가비지 수집 고급

자바대학교 최적화학과 이원영

7.1 트레이드오프와 탈착형 수집기

가비지 수집기 선정 시 고려해야 할 항목

- **중단 시간**: 중단 길이 또는 기간, 최고 관심사
- **처리율**: 애플리케이션 런타임 대비 GC 시간 %
- **중단 빈도**: 수집기 때문에 애플리케이션이 얼마나 자주 멈추는가?
- **회수 효율**: GC 사이클 당 얼마나 많은 가비지가 수집되는가?
- **중단 일관성**: 중단 시간이 고른 편인가?

7.2 동시 GC 이론

가비지 수집기 선정 시 고려해야 할 항목

- **중단 시간**: 중단 길이 또는 기간, 최고 관심사
- **처리율**: 애플리케이션 런타임 대비 GC 시간 %
- **중단 빈도**: 수집기 때문에 애플리케이션이 얼마나 자주 멈추는가?
- **회수 효율**: GC 사이클 당 얼마나 많은 가비지가 수집되는가?
- **중단 일관성**: 중단 시간이 고른 편인가?

JVM 세이프 포인트란

- 세이프포인트 메커니즘은 뒷부분에 다시 설명

삼색 마킹

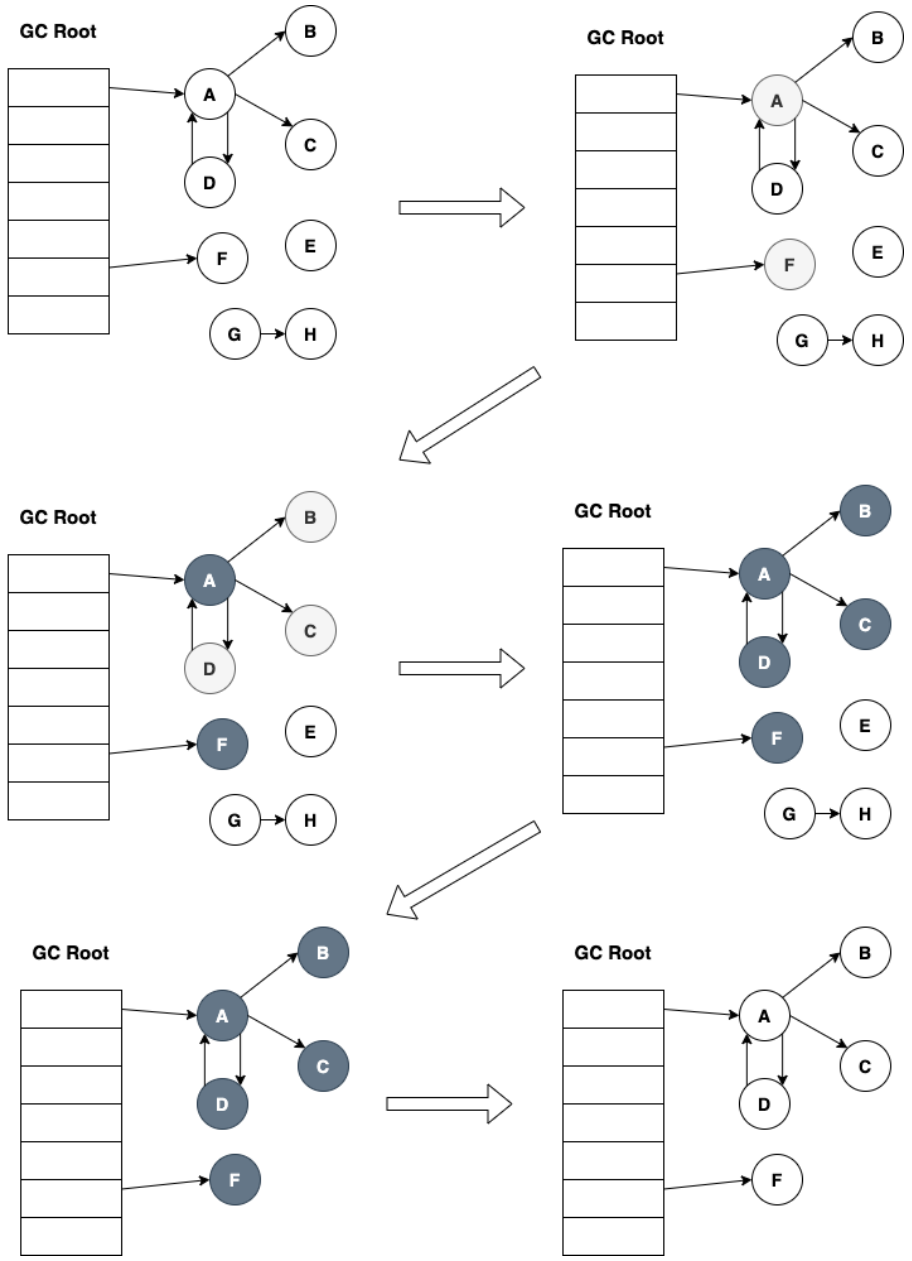
- 작동 원리
  - GC 루트를 흰색 표시한다.
  - 다른 객체는 모두 흰색 표시한다.
  - 마킹 스레드가 회색 노드로 랜덤하게 이동한다.
  - 이동한 노드를 검은색 표시하고 이 노드가 가리키는 모든 흰색 노드를 회색 표시한다.
  - 회색 노드가 하나도 남지 않을 때까지 위 과정을 되풀이한다.
  - 검은색 객체는 모두 접근 가능한 것이므로 살아남는다.
  - 흰색 노드는 더 이상 접근 불가한 객체이므로 수집 대상이 된다.

삼색 마킹

- **흰색**: 아직 검사되지 않은 객체들
- **회색**: 도달했지만, 참조하는 다른 객체들은 모두 검사되지 않은 상태
- **검은색**: 이 객체와 이 객체가 참조하는 모든 다른 객체들이 가비지 컬렉터에 의해 검사 완료된 상태

-> GC 루트에서 DFS 순회 후, 흰색 객체 삭제

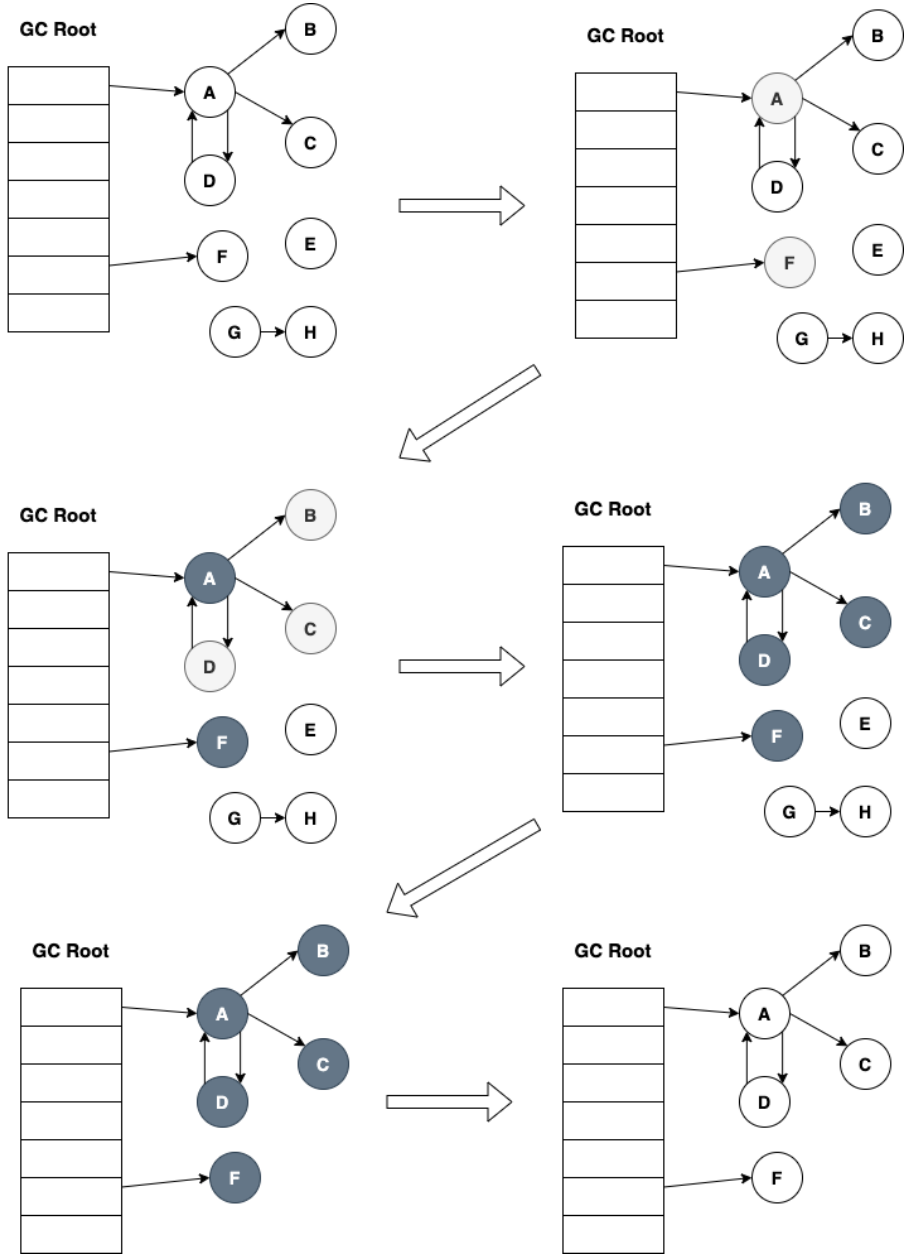
1.로컬 변수들(Local Variables)
2.활성화된 스레드들(Active Threads)
3.정적 필드(Static Fields)
4.JNI(Java Native Interface) 참조들



삼색 마킹

- **SATB(snapshot at the beginning) 기법 활용**
  - DFS 시작 이후에 할당된 객체는 라이브 객체로 간주
  - 변경자 스레드가 수집을 하는 도중에는 **검은색**
  - 수집을 안 하는 동안에는 **흰색**으로 객체 생성

- 문제: 변경자 스레드가 계속 객체 그래프 변경
  - 검은색 객체가 흰색을 참조하는 경우 등
  - -> 이러한 문제를 해결한 수집기들 소개



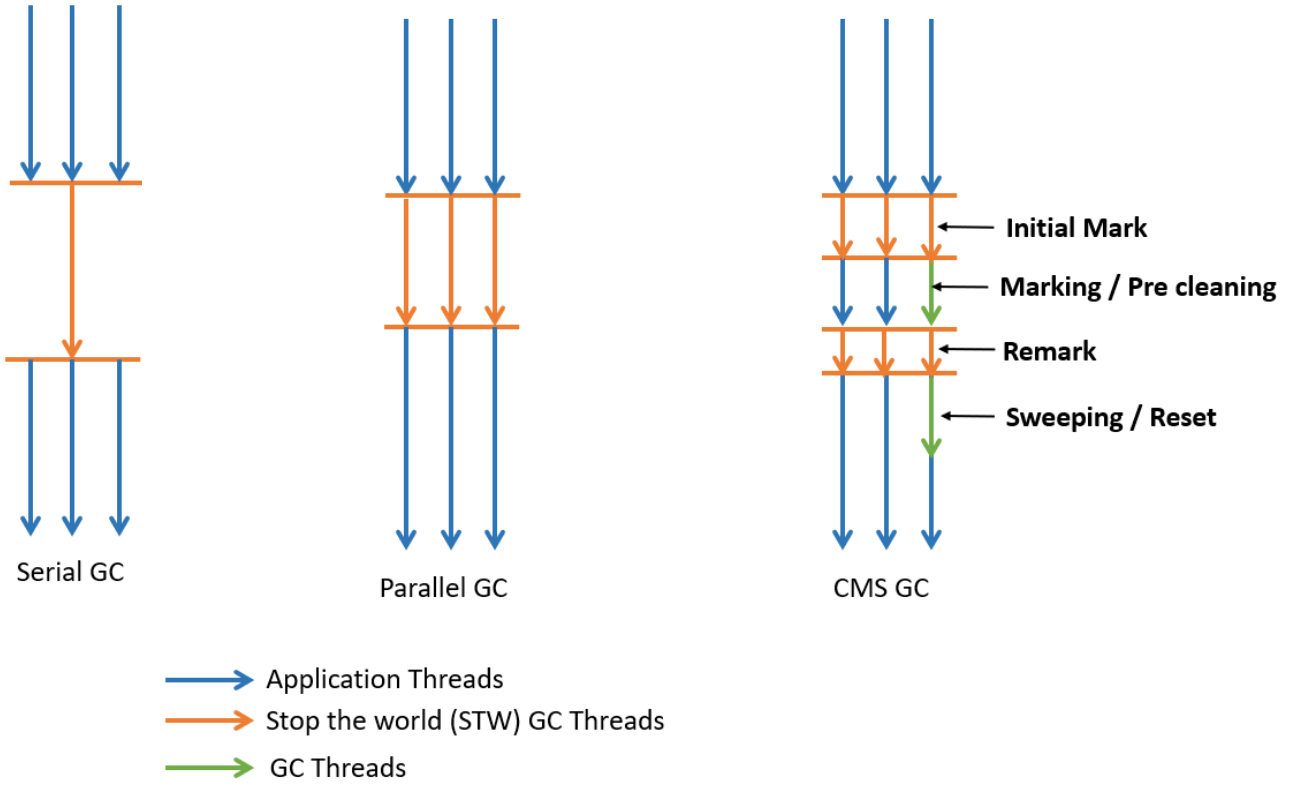
7.3 CMS

CMS

- 목표: STW로 인해 응답하지 못하는 시간 최소화
- 여담: Java 9부터 deprecated 되었고, Java 14에서 drop
- 특징:
  - 테뉴어드(올드) 공간 전용 수집기
  - 보통 영 세대 수집용 병렬 수집기 변형형을 함께 사용

CMS 수행 단계

- 초기 마킹 (STW)
- 동시 마킹
- 동시 사전 정리
- 재마킹 (STW)
- 동시 스위프
- 동시 리셋

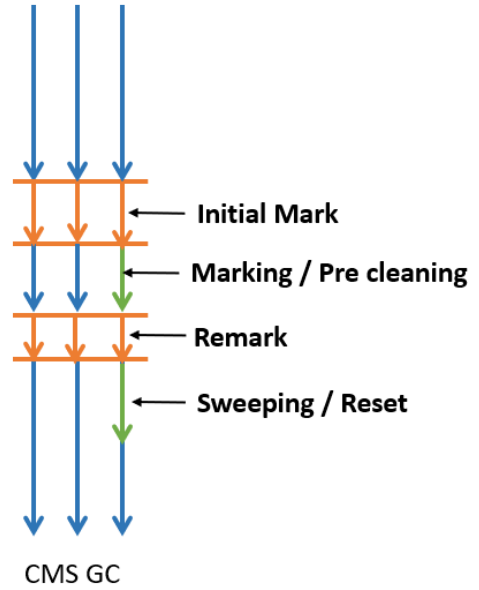


-> 동시: 애플리케이션 스레드와 동시라는 뜻

-> 동시 마킹/ 동시 사전 정리 단계에서 생성된 객체들을 재마킹 단계에서 빠르게 체크

CMS 효험

- 애플리케이션 스레드가 오랫동안 멈추지 않는다.
- 단일 풀 GC 사이클 시간이 더 길다. (= 마킹 시간 총합이 일반보다 길다?)
- CMS GC 사이클이 실행되는 동안, 애플리케이션 처리율은 감소한다.
- GC가 객체를 추적해야 하므로 메모리를 더 많이 쓴다.
- GC 수행에 훨씬 더 많은 CPU 시간이 필요하다.
- CMS는 힙을 압착하지 않으므로 테뉴어드 영역은 단편화될 수 있다.



동시 모드 실패 (CMF)

- CMS 도중에도 객체가 생성됨 -> 생성된 객체는 에덴으로!
- 에덴이 꽉차면? 실행중단 -> 영 GC 시작
- CMS 특성 상 *극히* 일부 객체만 테뉴어드로 승격

- 그래도 테뉴어드에 공간이 부족하면? -> 동시 모드 실패
- -> JVM은 어쩔 수 없이 ParallelOld GC 수집 시작 (풀 SWT)

- 공간있지만 연속된 공간이 없으면(단편화)? -> 동시 모드 실패
- -> 또또 ParallelOld GC ㅠㅠ
- -> 이 문제는 내부적으로 **프리 리스트**로 빈 공간 관리로 최대한 보완

7.4 G1

G1

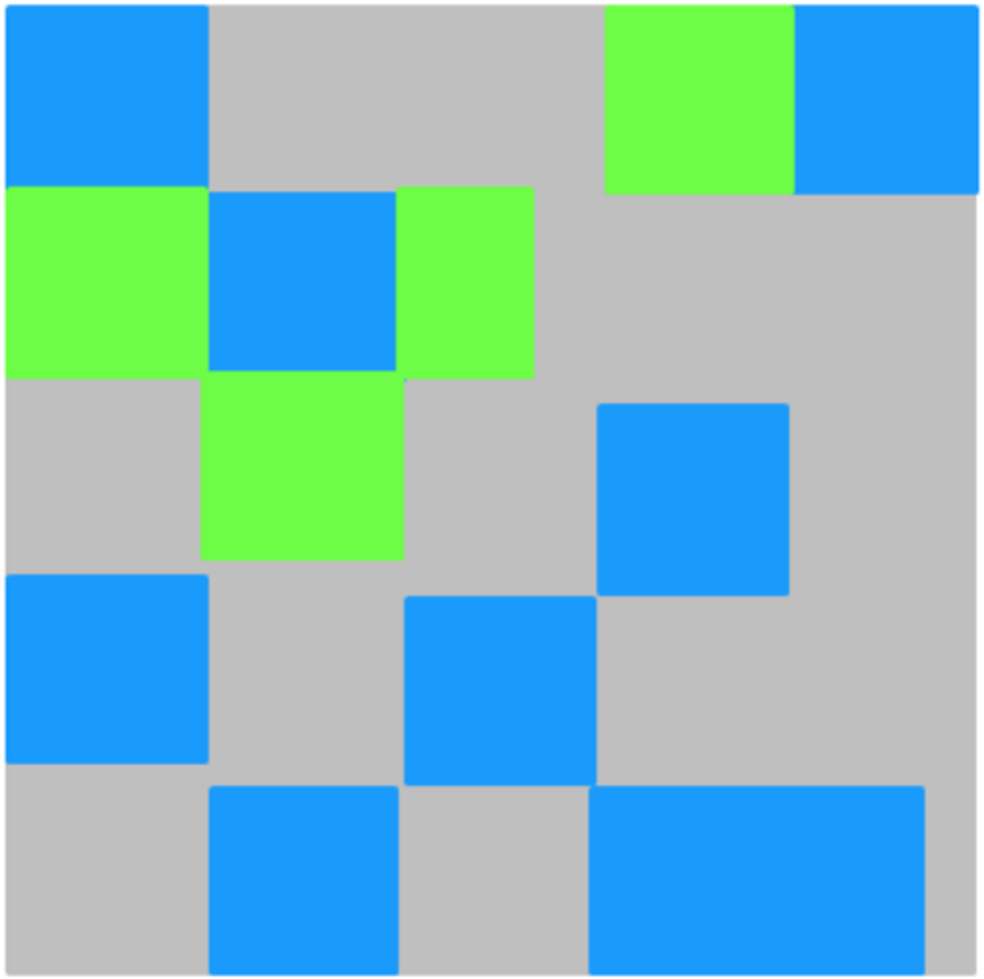
- 목표: 중단 시간이 짧도록
- 특징:
  - CMS보다 훨씬 튜닝하기 쉽다.
  - 조기 승격에 덜 취약하다.
  - 대용량 힙에서 확장성이 우수한다.
  - 풀 STW 수집을 없앨 수(또는 풀 STW 수집으로 되돌아갈 일을 확 줄일 수) 있다.

G1 수행단계

- (영 세대 수집)
- 초기 마킹 (STW)
- 동시 루트 탐색
- 동시 마킹
- 재마킹 (STW)
- 정리 (STW)

- **(영 세대 수집)**
- 초기 마킹 (STW)
- 동시 루트 탐색
- 동시 마킹
- 재마킹 (STW)
- 정리 (STW)

Young Generation in G1

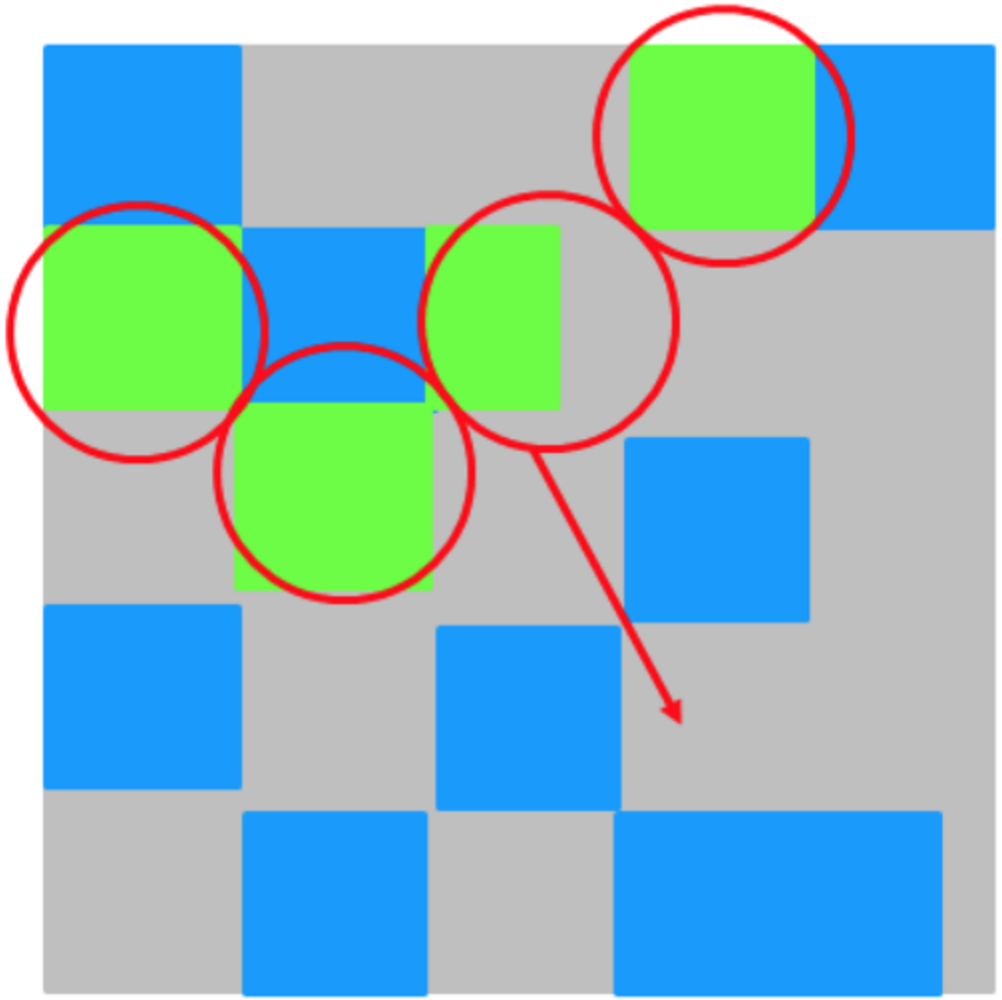


- 연속되지 않은 메모리 공간에 **영 세대**가 **영역** 단위로 메모리에 할당
- 영 세대: 에덴 + 서바이버

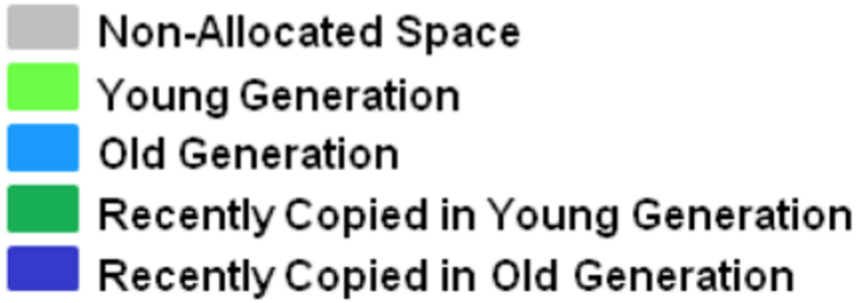
Non-Allocated Space
Young Generation
Old Generation
Recently Copied in Young Generation
Recently Copied in Old Generation

- (영 세대 수집)
- 초기 마킹 (STW)
- 동시 루트 탐색
- 동시 마킹
- 재마킹 (STW)
- 정리 (STW)

A Young GC in G1

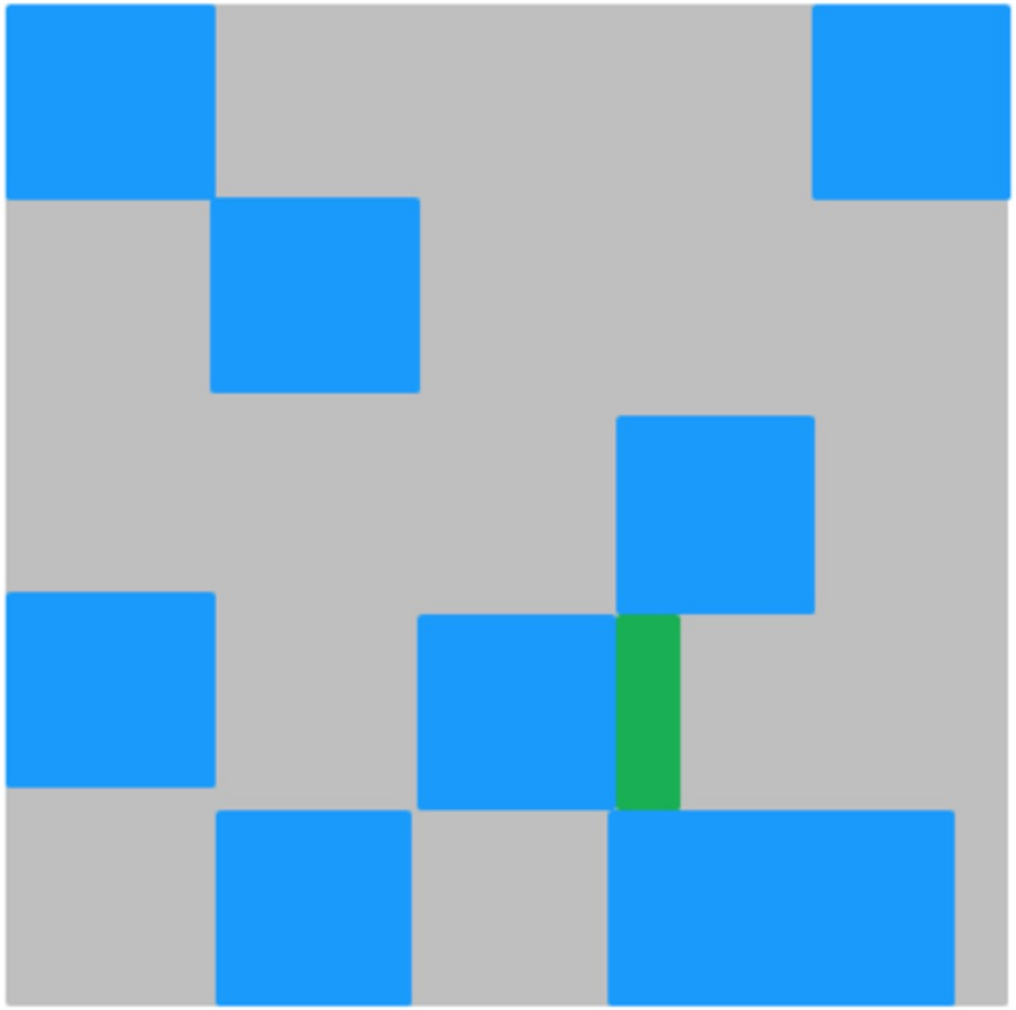


- **영 세대**에 있는 유효 객체를 서바이버 영역이나 올드 세대로 이동 (SWT)
- 에덴 영역과 서바이버 영역 재계산

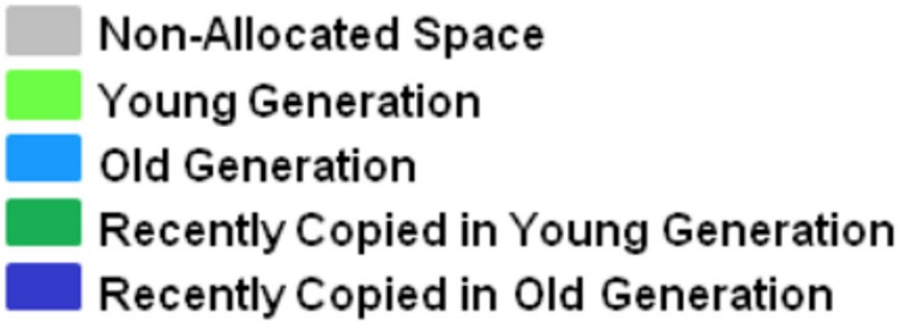


End of Young GC with G1

- (영 세대 수집)
- 초기 마킹 (STW)
- 동시 루트 탐색
- 동시 마킹
- 재마킹 (STW)
- 정리 (STW)

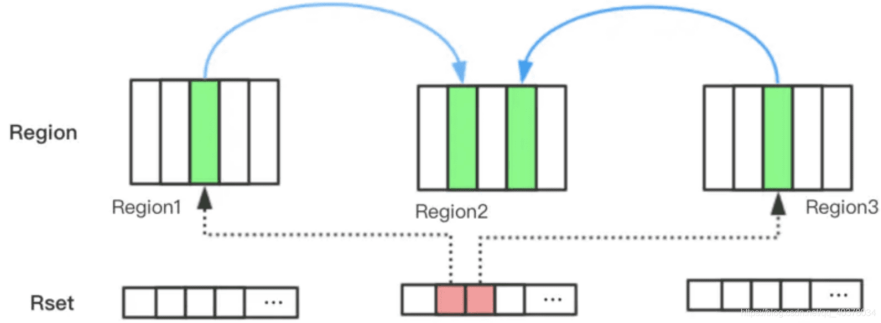


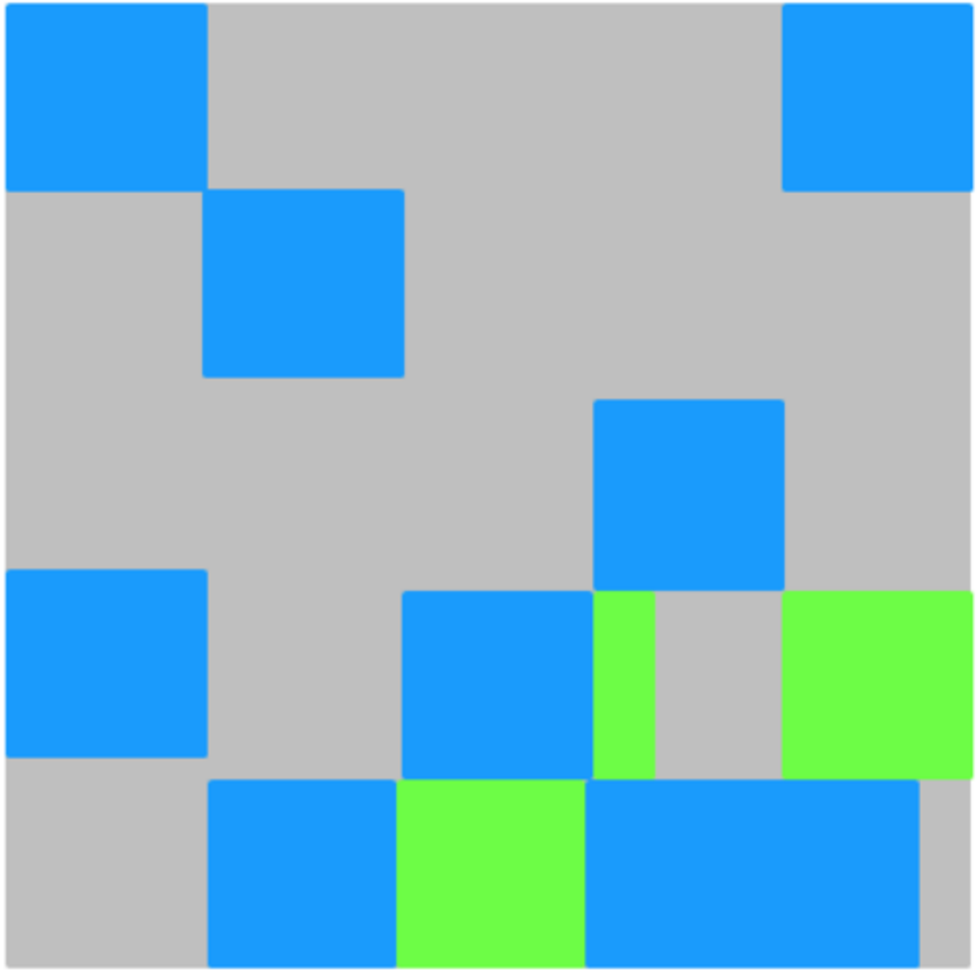
- 영 세대 수집 완료 모습
- **초록색** 부분은 [에덴->서바이버] 혹은 [서바이버->서바이버]로 이동한 객체



- (영 세대 수집)
- **초기 마킹 (STW)**
- 동시 루트 탐색
- 동시 마킹
- 재마킹 (STW)
- 정리 (STW)

Initial Marking Phase

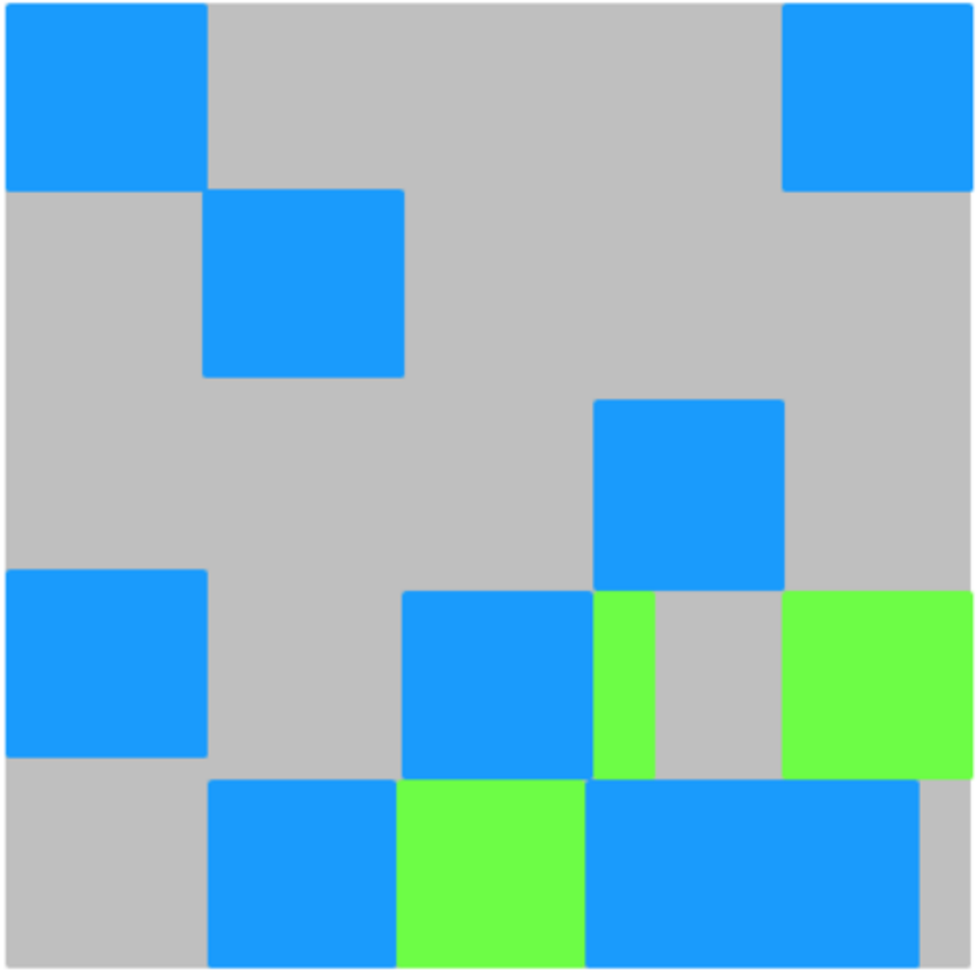




- 서바이버 영역 객체에서 참조하고 있는 올드 영역 객체 파악하고 마킹
- RSet를 활용하여 전체를 탐색할 필요 X
- 서바이버에 대해 의존적이기 STW 발생

Non-Allocated Space
Young Generation
Old Generation
Recently Copied in Young Generation
Recently Copied in Old Generation

- (영 세대 수집)
- 초기 마킹 (STW)
- **동시 루트 탐색**
- 동시 마킹
- 재마킹 (STW)
- 정리 (STW)

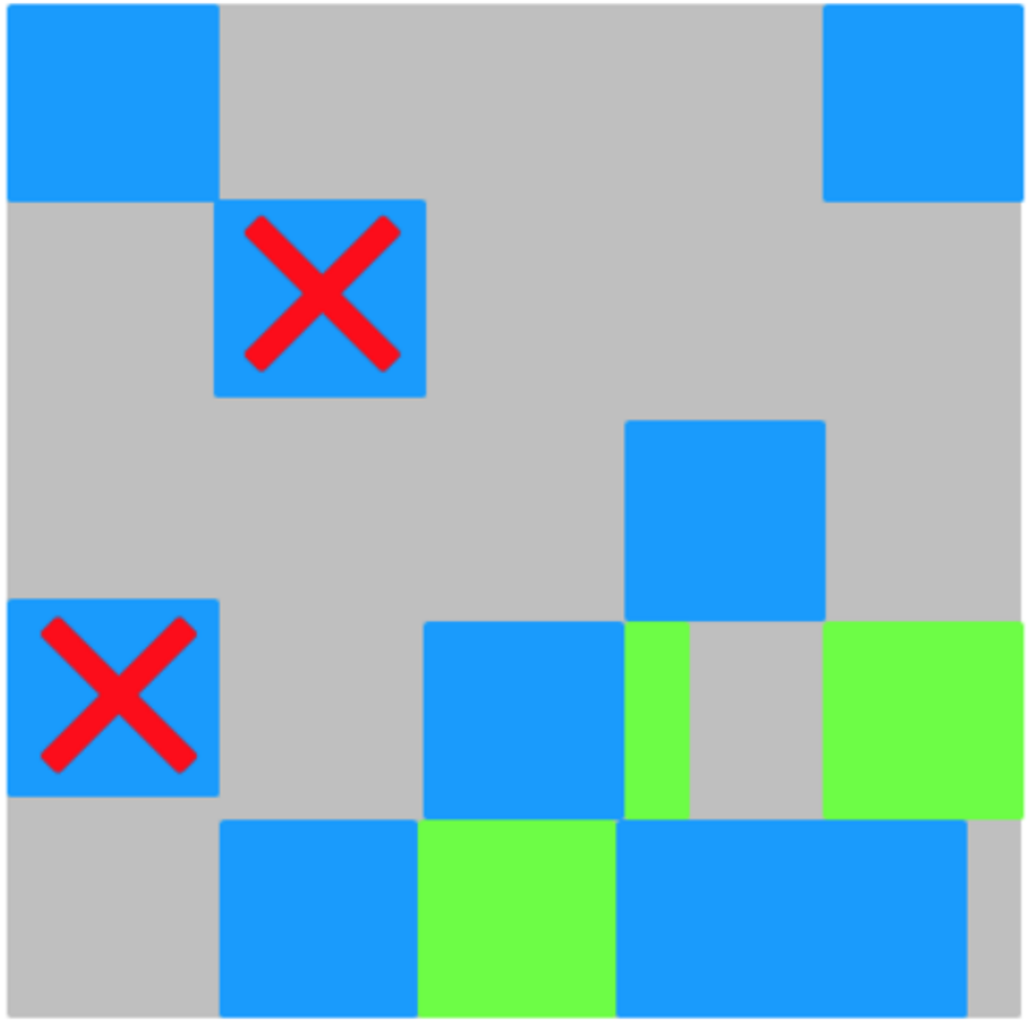


- 올드 영역에서 참조된 객체 마킹 및 루트 지정
- 멀티 스레드로 동작

Non-Allocated Space
Young Generation
Old Generation
Recently Copied in Young Generation
Recently Copied in Old Generation

Concurrent Marking Phase

- (영 세대 수집)
- 초기 마킹 (STW)
- 동시 루트 탐색
- **동시 마킹**
- 재마킹 (STW)
- 정리 (STW)

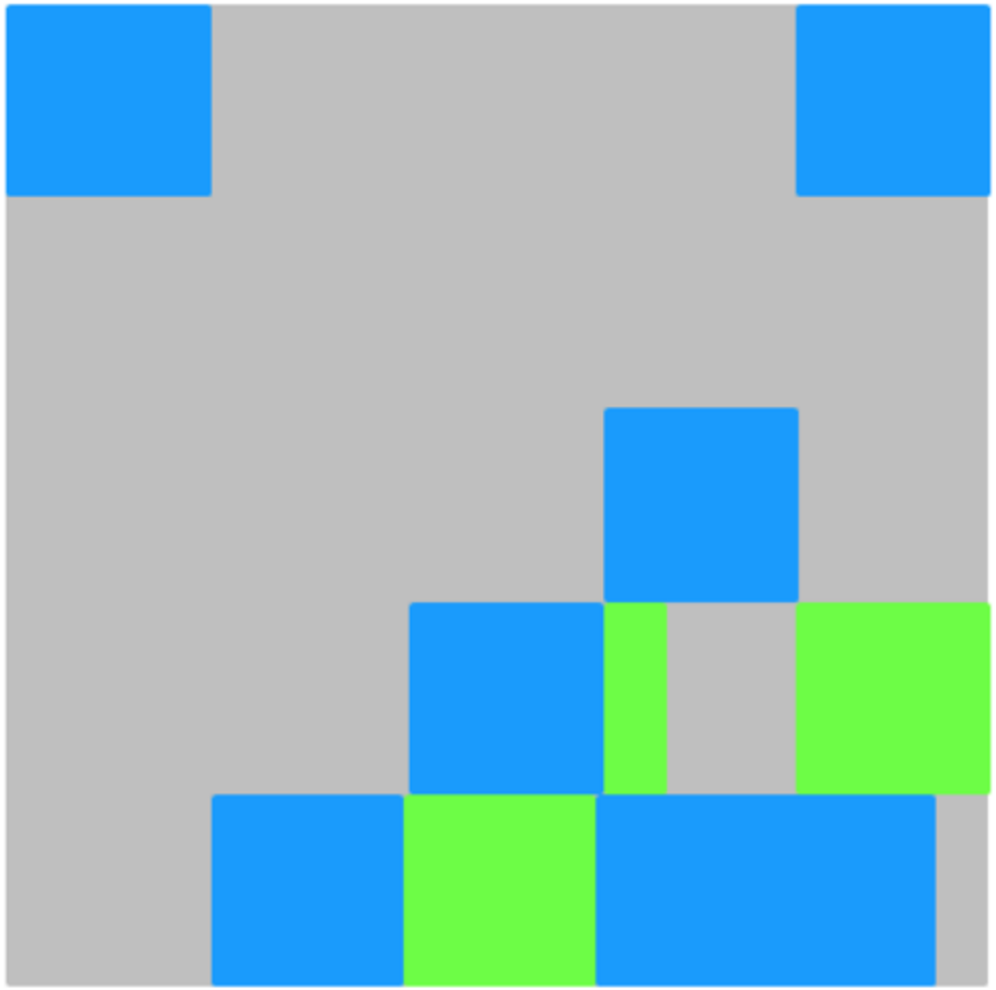


- 올드 세대에 있는 모든 객체 마킹
- 동시에 진행됨
- x 표시는 영역의 모든 객체가 가비지 상태인 영역

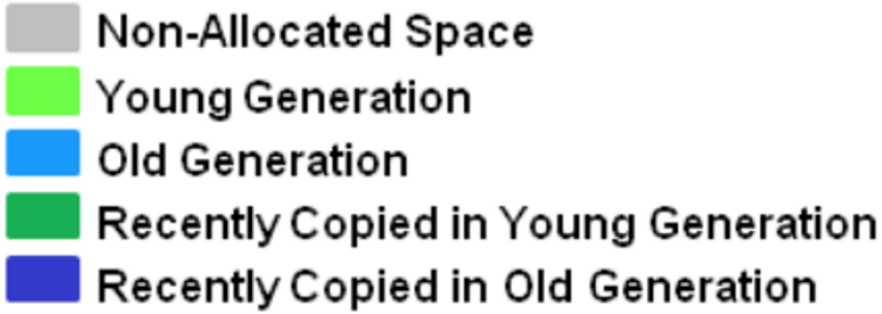
Non-Allocated Space
Young Generation
Old Generation
Recently Copied in Young Generation
Recently Copied in Old Generation

- (영 세대 수집)
- 초기 마킹 (STW)
- 동시 루트 탐색
- 동시 마킹
- **재마킹 (STW)**
- 정리 (STW)

Remark Phase

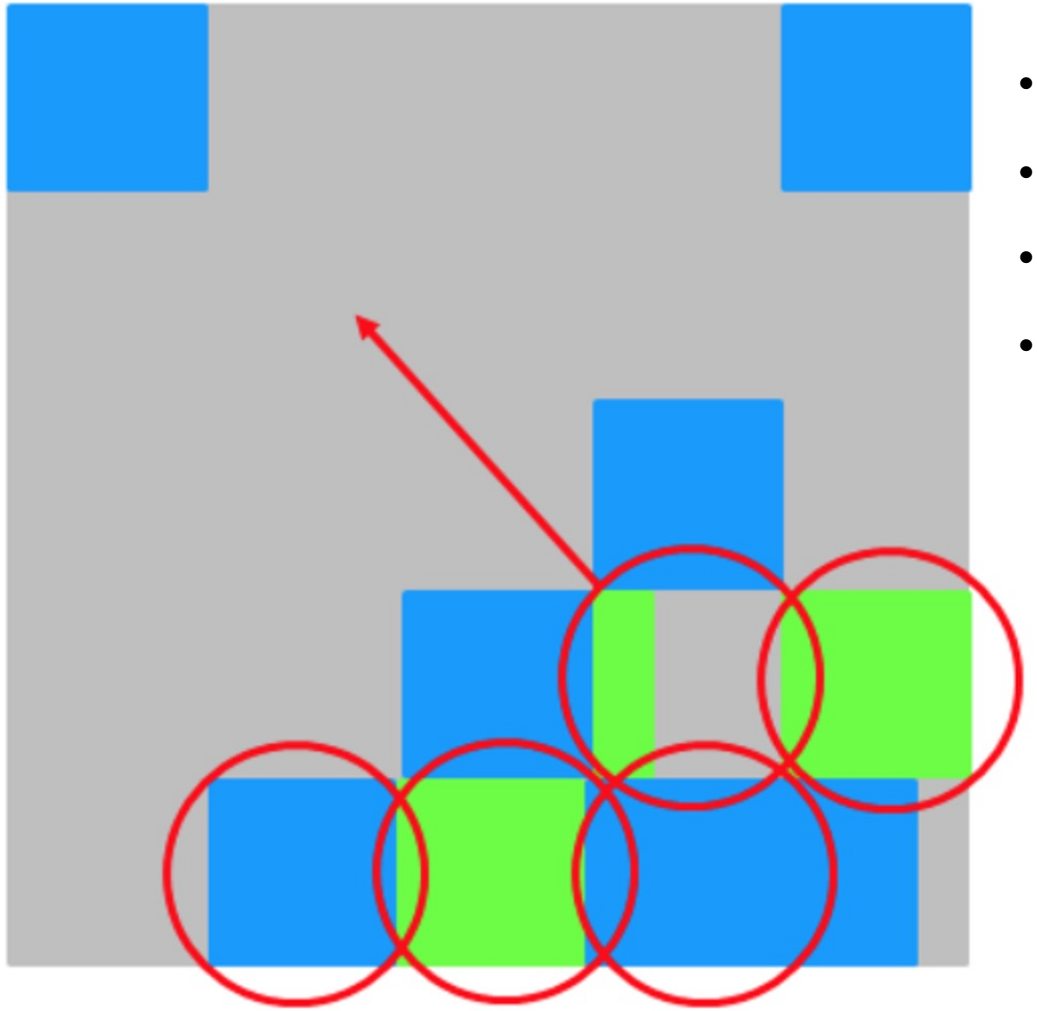


- 아까 x표시 영역 바로 회수 -> STW
- 이전 단계에서 작업하던 마킹을 이어서 작업 -> 완전히 종료



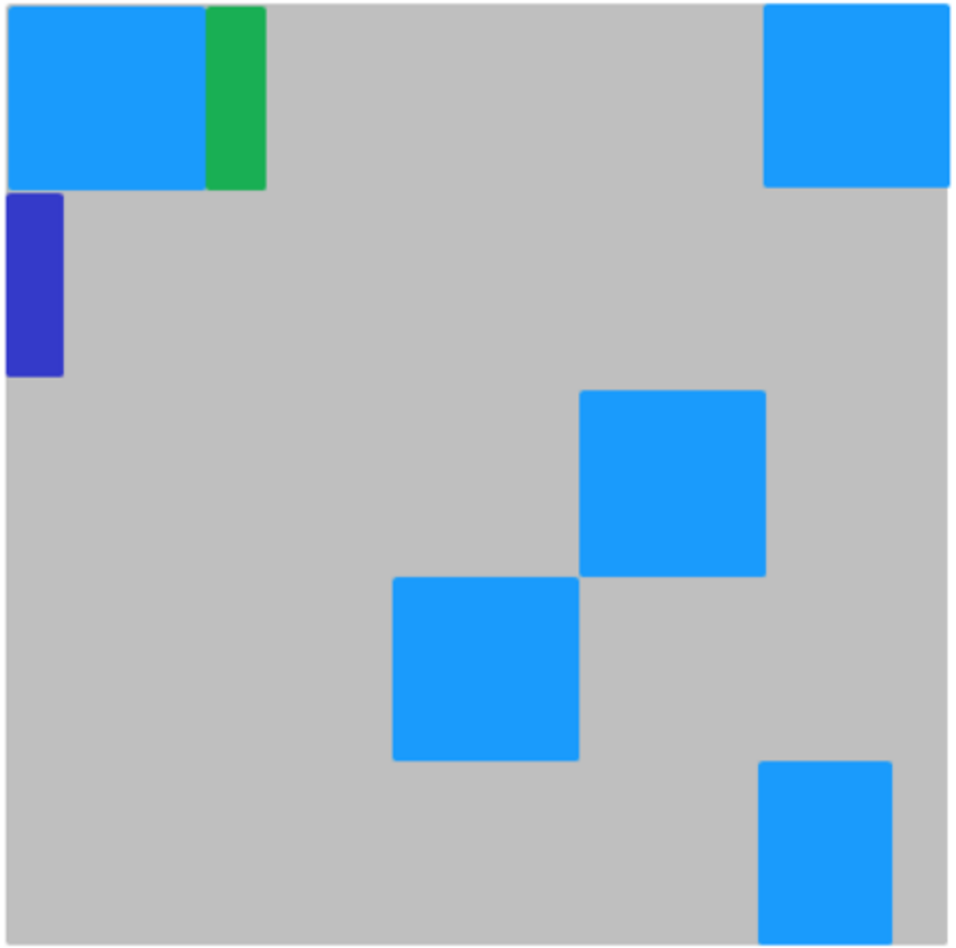
- (영 세대 수집)
- 초기 마킹 (STW)
- 동시 루트 탐색
- 동시 마킹
- 재마킹 (STW)
- 정리 (STW)

Copying/Cleanup Phase



- 라이브 객체의 비율이 낮은 영역 순으로 순차적 수거 -> STW
- 라이브 객체를 다른 영역으로 이동
- 모두 이동한 영역은 바로 제거 (가비지 수집 우선 First)
- = 신속한 공간 확보

After Copying/Cleanup Phase



깔끔하게 제거 & **압착** 완료!
-> 단편화 걱정 NO NO

Non-Allocated Space
Young Generation
Old Generation
Recently Copied in Young Generation
Recently Copied in Old Generation

7.5 셰난도아

셰난도아

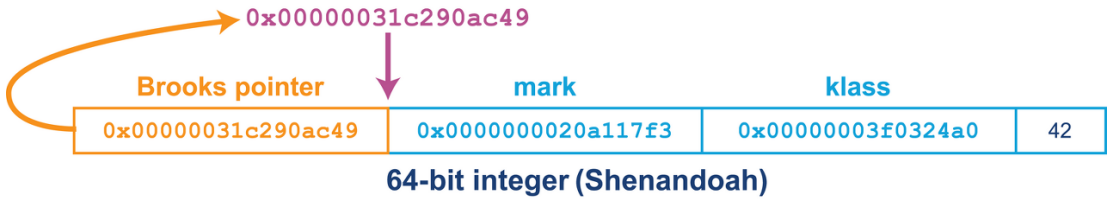
- 목표: 중단 시간 단축
- 해결: 동시 압착!
- 특징:
  - 아직 많이 안 쓰지만 유망하다~

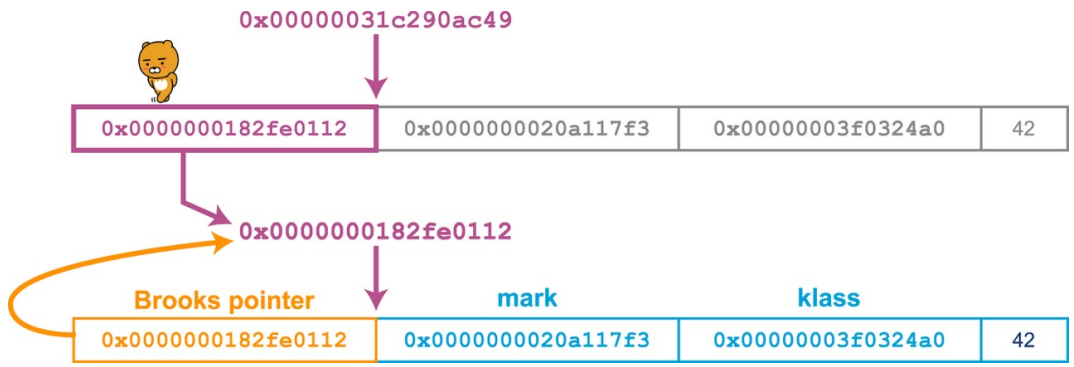
브룩스 포인터

- 객체에 워드 추가해서 새 버전 객체 콘텐츠 주소 추가

브룩스 포인터

- 객체에 워드 추가해서 새 버전 객체 콘텐츠 주소 추가
  - 초기 상태 (재배치 전) : 그냥 메모리 다음 워드
  - 재배치 후: 새 객체 위치의 주소





셰난도아 수행 단계

- 초기 마킹 (STW)
- 동시 마킹
- 최종 마킹
- **동시 압착**
  - 객체를 TLAB로 복사한다.
  - CAS로 브룩스 포인터가 추측성 사본을 가리키도록 수정한다.
  - 이 작업이 성공하면 압착 스레드가 승리한 것으로, 이후 이 버전의 객체는 모두 브룩스 포인터를 경유해서 액세스하게 된다
  - 이 작업이 실패하면 압착 스레드가 실패한 것으로, 추측성 사본을 원상복구하고 승리한 스레드가 남긴 브룩스 포인터를 따라간다

  -> 누가 먼저 이정표를 놓았는가? 가리킨 곳 보니까 맞으면 내가 먼저 놓은거니 다 나를 따르고, 아니면 다른 애가 놓은 이정표 따라야징

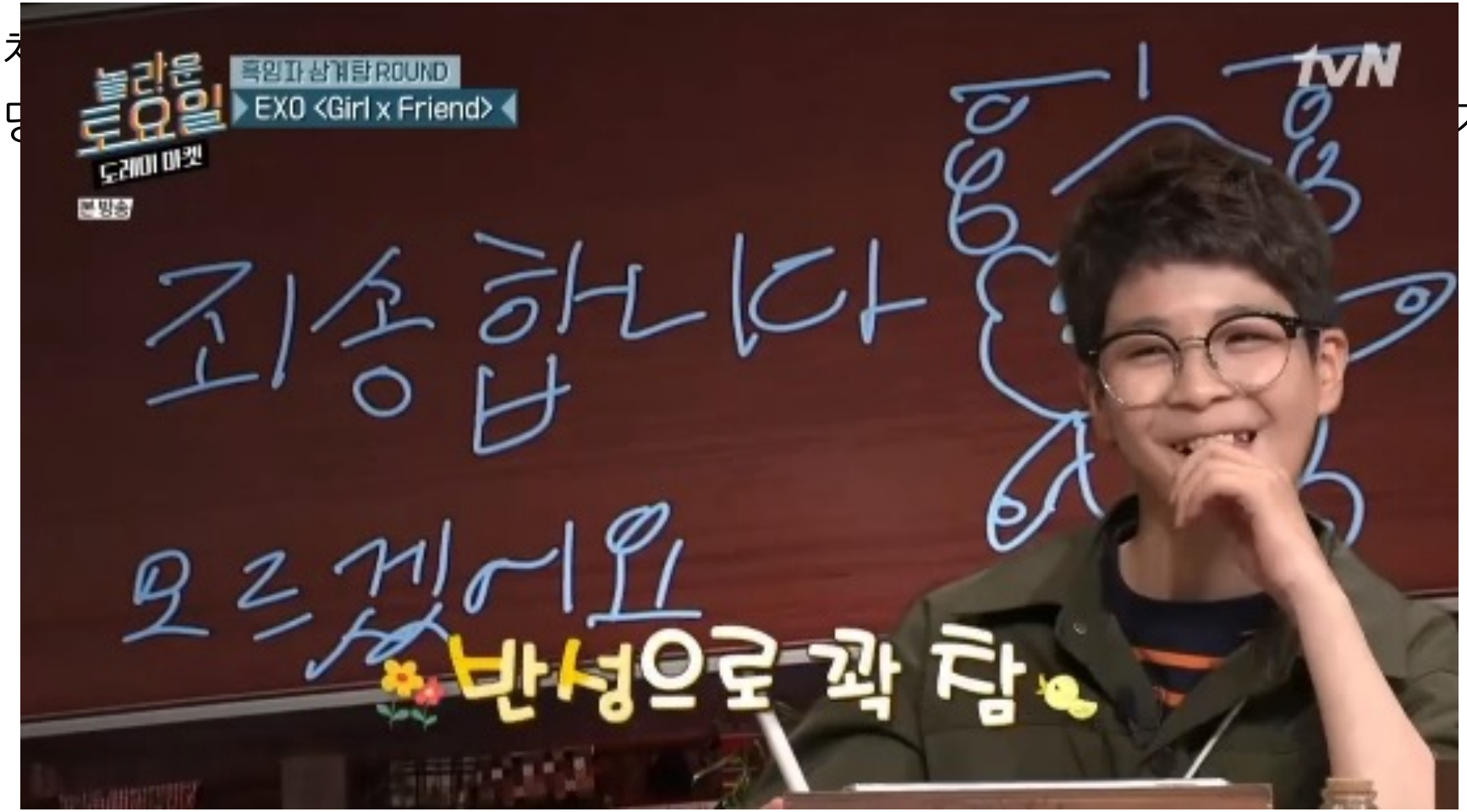
7.6 C4(아줄 징)

C4 (아줄 징)

- 특징:
  - 셰난도아처럼 **동시 압착 알고리즘**을 사용하지만, 브룩스 포인터 대신 **로드-값 배리어** 활용

로드-값 배리어(Load Value Barrier, LVB)

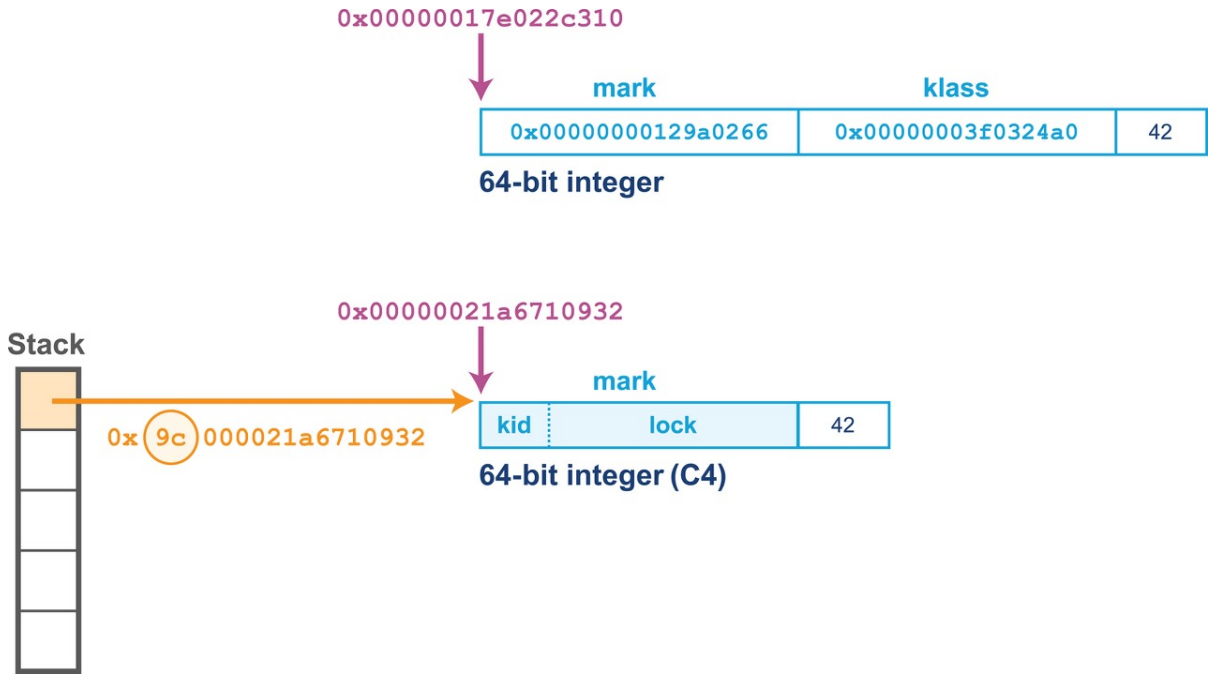
- 어떤 객체
- 징은 해당



져온다.

로드-값 배리어(Load Value Barrier, LVB)

- 어떤 객체를 참조해야 할 때,다른 스레드가 해당 객체를 재배치하고 있다면,
- 징은 해당 객체가 재배치될 때까지 기다리다가, 안전하게 해당 객체의 최신 참조를 가져온다.
- **메모리에서 데이터를 읽어올 때, 그 데이터가 최신의 정확한 데이터인지를 보장하기 위해 사용되는 기술**



C4 (아줄 징) 수행 단계

- 마킹
- 재배치 -> **교대 압착** (로드-값 배리어를 통해 안전하게 이동 및 압착 가능)
- 재매핑 -> 객체의 주소가 실시간으로 업데이트

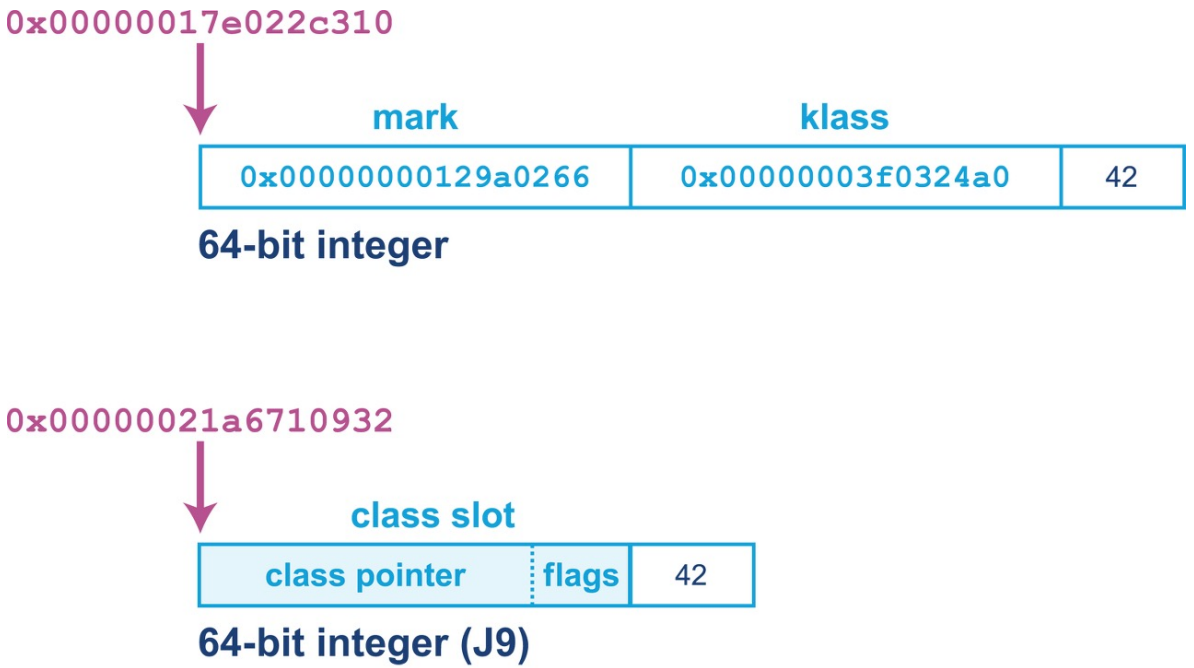
7.7 밸런스드(IBM J9)

밸런스드 (IBM J9)

- J9: IBM에서 제작한 JVM
- 밸런스드: J9의 수집기 중 하나로 영역 기반
- 목표:
  - 대용량 자바 힙에서 중단 시간이 길어지는 현상을 개선한다.
  - 중단 시간이 최악인 경우를 최소화한다.
  - 불균일 기억 장치 액세스 성능을 인지하여 활용한다.

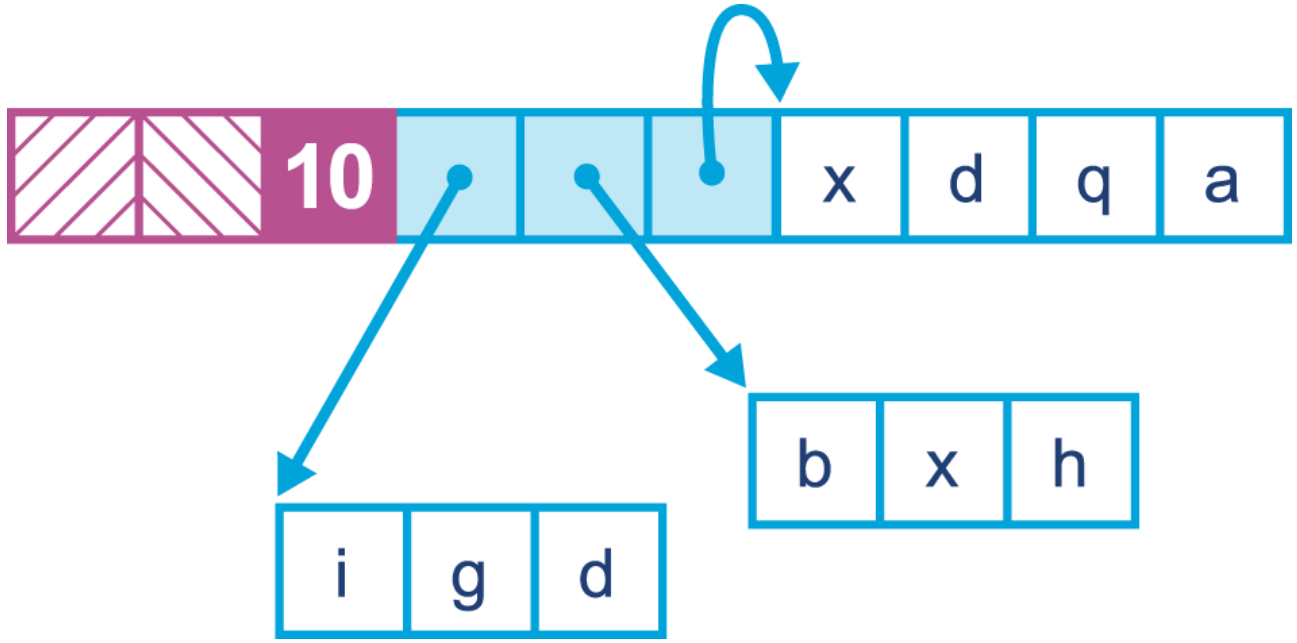
J9 객체 레이아웃

- 기본 헤더에 클래스 슬롯
- 클래스 포인터: 클래스 구조를 가리키는 포인터
- 오프-힙 메모리로 효율적으로 공간 관리



밸런스드 큰 배열 처리하기

- 어레이릿: 불연속되 여러 덩이에 큰 배열을 할당할 수 있는 구조
- 여러 영역에 걸치면 배열 참조할 때 오버헤드가 발생하지만, **평균 중간 시간 감소**



NUMA와 밸런스드

- NUMA: 프로세스와 가까운 메모리를 노드로 묶는 설계 방법

-> 로컬 메모리(같은 노드에 속한)에 액세스할 때가 가장 빠르다.

- 밸런스드는 노드 별로 자바 힙을 분리할 수 있으니

-> 최대한 같은 노드에 객체의 할당/수집이 일어나도록 해서 성능을 좋게 하도록~!

7.8 레거시 핫스팟 수집기

레거시는~

참고

- https://azderica.github.io/til/docs/java/optimizing-java/ch7/#c4%EC%95%84%EC%A4%84-%EC%A7%95
- https://steady-coding.tistory.com/590
- https://mangkyu.tistory.com/119
- https://unialgames.tistory.com/entry/CSharpMemoryBarrier

고맙습니다