Lua's Garbage Collector

Introduction

Datta Ksheeraj(21CS30037)

Lua is a lightweight, high-level programming language designed primarily for embedded systems and client-server applications. One of its core components is its garbage collector (GC), a system responsible for automatically managing memory by reclaiming memory that is no longer in use. Lua's GC is a key part of its memory management, enabling efficient and automatic memory handling without requiring manual intervention by the programmer. This report delves into the mechanisms, algorithms, and implementation details of Lua's garbage collector, based on the code provided.

Garbage Collection in Lua

Lua's garbage collector is designed to balance performance and memory efficiency. It is implemented with three modes of operation: **full GC**, **incremental GC**, and **generational GC**.

1. **Full GC**: This mode performs a complete sweep of all objects, marking and collecting unused objects in one go.
2. **Incremental GC**: This mode splits the garbage collection process into small steps, interleaving it with the execution of the program. This reduces the overhead and avoids long pauses in the program.
3. **Generational GC**: This mode optimizes for the common case where most objects die young, focusing GC efforts on younger objects more frequently while older objects are collected less often.

Memory Management in Lua

Lua's memory management revolves around managing dynamically allocated objects, which include tables, functions, threads, and user data. Lua uses a memory allocator to manage these objects, with the garbage collector ensuring that memory for objects that are no longer referenced is reclaimed.

Mark-and-Sweep Algorithm (Common to all types of GC)

Lua’s GC is based on the **mark-and-sweep** algorithm. This algorithm is divided into two phases:

1. **Mark Phase**: In this phase, the collector traverses all accessible objects, starting from the root set (global variables, stack, etc.), and marks them as "alive." Any object not marked as alive is considered garbage.
2. **Sweep Phase**: In this phase, the collector scans all objects and frees those that were not marked as alive in the previous phase.

Mark Phase Details:

- **Traversing Objects**: Lua's GC starts marking from the root set, which includes all global variables, the stack, and the registry. It traverses through all references from these roots to other objects and marks them.
- **Marking Strategy**: Lua uses a color scheme to mark objects. The colors are typically represented as **white** (unmarked, collectible), **gray** (marked but not fully traversed), and **black** (marked and fully traversed). The root set objects are initially marked gray, and as they are fully traversed, they turn black. If any new references are found during traversal, they are marked gray and later turned black.

Sweep Phase Details:

- **Sweeping Process**: After the mark phase, Lua's GC sweeps through all objects in memory, checking their color. Objects that remain white (unmarked) are considered unreachable and are thus collected (i.e., their memory is freed).
- **Incremental Sweeping**: In incremental mode, this sweeping process is broken down into smaller steps to interleave with the program's execution, thereby minimizing pauses.

Incremental Garbage Collection

The incremental mode of Lua's GC aims to reduce the long pauses that can occur during garbage collection by breaking down the mark-and-sweep process into smaller increments. Here's how it works:

1. Step-wise Collection

- **Marking Phase**: In traditional garbage collection, all objects are marked in one go, which can cause noticeable pauses in the program. In incremental GC, this process is split into smaller steps. Each step marks a fixed number of objects, allowing the program to continue executing between these steps.
- **Sweeping Phase**: Similarly, the sweeping phase, where unmarked objects are collected (i.e., their memory is reclaimed), is also broken down into smaller steps.

Key Functions:

- `luaC_step`: This is the entry point for Lua's incremental GC. It determines what phase the GC is in and advances the collection process. Depending on the current state, it might:
    - Start a new marking phase.
    - Mark some objects.
    - Perform a sweeping step.
    - Pause the GC if enough work has been done.

- `gc_onestep`: This function is responsible for executing a single step in the incremental GC process. It decides whether to continue marking, to switch to sweeping, or to pause the GC until the next step.

2. Barriers

- Barriers are crucial in incremental GC to maintain consistency in the garbage collector's view of the object graph. They ensure that the GC correctly tracks object references as they change during the program's execution.
- **Forward Barriers**: When a reference to a new object is written, a forward barrier ensures that the new object is marked if necessary. This prevents newly created objects from being prematurely collected.
- **Backward Barriers**: When a reference is removed, a backward barrier ensures that the old object is revisited by the GC if needed. This is important to ensure that any objects that might have been marked "live" due to the removed reference are correctly updated.

Key Functions:

- `luaC_barrier_`: This is the main barrier function used for forward barriers. It ensures that when a reference is written to an object, the GC state is updated to mark the newly referenced object if it is not already marked. It is called whenever an object reference is updated.
- `luaC_barrierback_`: Used for backward barriers, this function ensures that when a reference is removed, the GC revisits the old object to potentially re-mark it, preventing it from being mistakenly collected.
- `luaC_barrierproto_`: This function ensures the consistency of prototype objects in Lua. Prototypes are a special kind of object in Lua, representing functions, and this function makes sure that the GC tracks changes to these objects correctly
- `reallymarkobject`: This function is used by the barrier functions to mark an object as "reachable" during the marking phase. It is a key part of ensuring that objects are correctly marked during the incremental marking process.
- `setpause`: The function in Lua's garbage collector is responsible for adjusting the threshold for garbage collection based on the current state of the program's memory usage

Generational Garbage Collection

Generational GC in Lua takes advantage of the observation that most objects die young. This mode divides objects into two generations: young and old.

- **Young Generation**:
    - Newly allocated objects are placed in the young generation. This generation is collected frequently because most objects tend to be short-lived, meaning they are likely to become unreachable soon after being created.

    - **Collection**: The GC frequently scans this generation for unreachable objects, reclaiming memory as soon as possible.
- **Old Generation**:
    - Objects that survive multiple collections in the young generation are promoted to the old generation. Objects in the old generation are less likely to be garbage, so the GC scans this area less frequently.
    - **Collection**: The old generation is collected less often, reducing the overhead of repeatedly checking objects that are likely still in use.
- **Promotion**:
    - When an object survives a certain number of GC cycles in the young generation, it is promoted to the old generation. This is done to avoid repeatedly checking the same objects during young generation collections.
    - **Mechanism**: Objects that have proven to be long-lived are moved out of the young generation to optimize the efficiency of the garbage collector.
- **Ephemeral Collections**:
    - Quick, lightweight collections can be performed on the youngest objects (i.e., those recently created). This helps optimize performance by focusing the GC efforts on those objects most likely to be garbage.

Lua Source Code Functions Related to Generational GC

1. **Age Stages**:
    - **G_NEW**: Represents newly allocated objects.
    - **G_SURVIVAL**: Objects that have survived one collection cycle.
    - **G_OLD0**: Objects that have survived multiple cycles but are not yet considered fully old.
    - **G_OLD1**: Objects on the verge of becoming fully old; may still be blackened (marked as used) during a major collection.
    - **G_OLD**: Fully promoted objects that are considered long-lived.
    - **G_TOUCHED1 and G_TOUCHED2**: Intermediate states used during the collection process to track certain objects' states.
2. **Key Functions**:
    - `youngcollection`: Handles the collection of the young generation. It focuses on reclaiming memory from objects that are likely to be unreachable soon after allocation.
    - `fullgen`: Performs a full (major) collection that includes both young and old generations.
    - `sweepgen`: Sweeps through objects, deleting dead ones and promoting surviving objects to the next age stage.
    - `setminordebt`: Adjusts the debt (a measure of how much memory has been allocated since the last collection) to trigger minor collections.
    - `atomic`: Marks all objects during a major collection, ensuring that all reachable objects are properly marked before sweeping.

    - `markold`: Marks OLD1 objects when starting a new young collection, ensuring they are correctly tracked.
    - `sweep2old`: Sweeps through a list of objects, promoting non-dead objects to the old generation.

Full Garbage Collection

Purpose:

- Full GC aims to perform a comprehensive cleanup of all objects in memory, regardless of their age or generation. It is typically used when a thorough collection is required, such as when memory pressure is high or after a significant change in the program's memory usage.

Characteristics:

- **Complete Sweep:** Full GC involves marking and sweeping all objects, not just those in the young or old generation. This ensures that no object is left unchecked, making it effective at reclaiming memory but potentially more disruptive due to longer pauses.
- **Major Collection:** It often involves a major collection step that looks at the entire heap, reclaiming all unreferenced objects and resetting the memory state.
- **Transitioning Between Modes:** During a full GC, the collector may switch between incremental and generational modes based on performance and memory usage. For example, if the collector detects that recent collections were inefficient (bad collections), it may perform a full GC to reset and optimize.

Key Functions:

- `luaC_fullgc`: Forces a complete garbage collection cycle, cleaning up all objects and reclaiming memory.
- `sweepgen`: Handles the sweeping phase in generational mode but is used here to manage all objects, promoting or removing them as necessary.
- Along with these it uses some functions from generational and incremental GC's

Performance Considerations

The performance of Lua's GC is influenced by various factors:

- **Incremental vs. Full GC**: Incremental GC offers better performance during normal execution by spreading out the GC work but may not be as efficient in reclaiming memory as full GC.
- **Generational GC**: This mode can significantly improve performance by reducing the frequency of collections for long-lived objects, focusing on collecting short-lived objects.