**CS60203: Design Optimization of Computing Systems** *August 15, 2024*

# Garbage Collection

*Instructor: Mainack Mondal* *Scribe-By: Somya Kumar*

## 1 Mark and Sweep GC

Mark and Sweep is a tracing garbage collection (GC) algorithm used in memory management to automatically recycle unused memory in programs. It processes which objects are reachable by a chain of references from certain "root" objects, considers the rest as "garbage," and collects them.

### 1.1 The Algorithm

The Mark and Sweep algorithm works in two primary phases: the marking phase and the sweeping phase. Each addressable memory location has a mark bit associated, which is always unset except during the collection phase. The algorithm suspends program execution when it runs and operates as follows:

#### 1.1.1 Marking Phase

The first stage is the mark phase, where the algorithm performs a tree traversal of the entire 'root set' and marks each object referenced by a root as 'in use.' Subsequently, it marks all objects referenced by these initially marked objects, continuing this process recursively.

#### 1.1.2 Sweeping Phase

In this phase, the entire memory is scanned from start to finish, and all blocks are checked, whether free or in use. Blocks not marked as 'in use' are considered unreachable from any roots, and their memory is freed. The mark for the objects marked as in-use is cleared, preparing them for the next cycle.

### 1.2 Pros and Cons of Mark and Sweep GC

#### 1.2.1 Pros

- Correctly handles cyclic data structures
- Less memory overhead as it requires only one bit per memory cell

#### 1.2.2 Cons

- Suspends normal program execution
- The entire working memory must be examined, much of it twice, potentially causing problems in paged memory systems.
- The heap may become fragmented over time, leading to cache misses, page thrashing, and complicated memory allocation.

# 2 Generational GC

The concept behind generational garbage collection is to partition memory cells into different groups (*Generations*) and prioritize running the GC more frequently on one of these groups (*Younger Generation*) compared to others. Based on the empirical observation, the most recently created objects will likely become unreachable quickly (known as *infant mortality*).

## 2.1 Generational GC in Python

Python's heap memory is divided into three distinct generations:

- **Generation 0 (Young Generation):** Newly allocated objects are placed in Generation 0.
- **Generation 1 (Middle Generation):** Objects that survive one or more collections in Generation 0 are promoted to Generation 1.
- **Generation 2 (Old Generation):** Objects that persist through multiple collections in Generation 1 are further promoted to Generation 2.

To decide when to run, the collector keeps track of the number of object allocations and deallocations since the last collection. The collection starts when the number of allocations minus the number of deallocations exceeds *threshold_0*. Initially, only Generation 0 is examined. If Generation 0 has been examined more than *threshold_1* times since Generation 1 has been examined, then Generation 1 is also examined.

GC Threshold Count and Object Creation in Python

```
>>> import gc

>>> gc.get_threshold()
(700, 10, 10)

>>> gc.set_threshold(500, 20, 20)

>>> gc.get_threshold()
(500, 20, 20)

>>> gc.get_count()
(0, 0, 0)

# Create an object
>>> a = [1] * 1000

# Force a garbage collection to see the effect
>>> gc.collect()

>>> gc.get_count()
(1, 0, 0)
```

Example of Generational GC in Python

```
>>> import gc
>>> class MyObj:
...         pass
...

# Move everything to the last generation so it's easier to inspect
# the younger generations.

>>> gc.collect()
0

# Create a reference cycle.

>>> x = MyObj()
>>> x.self = x

# Initially, the object is in the youngest generation.

>>> gc.get_objects(generation=0)
[..., <__main__.MyObj object at 0x7fbcc12a3400>, ...]

# After a collection of the youngest generation, the object
# moves to the next generation.

>>> gc.collect(generation=0)
0
>>> gc.get_objects(generation=0)
[]
>>> gc.get_objects(generation=1)
[..., <__main__.MyObj object at 0x7fbcc12a3400>, ...]
```

# 3 GC in Java

## 3.1 Serial GC

- Uses a Single thread to perform all Garbage Collection work.
- Default on certain hardware and operating system configurations, or can be explicitly enabled with the option `-XX:+UseSerialGC`.
- **Useful in:**
  - Applications running on single-core processors.
  - Environments with small heap sizes where low pause times are critical.

- **Cons:**
  - Not suitable for multi-core processors, as it doesn't take advantage of parallelism.
  - Can lead to longer garbage collection pauses in applications with large heap sizes.

### 3.2 Parallel GC

- Uses multiple threads to perform Garbage Collection work; through *Stop the World* approach
- Default Garbage Collector on many server-class machines can be explicitly enabled with the option `-XX:+UseParallelGC`.
- **Useful in:**
  - Applications running on multi-core processors
  - When throughput is preferable over latency, like large data processing apps
- **Cons:**
  - May lead to longer pause times than more advanced collectors, like the G1 GC or Z GC.
  - Less effective in reducing latency-sensitive pause times in real-time applications.

### 3.3 CMS (Concurrent Mark and Sweep) GC

- Uses multiple threads to perform most of the GC, but unlike Parallel GC, it does it concurrently with the application threads
- Can be explicitly enabled with the option `-XX:+UseConcMarkSweepGC`.
- **Useful in:**
  - Latency-sensitive applications where low pause times are critical.
  - Environments with large heap sizes requiring minimal disruption to application performance.
- **Cons:**
  - Requires more CPU resources than simpler collectors like the Serial GC.
  - May lead to heap fragmentation, potentially triggering Full GC cycles, which can cause long pauses.

### 3.4 G1 GC

- Implements Generational GC using multiple threads
- Default Garbage Collector in many recent Java versions can be explicitly enabled with the option `-XX:+UseG1GC`.
- **Useful in:**
  - Applications requiring predictable pause times while maintaining high throughput.
  - Environments with large heap sizes where balancing latency and throughput is essential.
- **Cons:**
  - More complex than simpler collectors, requiring more tuning to achieve optimal performance.
  - May incur higher overhead in terms of CPU and memory usage compared to collectors like the Parallel GC.

### 3.5 Epsilon GC

- A No-Op Garbage Collector that allocates memory but does not perform any Garbage Collection.
- Can be explicitly enabled with the option `-XX:+UseEpsilonGC`.
- **Useful in:**
  - Testing, benchmarking, and performance experiments where garbage collection behavior should be excluded.
  - Short-lived applications where memory usage is not expected to reach its limits.
- **Cons:**
  - No memory reclamation, leading to eventual out-of-memory errors as the heap fills up.

### 3.6 Shenandoah GC

- Generational GC, but runs concurrently with the application threads
- Can be explicitly enabled with the option `-XX:+UseShenandoahGC`.
- **Useful in:**
  - Latency-sensitive applications where minimizing pause times is crucial.
  - Environments with large heap sizes require efficient, concurrent garbage collection.
- **Cons:**
  - Higher CPU and memory overhead compared to simpler collectors like the Parallel or Serial GC.

### 3.7 Z GC

- A low-latency GC that performs most of its work concurrently. Designed to maintain extremely short pause times regardless of heap size.
- Can be explicitly enabled with the option `-XX:+UseZGC`.
- **Useful in:**
  - Latency-critical applications where sub-millisecond pause times are required.
  - Environments with very large heap sizes, often in the range of terabytes, that demand efficient, scalable garbage collection.
- **Cons:**
  - Higher CPU and memory overhead compared to simpler collectors like Parallel or Serial GC.
  - Limited support in older Java versions and requires a 64-bit system with large amounts of RAM.

| GC Type | Heap Size Support | Pause Times | Throughput | Performance | Application | CPU Overhead |
|---|---|---|---|---|---|---|
| **Serial GC** | Small to medium | Longer | Low | Lower | Single-threaded applications, Development environments | Low |
| **Parallel GC** | Medium to large | Moderate | High | Higher | Batch processing, Scientific computing, Data analysis | Moderate |
| **CMS GC** | Medium to large | Moderate | Moderate | Moderate | Web applications, Medium-sized enterprise systems | Moderate |
| **G1 GC** | Medium to large | Short to medium | High | Higher | Mixed workloads, Large enterprise systems | Moderate to High |
| **Z GC** | Large | Very Short | High | Very High | Latency-sensitive applications, Large-scale systems | Low to moderate |
| **Shenandoah GC** | Medium to large | Very Short | High | Very High | Low-latency applications, Large-scale systems | Low to Moderate |

Figure 1: Comparison between available Collectors in Java

| Criteria | Comments |
|---|---|
| Heap Size | G1, ZGC and Shenandoah can handle large heap size |
| Pause time | For strict pause time requirements, use G1, CMS, ZGC or Shenandoah |
| Throughput | Parallel GC for maximizing throughput |
| Memory Overhead | ZGC and Shenandoah can consume more memory |
| CPU overhead | ZGC and Shenandoah consume more memory than others |

Figure 2: Choosing the right Collector

# References

[1] M. Mondal, *Garbage Collection Part 2*, Lecture Slides, Design Optimization of Computer Systems (CS60203).

[2] Generational GC in Python, Python Developer's Guide.

[3] Java Garbage Collectors, their working and comparisons, Medium Article.

[4] Java Garbage Collectors, Oracle Documentation.

[5] Java Garbage Collection Basics, Oracle Documentation.