R in RootSet
            *R = CopyNode(*R)
    loop
        atomic
            if QueueEmpty()
                break
            Node *N = QueuePop();
            for Node **C in N->Child
                *C = CopyNode(*C)
}

void Flip()
{
    char *Temp = FromSpace;
    FromSpace = ToSpace
    ToSpace = Temp

    Free = ToSpace
}


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)
}
```

## CONCURRENT MARK-COMPACT COLLECTOR

앞에서 배운 Baker 알고리즘은 read-barrier 를 사용해서 좀 찜찜하다. 이번에는 read-barrier 를 사용하지 않는 compressor 방식에 대해서 알아보자.

### COMPRESSOR ALGORITHM

compressor 방식에서는 virtual memory system 이 read-barrier 의 역할을 대신한다.

```
struct Node
{
        Node *  Child[...];
        Node*   ForwardAddress;
        int     Color;
};

void    Collect()
{
        Flip();
        Mark()
        ComputeAddress()
        Relocate()
}
```

Mark() 함수는 concurrent mark-sweep 방식과 같다.

```
char Heap[HEAP_SIZE]; // HEAP_SIZE is multiple of 2 * PAGE_SIZE
char *ToSpace;
char *FromSpace;
char *ToEnd;
char *FromEnd

void Flip()
{
        char *Temp = FromSpace;
        FromSpace = ToSpace
        ToSpace = Temp

        FromEnd = FromSpace+(ToEnd-ToSpace)
}
```

Flip()함수는 copying collector 와 비슷하다. Mark()는 from-space 영역에서 garbage 인 노드를 구별해 낸다.

```
Node*   PageInfo[HEAP_SIZE/PAGE_SIZE/2];

void    ComputeAddress()
{
        ToEnd = ToSpace
        Node *N = FromSpace;
        int page = 0;
        while N < FromEnd
                if N->Color == BLACK
                        if ToEnd + Size(N) > ToSpace + page * PAGE_SIZE
                                // this page's space is insufficient
                                PageInfo[page] = N
                                ToEnd = ToSpace + page * PAGE_SIZE
                                page = page + 1
                        N->ForwardAddress = ToEnd
                        ToEnd = ToEnd + Size(N)
                N = INT_PTR(N) + Size(N)
}
```

이 방식에서는 to-space 영역을 PAGE_SIZE 크기의 블록으로 나누어 처리한다. PageInfo 는 from-space 에서 to-space 로 노드를 복사할 때, 각각의 페이지에 첫 번째로 복사되는 from-space 의 노드 주소를 나타낸다. .

ComputeAddress()는 non-concurrent mark-compact collector 의 것과 비슷하고, PageInfo 를 계산하는 부분이 추가되었다. 이 단계까지는 mutator 쓰레드에서는 from-space 영역의 주소만 존재한다.

```
void    Relocate()
{
        <uncommit to-space region>
        atomic
                for Node **R in RootSet
                        if *R != 0
                                *R = (*R)->ForwardAddress

        int page = 0
        while ToSpace + page * PAGE_SIZE < ToEnd
                RelocatePage(page)
                page = page + 1
        <commit to-space region>
}

void    RelocatePage(int page)
{
        atomic
                Node *From = PageInfo[page];
                if From == 0
                        return
                PageInfo[page] = 0
                Node *To = ToSpace + page * PAGE_SIZE;
                while From < FromEnd
                        if From->Color == BLACK
                                if INT_PTR(To)+Size(From)
                                > ToSpace + (PageSize+1)*PAGE_SIZE
                                        break
                                memcpy (To, From, Size(From))
                                To->ForwardAddress = 0
```

```
                To->Color = WHITE
                for Node **C in To->Child
                        *C = (*C)->ForwardAddress
                To = INT_PTR(To) + Size(To)
        From = INT_PTR(From) + Size(From)
}
```

Relocate()에서는 우선 mutator 쓰레드에서 to-space 영역을 액세스하지 못하도록 virtual memory system 을 설정한다. 그리고, 글로벌 변수를 저장하는 곳도 액세스하지 못하도록 한다.

그 다음, Rootset 의 내용을 from-space 주소에서 to-space 주소로 바꾼다. 이제 mutator 쓰레드에서는 to-space 영역의 주소만 존재한다. 그러다가, 노드의 내용을 읽으면, access violation 이 발생한다. 이 page fault 를 처리하는 함수에서 해당된 주소의 RelocatePage()를 호출한다. 그 후, mutator 쓰레드에서 그 페이지를 액세스할 수 있게 설정한다.

여기서는 Heap 에 하나의 큰 메모리를 할당하는 것으로 표현했지만, 실제로 구현할 때는 가상 메모리 시스템 상에서 Physical memory 를 할당/반환하게 구현한다. 그래서, 실제 메모리를 page 단위로 compact 한다.