Project 3: Generational Garbage Collector

Goal

Implement a generational garbage collector, using mark-and-sweep in the old generation.

Time

This project is assigned as of 9AM Monday, October 26th and is due by 12PM (**noon**, not midnight), Monday, November 9th.

Requirements

- The GC must be compatible with the public GGGGC API, accessible through `ggggc/gc.h`, plus the functions `void ggggc_collect(void)` and `ggggc_collectFull(void)`. Simply using the GGGGC template provided is sufficient so long as you don't reduce the API, and name your full collection function `ggggc_collectFull`.
- The GC must include a `Makefile`, and `make` with no arguments should build `libggggc.a`. `libggggc.a` should have no further library requirements.
- The GC must compile on `linux.student.cs.uwaterloo.ca` in 64-bit mode.
- The GC must include a file named `project3.txt` which briefly details the techniques used in the implementation of the GC.
- A single-file test program must be included named `project1-test.c` if implemented in C, or `project1-test.cc` if implemented in C++, expected to be the same test program as in project 1 (hence the name). It may be different if you prefer, but must follow the same requirements as project 1's test program.
- The GC must be implemented in C or C++. If implemented in C++, the Makefile must be updated to call `g++` or `c++` as appropriate, and its API must be in C. The template provided will expose a C API even if compiled as C++.
- The GC must partition objects into "young" and "old".
- Newly allocated objects must be allocated in the "young" partition. No public API for allocation in the old generation is required.
- A collection of the young generation must treat certain objects in the old generation as roots, but must *not* treat the entire old generation as a root. As such, your collector will have to implement a write barrier to build a remembered set, typically in the `GGGGC_WP` macro in `ggggc/gc.h`.
- Objects must be promoted to the old generation based on some time metric. i.e., promotion must occur, but must not be en masse.
- The young and old generations are to be distinguished by location (i.e., which pool contains an object), so copying is, at a minimum, required from the young generation to the old.
- The old generation 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.
- Unlike in project 1, the old generation must coalesce free space. In the case of a typical free-list, this means that if two or more free objects abut each other, they must eventually be combined into one. The obvious time to do this is during collection. If your implementation coalesces at a different time, please document that in `project3.txt`. In the case of bitmapped free-list, coalescence is automatic.
- After an explicit call to `ggggc_collect`, all references must be updated correctly, all living objects copied at most once, and no dead objects copied at all.

- After an explicit call to `ggggc_collectFull`, all unreachable objects must be revoked, available for future calls to `ggggc_malloc` or future promotions, as appropriate. 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 amongst reachable objects. i.e., no two calls to `ggggc_malloc` without an intervening collection should return overlapping space.

Options

- The young generation may use any strategy, so long as it implements a timing metric of some kind (such as a collection counter or bucket brigade) and objects are copied to the old generation for promotion.
- In the old generation, 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 `project1.txt` and it coalesces.
- The included test program, `project1-test.c`, may perform any task you please, so long as it allocates enough memory to require GC.
- When `ggggc_collect` and `ggggc_fullCollect` are 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.
- Space required for collection itself may be allocated from the C heap to ease collection.

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:

- 25%: 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.
  - 8.34%: All tests compile against the submitted GC.
  - 8.33%: All included and grading tests give correct output.
  - 8.33%: Memory consumption on tests is not excessive[1].
- 25%: Allocation is implemented correctly in both generations. 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:
  - 5%: 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.
  - 5%: Splitting is implemented correctly.
  - 5%: Overallocation is either unnecessary or implemented correctly: Overallocated space is not lost due to mismatches between descriptor size and allocated size.

[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.

  - 4%: Objects are always correctly removed from the free-list, and never reused.
  - 2%: Allocation from free space is performed correctly.
  - 2%: Allocation proceeds through multiple pools correctly when necessary.
  - 2%: Allocation of new pools is implemented correctly.

- 25%: Collection is implemented correctly.
  - 5%: All roots are correctly identified.
  - 10%: All reachable objects are handled correctly with any form of collection.
  - 10%: All references are correctly updated.

- 25%: Partitioning is implemented correctly.
  - 8: When cross-generational pointers are created, whether explicitly or through promotion, the relevant object is added to a remembered set.
  - 5: Young collection correctly scans the remembered objects (some imprecision is expected here) as roots.
  - 4: Young collection never scans old objects that were not in the remembered set.
  - 8: Objects are promoted correctly, by copying, and with some correct time metric.

Notes

- GGGGC is released under the ISC license. You are free to look at the original source, but since it implements a very different GC, it's unlikely to help much. 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. If you expand the header, you will have to change `GGGGC_DESCRIPTOR_DESCRIPTION`, which is the pointer bitmap for `struct GGGGC_Descriptor`.

- 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.

- Since descriptors are part of the heap, GGGGC creates "descriptor descriptors". This is a complicated problem and you shouldn't have to study the code that does it in detail, but there is one important point: It sometimes allocates descriptor-descriptors by calling `ggggc_malloc` with a temporary descriptor that is not actually on the heap. If your GC runs at the wrong time, it could find these objects and be confused. Your GC may easily distinguish such temporary descriptors by the fact that their `descriptor__ptr` field is set to `NULL`. Simply ignore any object with a `NULL` descriptor.

- 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_collect` or `ggggc_collectFull` directly. However, test code for this project will.

- This project is expected to reuse code from both projects 1 and 2.

- 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": Fields in objects, in object headers or between objects in the heap which have fixed values, typically `0xDEADBEEF`. By inserting canaries and then checking them frequently, it is possible to determine what code corrupted the heap.
- 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 through the `submit` command available on `linux.student.cs.uwaterloo.ca` or its web interface. The course is `cs842` and the project is `gcproject3`. e.g.: `submit cs842 gcproject3 ./mygcprojectcode`. See `man submit` for further instructions on the `submit` command.

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

Due to the size of this course, 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. My email is available at my web site, `http://the.gregor.institute/`.