Report on Lua’s GC

A brief overview

Lua’s garbage collector (GC) is a system designed to manage memory automatically by reclaiming memory occupied by objects that are no longer in use. Lua uses a tri-color marking scheme for its garbage collection process, a method that helps efficiently identify which objects in memory are no longer needed.

Tri-Color Marking Scheme

The tri-color marking scheme in Lua's GC involves three colors to represent the state of objects during the garbage collection cycle:

1. **White:** Represents objects that are candidates for garbage collection. Objects start out white, indicating that they are not reachable or have not been processed. Lua uses two shades of white to differentiate between objects from the previous and current collection cycles.
2. **Gray:** Represents objects that are reachable and have been marked for scanning, but their children (objects referenced by them) have not yet been fully processed.
3. **Black:** Represents objects that are reachable and have been fully processed, including their children. These objects are considered live and will not be collected.

The garbage collection cycle involves the following phases:

- **Mark Phase:** The GC starts by marking all root objects (objects that are directly accessible from the program) as gray. The process continues by marking objects reachable from the gray objects. Once an object and all its children are fully processed, it is marked black. Any object that remains white at the end of this phase is not reachable and is eligible for collection.
- **Sweep Phase:** In this phase, the garbage collector sweeps through all objects and collects those that are still marked white, as these objects are no longer reachable. The collector also handles objects that are in the "old white" state from the previous cycle, ensuring they are correctly processed.

Lua's garbage collector can operate in three different modes: **Full GC**, **Incremental GC**, and **Generational GC.** Each mode offers different strategies for managing memory but underlying method is same - mark and sweep.

Full Garbage Collection

Full GC is a mode where the garbage collector stops the entire program (a process known as "stop-the-world"), performs a complete mark-and-sweep cycle, and then compresses the memory if necessary.

- **Characteristics:**
  - **Stop-the-World:** The program execution is halted entirely during the GC cycle.
  - **Single Generation:** Full GC operates on a single heap space without differentiating between younger and older objects.
  - **Usage:** This mode is thorough and ensures all unreachable objects are collected. However, it can introduce significant pauses, especially in programs with large amounts of memory in use.
- **Process:** The `luaC_fullgc` function initiates the full garbage collection process. It first checks if a GC cycle is already in progress. If not, it proceeds with the full mark-and-sweep operation, regardless of whether the GC type is incremental or generational.

Incremental Garbage Collection

Incremental GC is designed to minimize the impact of garbage collection on program execution by dividing the work of the mark-and-sweep phases into smaller increments. Instead of stopping the world for a long period, the GC performs short, more frequent pauses, allowing the program to continue running between them.

- **Characteristics:**
  - **Incremental Collection:** The GC performs a small amount of work each time it is triggered, reducing the length of each pause.
  - **Three Control Parameters:** The incremental GC is controlled by three parameters: `pause`, `step multiplier`, and `step size`.
    - **Pause:** Determines how long the GC should wait before starting a new cycle after completing the previous

one. It is defined as a percentage of the total memory in use; if memory usage grows to a certain percentage (determined by the pause parameter) above its previous level, a new GC cycle begins.
        - **Step Multiplier:** Adjusts the amount of work the GC does in each incremental step, effectively controlling how much time the GC spends relative to program execution.
        - **Step Size:** Determines the size of each step in the GC cycle.
- **Usage:** Incremental GC is suitable for applications where minimizing pause times is more critical than optimizing overall memory usage.

Generational Garbage Collection

Generational GC leverages the observation that most objects die young—meaning that many objects become unreachable shortly after they are created. This mode divides the heap into two or more generations: typically, a **young generation** for newly created objects and an **old generation** for objects that have survived several GC cycles.

- **Characteristics:**
    - **Young and Old Generations:** The heap is divided into a young generation (where most allocations happen) and an old generation (where objects that survive multiple collections are promoted).
    - **Minor and Major Collections:**
        - **Minor Collection:** Focuses on collecting garbage in the young generation. It is triggered frequently and is relatively quick because it only scans a small part of the heap.
        - **Major Collection:** Involves collecting garbage from both the young and old generations. It occurs less frequently and usually after a minor collection fails to reclaim enough memory or when the old generation grows beyond a certain threshold.
- **Parameters:**
    - **Major Multiplier (`genmajormul`):** Defines the threshold for triggering a major collection based on memory growth relative to the previous cycle.
    - **Minor Multiplier (`genminormul`):** Defines the frequency of minor collections based on memory usage increase.

    - **Default Values:** By default, `genminormul` is set to 20, and `genmajormul` is set to 100. These values determine how aggressively the GC will collect garbage from the young and old generations.
- **Usage:** Generational GC is efficient for applications with a large number of short-lived objects, as it minimizes the work required to manage memory by focusing on collecting recently allocated objects.

Summary

Lua's garbage collection system provides flexibility with three different modes, each suited to different types of applications and usage patterns. Full GC offers a thorough but potentially disruptive collection process. Incremental GC provides a way to reduce pauses by breaking the collection into smaller steps. Generational GC optimizes for the common case of short-lived objects, improving efficiency for many real-world programs.

On a side note ...

There is a mention of weak table and finalizers in the source code and the docs, so here we go

Finalizers

Finalizers allow Lua to clean up resources that are not automatically managed by the garbage collector. If an object has a finalizer, Lua will run this function when the object is about to be collected. This gives developers a chance to perform any necessary custom cleanup actions, such as saving data, logging messages, or releasing external resources, just before the object is destroyed. Finalizers extend the lifecycle of an object slightly. When an object with a finalizer is ready to be collected, Lua first runs the finalizer. Only after the finalizer has run does Lua actually collect the object. This ensures that cleanup tasks are performed in a controlled manner. Lua's garbage collector knows to look for objects with finalizers during the collection cycle. It places these objects in a special list and processes them differently. This is all handled by Lua internally, ensuring that finalizers are called at the right time.

Weak Tables

Weak tables in Lua are tables that do not prevent their keys or values from being collected by the garbage collector. This means if an object is only referenced by a weak table, it can still be collected when no other strong references exist. Normally, if we store an object in a table, the table holds a strong reference to that object, preventing it from being collected. A weak table, however, allows the garbage collector to collect objects that are only referenced by that table. This prevents memory leaks because objects can be collected when they are no longer needed elsewhere, even if they are still in the weak table. When the garbage collector runs, it automatically removes any entries in weak tables where the key or value has been collected. This means weak tables are automatically cleaned up, reducing the memory footprint and ensuring that they do not hold onto dead objects unnecessarily.