Conservative Garbage Collection for C

October 25, 2020

1 Introduction

In this assignment, you will implement a conservative garbage collector (`SafeGC`) for "C" language. Unlike, managed languages (e.g., Java), "C" is not designed for memory safety. "C" allows arbitrary typecasts and pointer arithmetic. For example, casting a pointer to an integer and vice-versa is permitted in "C". Due to the above reason, it is difficult to emit the precise type information at every program point statically. Without the accurate type information, the existing garbage collection schemes (which were primarily designed for managed languages) don't work with "C". We will discuss conservative garbage collection approach that works for "C" in the next section.

2 Conservative garbage collection

A conservative garbage collector requires applications to have the following properties.

- Heap memory is always allocated through a memory management API (e.g., malloc).
- If a heap object (say obj) is live, then the application must store an address in the range "`obj` to `obj + sizeof(obj)`" in its address space.

Many applications satisfy the above two conditions and work well with a conservative garbage collector. One of the challenges in implementing a garbage collector is to identify reachable pointers in the application address space (stack + heap). In the absence of the compiler assistance in identifying the pointers, the conservative garbage collector treats all values in the application address space that look like heap addresses as pointers. As a consequence, a conservative garbage collector incorrectly identifies unreachable objects (whose addresses matches with an integer) as reachable, which could lead to a memory leak. Fortunately, in most applications, such cases are few, and thus it is practical to implement conservative garbage collection for them.

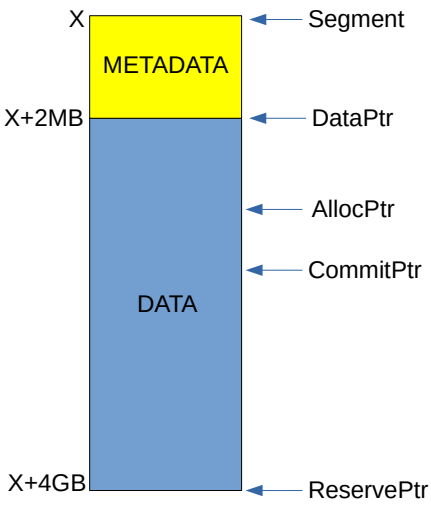

Figure 1: The layout of a segment.

3 SafeGC

`SafeGC` implements a bump allocator that always returns an 8-byte aligned address. `SafeGC` has three components: `allocator`, `mark`, and `sweep`. The assignment repository already contains the `allocator`. You have to implement `mark` and `sweep` phases of the `SafeGC`. We will now discuss the components of `SafeGC`.

3.1 Allocator

`SafeGC` implements a bump allocator and a stop-the-world conservative garbage collector. The allocator maintains a list of segments. All objects are allocated from a segment. A segment is a 4-GB contiguous memory area, as shown in Figure 1. Physical pages to the segment memory are allocated on demand. `ReservePtr` points to the end of the segment. `CommitPtr` points to a memory area until which the physical pages have been allocated. `AllocPtr` is the head of the bump allocator. The first 2-MB of the segment is reserved for metadata (discuss later in this section). The rest of the segment is used by the bump allocator. `DataPtr` points to the first byte in the data region of a segment.

`SafeGC` adds an object header to each object. An object header contains the size, status, and type of the object. `SafeGC` doesn't use the type field of the object header (it will be used in next assignment). The status field contains the validity of an object. When an object is freed, the status filed of the object is set to `FREE`. The status field can also be used by the `mark` phase to mark reachable objects. The bump allocator never reuses a virtual address. When a

segment is full, a new segment is created for future allocations.

`myfree` is called to free an object. `myfree` reclaims the physical page associated with a virtual page when all the objects on the virtual pages are freed. A virtual page is a 4-KB contiguous memory area in the segment address space. A virtual page is also aligned to 4-KB size. Future accesses to a freed virtual pages result in segmentation faults. `SafeGC` reserves two bytes metadata corresponding to each virtual page in the segment. The metadata is stored in a contiguous 2-MB memory area at the top of the segment. The virtual page metadata tracks the number of free bytes on a virtual page. When the number of free bytes becomes equal to the size of a virtual page, the corresponding physical page is reclaimed. When the allocation size of an object is less than 4-KB (page size), `SafeGC` ensures that the object always lies on a single page. The top of a virtual page always contains the starting address (address of object header) of an object.

Allocations of size more than 4-KB allocations are called big allocations. `SafeGC` implements a different allocation scheme for big allocations. For big allocations, the allocation size is adjusted to the nearest multiple of the page size. For these allocations, `myfree` immediately reclaims the physical pages. The metadata corresponding to the first page of a big allocation is set to one to identify the first byte of these objects. `myfree` sets the metadata corresponding to all pages of a big allocation to the page size. The metadata can be used by the `mark` and `sweep` phase to check the validity of a page before access, and also for the finding the object headers of big allocations.

4 Mark

`Mark` phase is triggered when 32-MB data is allocated since the last mark phase. In the mark phase, `SafeGC` checks each address on the stack and in the data section (global variables) for a heap address. These heap addresses are marked and added to a scan list. A scanner scans all the objects in the scan list for unmarked heap addresses. On encountering an unmarked heap address, the scanner marks it before adding to the scanner list. Already marked objects are ignored by the scanner. Notice that a heap address may also point to an internal field of an object. `SafeGC` always adds the start of an object (object header) to the scan list.

5 Sweep

`Sweep` phase walks all the pages that are not yet reclaimed by the `myfree`. All the objects that are not marked and not free are freed in this step. The status of marked objects is reset in this phase.

6 Environment

For this assignment, you need to clone the project repository `https://github.com/Systems-IIITD/SafeGC`.
`SafeGC` contains a test case (`RandomGraph.c`) and partial implementation of a conservative garbage collector (`memory.c`). `SafeGC` provides three interfaces to the programmers:

1. `mymalloc` for memory allocation.
2. `runGC` for explicit GC invocation.
3. `printMemoryStats` prints the statistics of the memory manager.

To run the test case, run "`make run`" after executing `make`. You are not supposed to change the test case.

7 Implementation

You have to implement `scanRoots`, `scanner`, and `sweep` routines in `memory.c`. Please feel free to add new routines or change parameters to these routines. You are not supposed to change the `_mymalloc` and `myfree` implementation. You can add new members to `Segment` and other `structs`, but it should be compatible with existing allocator implementation. The existing implementation invokes `_runGC` when 32-MB of data is allocated since the last garbage collection. `_runGC` already identifies stack and global variables locations that require scanning. You have to implement the rest of the garbage collection algorithm.

7.1 How to submit.

You have to implement everything in `memory.c`. You also have to create a report that contains the following details.

- How did you find the object header corresponding to a heap address (including big allocations)?
- Discuss your implementation of sweep.
- Did you add any new "struct" or change the existing "struct"? If yes, discuss the purpose of these changes.
- The output of "make run".

Add `memory.c` and your report to a zip folder and upload it to the submission site.