CS60203: Design Optimization of Computing Systems *August 28, 2024*

Assignment 1 : Part 1

Understanding Lua's GC

Instructor: Mainack Mondal *Written-By: Sanskar Mittal*

1 Introduction

Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

2 Garbage Collection

Lua's garbage collector (**GC**) is an automatic memory management system designed to handle the deallocation of unused memory, thereby preventing memory leaks and optimizing performance. It primarily works by tracking object references and identifying objects that are no longer accessible from the program. Once such objects are found, they are automatically reclaimed, freeing up memory for other uses. Lua uses two types of garbage collection modes: **incremental** and **generational**.

2.1 Incremental GC

The garbage collection process is broken down into small, incremental steps that run concurrently with the program's execution, minimizing noticeable pauses. It operates in two phases: **mark**, and **sweep**.

1. **Mark Phase**
    - The GC begins by identifying and marking all **live** objects directly reachable from the **root set** (e.g., global variables, stack).
    - Initially, objects are marked as "**white**" (unmarked and potentially garbage) and "**gray**" (reachable but requiring further inspection).
    - During traversal, the GC moves objects from "gray" to "black," marking all objects referenced by the current gray objects, ensuring all reachable objects are identified.
    - The "**barrier**" mechanism tracks and manages changes to objects, ensuring objects created or modified during the GC cycle are properly processed.

2. **Sweep Phase**
    - After all reachable objects are marked, the GC **sweeps** through memory, reclaiming space occupied by objects that remain "white" (unmarked and no longer in use).
    - This phase is executed incrementally, with memory freed in portions, distributing the workload over time.

2.2 Generational GC

Lua's generational garbage collection is an optimization of its memory management system that improves efficiency by categorizing objects based on their **lifespan**. This approach is rooted in the observation that most objects are either very **short-lived** or **long-lived**, with relatively few objects falling in between. By exploiting this pattern, Lua's generational GC minimizes the overhead associated with garbage collection, particularly for long-lived objects.

Objects are divided into **generations**, typically categorized as young and old, based on how long they have been in memory. It is observed that most objects **die young**, meaning they become unreachable soon after allocation. Lua's GC separates objects into two generations: the **young generation**, where newly allocated objects reside, and the **old generation**, where objects that survive multiple GC cycles are promoted.

1. **Young Generation Collection**
    - The GC frequently collects the young generation because most objects in this generation are **short-lived**.
    - During this phase, the GC identifies and reclaims memory from objects that have become unreachable shortly after their creation.
    - Objects that survive several collection cycles in the young generation are **promoted** to the old generation, as they are likely to have a longer lifespan.
2. **Old Generation Collection**
    - The old generation is collected **less frequently**, as it contains objects that have already survived several young generation collections and are presumed to be long-lived.
    - Collecting the old generation involves more extensive processing since these objects are less likely to be garbage. However, because the collection is infrequent, the overall impact on performance is minimized.

Until Version **5.0**, Lua used basic **mark & sweep** collector or **fullGC**. It is a complete garbage collection cycle where all objects in memory are examined and processed, regardless of their generation or state.

Unlike incremental or generational GC, which target specific subsets of objects, a full GC cycle is **exhaustive**. It starts by marking all reachable objects from the root set and propagates this marking through all references, ensuring that no live objects are mistakenly collected. Following the marking phase, the GC then sweeps through all objects, reclaiming memory occupied by those that are unmarked (i.e., unreachable).

A full GC is more **resource-intensive** and can cause noticeable pauses in program execution, but it is necessary to thoroughly clean up memory, particularly in situations where incremental or generational GC might miss long-lived objects that have become unreachable.

3 Code Analysis

Lua's GC is implemented in C, and the main components are located in the 'lgc.c' file of the Lua source code.

3.1 States

Lua's GC operates through various states defined in the source code. Key states include:

- `GCSpause`: The GC is idle, waiting for the next cycle to begin.
- `GCSpropagate`: The GC is marking objects.
- `LUAI_GCPAUSE`: Wait memory to double before starting new cycle

3.2 Functions

- `luaC_step()`: This function progresses the GC state machine, performing small steps of work based on the current GC state. In the case of incremental GC, this function ensures that the collection process is distributed over time, preventing long pauses.
- `reallymarkobject()`: This is responsible for marking live objects during the mark phase.
- `luaC_barrier_(Lua_State *L, GCObject *o, GCObject *v)`: This function ensures that when a "black" object (`o`) references a "white" object (`v`), the garbage collector maintains its invariants by marking `v` and handling it correctly according to the current GC mode (incremental or generational).

  1. In generational GC mode, if the black object `o` is old, the white object `v` must be promoted through stages (`G_OLD0`, `G_OLD1` and `OLD`) to ensure that all objects it references will also become old.
  2. In incremental GC mode, the function may clear the black object's mark (turning it white) to avoid repeated barrier checks, thereby optimizing the sweeping process.

- `lu_mem_propagatemark()`: This function traverses one gray object, turning it to black.
- `sweeplist()`: sweep across a list of `GCObjects` erasing dead objects, where a dead object is one marked with the old (non current) white. It also changes all non-dead objects back to white, preparing for next collection cycle.
- `luaC_freeallobjects()`: This function calls all finalizers of the objects in the given Lua state, and then free all objects, except for the main thread.
- `luaC_fullgc(lua_State *L, int isemergency)`: This function performs a full GC cycle. If `isemergency`, set a flag to avoid some operations which could change the interpreter state in some unexpected ways (running finalizers and shrinking some structures).