Project 3: Mark and Sweep Garbage Collector

Goal

Implement a mark-and-sweep garbage collector for SDyn, and GGGGC more generally.

Time

This project is is due by 12PM (**noon**, not midnight), Friday, March 15th, 2019.

Requirements

- Your implementation must be based on SDyn 1.4 or later. NOTE: SDyn 1.3 and earlier had some bugs between SDyn itself and the GC which could have caused problems in testing different GC implementations. As such, please use SDyn 1.4; you do not need to retain changes from previous assignments.
- You must only modify the GC (i.e., the ggggc directory)
- You must not break any existing functionality of SDyn.
- Your code should compile and run on a standard 64-bit x86_64 Linux environment. It will be tested on `linux.student.cs.uwaterloo.ca`.
- The GC `Makefile` must be changed to compile with `collector-ms.c` rather than `collector-gembc.c`, its default. You must write `collector-ms.c`.
- You must implement `ggggc_malloc`, `ggggc_mallocRaw`, `ggggc_yield` and `ggggc_collect0`, which account for the entire external interface of the garbage collector.
- The GC must include a file named `project3.txt` which briefly details the techniques used in the implementation of the GC, in particular the choice of free-list strategy or other allocation strategies.
- The GC must use a conventional mark-and-sweep approach, with a trace to mark the reachable objects in the heap followed by a sweep of the heap to collect unreachable objects.
- After a call to `ggggc_collect0`, all unreachable space must be available to future calls of `ggggc_malloc`. That is, revocation must be correct.
- Objects returned by `ggggc_malloc` must be correctly allocated, include a pointer to their `struct GGGGC_Descriptor` in their object header (i.e., `obj->header.descriptor__ptr`), and otherwise be zeroed.
- Objects returned by `ggggc_malloc` must be unique among reachable objects. i.e., no two calls to `ggggc_malloc` should return overlapping space, unless an intervening collection revoked the first such object.

Options

- Although revocation must use the mark-and-sweep strategy, you are free to implement allocation in any way you see fit, so long as it is properly described in `project3.txt`.
- When `ggggc_collect0` is not explicitly called, you may use any heuristic you deem fit decide when to collect, so long as it is transparent to the user and does not substantially burden performance.
- Although all space which is freed by `ggggc_collect0` must be available in future calls to `ggggc_malloc`, the GC is not required to coalesce free objects. As a consequence, all space is only guaranteed available when calls to `ggggc_malloc` are made of the minimum object size.
- Space required for collection itself may be allocated from the C heap to ease collection.

- Your collector does not need to be thread safe, as SDyn does not use threads, but you may choose to make it so. With no thread support, `ggggc_yield` (the yieldpoint) may simply be an empty function.
- You may find it useful to implement `GGGGC_DEBUG_MEMORY_CORRUPTION`, which is intended to check for memory corruption by adding an extra field in object headers which is checked at every GC. You are not required to support this debugging flag. If you do implement it, bear in mind that you will need to make sure both GGGGC and SDyn are compiled with this flag; if they're not, they will assume different header sizes, and crash spectacularly.

Hints

It is highly recommended that you write smaller tests to use GGGGC on their own, rather than only using SDyn to test GGGGC. GGGGC will compile into libggggc.a, which can be linked against your own programs, using all the same interfaces as SDyn itself uses, which are documented in `ggggc/ggggc/gc.h`.

GGGGC uses "descriptors" to describe object sizes and pointers, and a standard GGGGC header consists simply of a descriptor pointer. The descriptors are themselves allocated on the heap, and thus have descriptors; the standard descriptor descriptor's descriptor is itself, so the chain of descriptors is short. In a non-moving garbage collector (e.g. the mark-and-sweep collector you are implementing), this should be largely irrelevant: The descriptor will always be available, so the size information will simply be in `obj->header.descriptor__ptr`. However, mind the note on `ggggc_mallocRaw` below.

`collector-gembc.c` may be a useful resource, as it is the only garbage collection core that GGGGC comes with. However, its technique is very different from yours, so be wary of basing your collector too much on it. In particular, note how it traces the pointer stacks (i.e., roots) and object pointers, as these are directly relevant on any garbage collector. Its `ggggc_collectFull` is relevant for that purpose, but read around the `FORWARD` macros, as a non-moving collector never forwards objects.

`ggggc_malloc` takes a descriptor as its input. However, as descriptors themselves must also be allocated, a lower-level function is provided, `ggggc_mallocRaw`, which allocates based only on size, without setting a valid descriptor. It is used only for the allocation of descriptors themselves. After you finish allocating with `ggggc_mallocRaw`, the heap will be in an unclean state: An object has been allocated with no descriptor. You are not expected to somehow be able to collect in this state, and every time `ggggc_mallocRaw` is used, the descriptor is immediately set after the call, by surrounding code in `allocate.c`. However, this does mean that you may *not* collect safely at the end of `ggggc_mallocRaw`. Be wary of when collection is safe.

Most of the SDyn tests are short-running and so will only invoke allocation with any usual GC heuristic. Make sure to test on `binsearch3.sdyn`, which runs for a long time and will certainly invoke garbage collection.

Make sure that you `make clean` in the `ggggc` directory before swapping `collector-gembc.c` for `collector-ms.c`. `ar` is not wise to the *removal* of files from an archive.

Marks

Your submission will be graded partially on "black-box" tests, which test its correctness with no examination of the code, and partially on "white-box" tests, which involve direct inspection:

- 33.34%: All tests included in the GGGGC template, plus your test, plus private grading tests, compile correctly, give correct output and run without consuming excessive amounts of memory. If the submitted GC does not compile, none of these points are available. Although no performance requirements are set in this project, they are graded by a human (me), so execution times which test human patience will be given no points.
  - 11.12%: The GC compiles, and SDyn compiles with the submitted GC.
  - 11.11%: All included and grading tests give correct output and do not crash. Note that grading tests include tests which use GGGGC but not SDyn.

  - 11.11%: Memory consumption on tests is not excessive[1].

- 33.33%: Allocation is implemented correctly. Because allocation and revocation are related, some aspects of revocation are included in these points. Because you are free to implement allocation in any way you please, there is no single break-down of these points for all projects. Here is an example break-down for an implementation of a simple first-fit free-list:
  - 7%: Size is correctly considered when selecting an object from the free-list. i.e., objects are never allocated in spaces too small to fit them.
  - 7%: Splitting is implemented correctly.
  - 7%: Overallocation is either unnecessary or implemented correctly: Overallocated space is not lost due to mismatches between descriptor size and allocated size.
  - 4%: Objects are always correctly removed from the free-list, and never reused.
  - 3%: Allocation from free space is performed correctly.
  - 3%: Allocation proceeds through multiple pools correctly when necessary.
  - 2.33%: Allocation of new pools is implemented correctly.
- 33.33%: Collection is implemented correctly.
  - 10%: Mark phase correctly identifies all roots.
  - 13.33%: Mark phase correctly identifies are reachable objects. In particular, this means it correctly uses descriptors and does not follow non-references.
  - 10%: Sweep phase correctly identifies all unreachable objects and formats them correctly for the allocator.

Notes

- SDyn and GGGGC are released under the ISC license. Your changes are your own.
- Typically, object headers start *before* the object, i.e. at `&((struct ObjectHeader *) ptr)[-1])`. In GGGGC, object headers start *with* the object, i.e. at `((struct GGGGC_Header *) ptr)`. The header is contained in the size of the object. Expanding the header will not break this invariant, but should be unnecessary in this project.
- Descriptors in GGGGC contain the relevant type information in terms of a pointer bitmap. The pointer bitmap is in a partially-reversed order: The *low* bit (last bit) of the *first* word of the pointer map corresponds to the *first* word of the object. Because GGGGC objects include the header, this is the first word of the header. The *high* bit (first bit) of the first word of the pointer map corresponds to the 64th word of the object on a 64-bit system. The *low* bit (last bit) of the second word of the pointer map corresponds to the 65th word of the object on a 64-bit system.
- The first word of the header should always be the object descriptor reference. As a result, the first word of every object is always a reference. Taking advantage of this fact, the low bit of the pointer bitmap is used for another purpose: If the object has no references aside from the descriptor, the low bit will be set to 0. A collector can thereby avoid scanning the whole object when it contains no references.
- Typical user code never calls `ggggc_collect0` directly. However, test code for this project will.
- SDyn uses a patched version of GGGGC, and so ships with an unpatched version in `ggggc-unpatched`. Don't touch the unpatched version, and definitely avoid `make distclean`, or the entire `ggggc` directory, including your changes, will be deleted!

[1] "Excessive" is not precisely defined. Generally speaking, if your collector is incorrect, it will leak memory, and thus any minor memory problem will be amplified over time, sometimes exponentially. Such bugs often crash the program or even cause system instability. Such problems are considered excessive. For these tests, minor memory bugs which do not present such visible errors will not be considered.

- There are very few software components more difficult to debug than garbage collectors. Valgrind will not help you. `gdb` will catch bugs long after the root cause. Print statements are crucial but not sufficient. Your best bet for debugging is what are called "canaries", implemented by `GGGGC_DEBUG_MEMORY_CORRUPTION`.
- Do not be afraid of rewriting. Rewriting a portion of this project, or even the entire project, may take considerably fewer hours than tracking down an obscure bug.

Submission

Project code must be submitted by email to the instructor. I have many email addresses, all of which go to the same inbox, but they get tagged differently; if you send your submissions to `cs842-2019@gregor.im`, I will appreciate it. As email isn't always the most reliable mechanism, I will respond to verify that I have received each submission. If you don't see a response from me, it's possible that something has gone wrong, so please tell me.

Late submissions are not accepted without prior agreement from the instructor. Extensions will rarely be granted. Requests for extensions sent within the 24 hours prior to the due date, or sent after the due date, will be ignored, except where existing university regulations on extensions, e.g. due to illness, are relevant.

Exceptions and help

For reasons of fairness, uniform projects are required. However, if you need to deviate from this document for exceptional reasons, or if you want any clarification or help, feel free to email the instructor.