Assignment 7: Heap allocator

Due: Thu Dec 7 11:59 pm
Ontime bonus 5%. Grace period for late submissions until Sat Dec 9 11:59 pm

Assignment by Julie Zelenski
based on idea from Randal Bryant & David O'Hallaron (CMU)

Learning goals

In completing this assignment, you will:

- appreciate the complexity and tradeoffs in implementing a heap allocator
- further level up your pointer-wrangling and debugging skills
- profile your code to find inefficiencies and work to streamline them
- consider improvements against the complexity of code required to achieve them
- bring together all of your CS107 skills and knowledge into a satisfying capstone experience-- you can do it!

Overview

Watch video walkthrough! (https://www.youtube.com/watch?v=3Ix8A1rOy9w)

You've been using the heap all quarter, now is your chance to peel back the covers and work out the magic underpinnings of `malloc`, `realloc`, and `free` for yourself! The key goals for a heap allocator are:

- correctness: services any combination of well-formed requests
- tight utilization: compact use of memory, low overhead, recycles
- high throughput: can service requests quickly

There are a wide variety of designs that can meet these goals, with various tradeoffs to consider. Correctness is non-negotiable, but is the next priority on conserving space or time? A bump allocator can be crazy-fast but chews through memory with no remorse. Alternatively, an allocator might pursue aggressive recycling and tight packing to squeeze into a small memory footprint but execute a lot of instructions to achieve it. A industrial-strength production allocator aims to find a sweet spot without sacrificing one goal for the other. This assignment is an exploration of implementing a heap allocator and balancing those tradeoffs.

Getting started

Check out a copy of your cs107 repository with the command:

```
git clone /afs/ir/class/cs107/repos/assign7/$USER assign7
```

1) Code study: test harness

In order to put a heap allocator through its paces, you will need a good testing regimen. Our provided `test_harness.c` will be a big help! It allows you to test an allocator on inputs supplied as "allocator scripts". An allocator script file contains a sequence of requests in a simple text-based format. The three request types are `a` (allocate) `r` (reallocate) and `f` (free). Each request has an `id-number` that can be referred to in a subsequent realloc or free.

```
a id-number size
r id-number size
f id-number
```

A script file containing:

```
a 0 24
a 1 100
f 0
r 1 300
f 1
```

is converted into these calls to your allocator:

```
void *ptr0 = mymalloc(24);
void *ptr1 = mymalloc(100);
myfree(ptr0);
ptr1 = myrealloc(ptr1, 300);
myfree(ptr1);
```

The test_harness program will parse a script file and translate it into requests. During execution, it checks that each request appears to be correctly satisfied and watches to confirm that the client's payload data is properly maintained.

Read over `test_harness.c` so that you understand how it can help you test your allocator. It's fine to skim over the script parsing, but please pay careful attention to the portions used to evaluate the allocator correctness. Answer these questions in your readme.txt file:

a. Identify the required alignment for memory blocks and explain how it is checked by the test harness.
b. How does the test harness verify that a memory block does not overlap any other?
c. At what point does the test harness call `validate_heap` to confirm that the heap internals are valid?
d. What value does the test harness write to the payload bytes? For what purpose is it writing to the payload?

One easy-to-overlook detail is that the test_harness calls the allocator's `myinit` function after executing one script and before starting another. The function is expected to wipe the slate clean and start fresh; if `myinit` doesn't do its job, you may encounter bugs caused by ghost heap housekeeping leftover from the previous script. Keep this in mind for when implementing `myinit` in your heap allocator!

2) Code study: bump allocator

Your next task is to study the bump allocator. This code demonstrates one of the simplest possible approaches, with blindingly-high throughput and zero concern for utilization. You will be working toward a much more balanced implementation, but read over this code to be introduced to the basics of how a heap allocator might work. Answer these questions in your readme.txt file:

a. The arguments to `myinit` are the bounds of the heap segment. Run `test_bump` under gdb and observe the boundaries. At what address does the segment start? How far does it extend? (hint: the segment size is not the same for all scripts, read test_harness.c to see how size is determined for a particular script) What happens if the allocator attempts to read/write past the end of the segment?
b. Each memory block is required to be aligned on a 8-byte boundary. How does the bump allocator organize blocks so as to conform to this requirement?
c. Free is implemented as no-op. Why is the bump allocator incapable of recycling memory?
d. Its `realloc` operation makes no attempt to resize in place, it always goes with malloc/memcpy/free. Focus in on the memcpy call that is used to move the payload data. Is it correct to use `memcpy` here or would `memmove` be more appropriate? Explain. How many bytes are copied from the old location to the new? This choice of size is a bit fishy. It is clearly wasteful if newsz is significantly larger than the original payload size but it won't cause an error to copy the excess data. Explain why it is guaranteed to succeed. (i.e. correctly preserves contents of original payload and no error comes from reading/writing the excess bytes).

As a consequence of its wanton use of memory, the bump allocator fails on many scripts. Run the test harness on the sample scripts to identify which scripts it can and can't process:

```
./test_bump samples/*.script
```

Once you know which scripts it can handle, run the bump allocator under callgrind to get some rough performance data. (See the advice page (advice.html) for tips on using callgrind to count instructions within the allocator functions.) Use the annotated source to answer the following question in your readme.txt file:

e. Use the callgrind-annotated source to find "hot spots", i.e. those lines with significantly higher concentration of instruction counts. Which lines are the hot spots of bump.c?

3) Implement implicit free list

Now it's your turn! The file `implicit.c` is ready and waiting for you to implement your own recycling allocator that puts the naive bump allocator's memory-chomping approach to shame.

The specific features that your implicit free list allocator must support:

- Headers track block information (size, status in-use or free)
- Free blocks are recycled and reused for subsequent malloc requests
- Malloc searches heap for free blocks via an implicit list (i.e. traverses block-by-block)
- Realloc resizes block in-place whenever possible, e.g. if resize smaller or neighboring block(s) free and can be absorbed (this is a limited form of coalesce only for realloc, no expectation to support coalesce more broadly for implicit)

This allocator won't be that zippy of a performer (an implicit free list is poky at best) but will do a good job of recycling freed nodes which enables it to successfully handle many workloads that cannot be accommodated by the bump allocator. Without coalescing, it is missing out on some opportunities to more aggressively recycle memory, but that comes up next in the explicit version!

The bulleted list above indicates the minimum specification that you are required to implement. Further details are intentionally unspecified as we leave these design decisions up to you; this means it is your choice how to structure your header, whether you search using first-fit versus next-fit, and so on. It can be fun to play around with these decisions to see the various effects, but any reasonable and justified choices will be acceptable for grading purposes.

Answer the following questions in your readme.txt:

a. What lines of implicit.c are its "hot spots"?
b. Compare the behavior/performance of your implicit free list allocator on the sample scripts to that of the bump allocator and briefly summarize your findings. Your answer should consider differences in robustness, throughput, and utilization.

4) Implement explicit free list

The file `explicit.c` is your chance to turn your attention from utilization to throughput and work at making your allocator fly.

The specific features that your explicit free list allocator must support:

- Block headers and in-place realloc (can copy from your implicit version)
- Explicit free list managed as a doubly-linked list, uses first 16 bytes of payload of freed block for next/prev pointers
- Malloc searches explicit list of free blocks
- Free immediately coalesces with any neighbor block(s) to right that are also free. Coalescing a pair of blocks must operate in O(1) time.

Note: the header should **not** be enlarged to add fields for the pointers. Since only free blocks are on the list, the more economical approach is to store those pointers in the otherwise unused payload. The minimum payload size will need be large enough to support this use.

The above bullets represent our requirements, all further decisions not specifically mandated are yours to determine.

Answer the following question in your readme.txt:

a. Compare the instruction counts of your implicit free list allocator versus your explicit free list and summarize the gains in throughput.

If you are looking for some extra challenge, use callgrind to find the hot spots and work to minimize them. Get help from gcc by editing the CFLAGS in your Makefile and then add your own efforts to optimize what gcc cannot. How much improvement do your efforts make? Use your readme to tell us about your adventures in optimization -- we'd love to hear about it!

5) Final readme question

What would you like to tell us about your quarter in CS107? Of what are you particularly proud: the effort you put into the quarter? the results you obtained? the process you followed? what you learned along the way? Tell us about it! We are not scoring this question, but wanted to give you a chance to share your closing thoughts before you move on to the next big thing.

Standard requirements for allocator

Requirements that apply to both allocators:

- The interface should match the standard libc allocator. Carefully read the `malloc` man page for what constitutes a well-formed request and the required handling to which your allocator must conform. Note there are a few oddballs (`malloc(0)` and the like) to take into account. Ignore esoteric details from the NOTES section of the man page.
- There is no requirement on how to handle improper requests. If a client reallocs a stack address, frees an already freed pointer, overruns past the end of an allocated block, or

other such incorrect usage, your response can be anything, including crashing or corrupting the heap. We will not test on these cases.
- An allocated block must be at least as large as requested, but it is not required to be exactly that size. The maximum size request your design must accommodate is `1 << 30` bytes (we originally set at INT_MAX, but that's not even a multiple of 4 or 8 which makes it irritating if you need to round up). If asked for a block larger than the max size, your allocate should return NULL, just as with any request that cannot be serviced. Every allocated block must be aligned to an address that is a multiple of the `ALIGNMENT` constant (8). It's your choice whether small requests are rounded up to some minimum size.
- Your allocator cannot invoke any memory-management functions. By "memory-management" we specifically mean those operations that allocate or deallocate memory, so no calls to malloc, realloc, free, calloc, sbrk, brk, mmap, or related variants. The use of other library functions is fine, e.g. `memmove` and `memset` can be used as they do not allocate or deallocate memory.
- Your allocator may use a small amount of static global data, limited to at most 500 bytes. This restriction dictates the bulk of the heap housekeeping must be stored within the heap segment itself.
- Your `validate_heap` function will be called from our test harness when checking correctness to verify the internal consistency of the heap data structures after servicing each request. This is here to help you with debugging. The validity check will not be made during any performance testing, so there is no concern about its efficiency (or lack thereof).

Advice/FAQ

Don't miss out on the companion advice page:

Go to advice/FAQ page (advice.html)

Grading

Here is the tentative point breakdown:

Readme questions

- **readme.txt**. (30 points) For the code-study questions, you will be graded on the understanding of the issues demonstrated by your answers and the correctness of your conclusions.

Functionality

- **Sanity/sample scripts** (25 points) Correct results on the default sanity check tests and published sample scripts.
- **Comprehensive cases** (55 points) Fulfillment of the specific implicit/explicit requirements and correct servicing of sequences of mixed requests.
- **Robustness** (20 points) Handling of required unusual/edge conditions (request too large, malloc 0 bytes, etc.)
- **Clean compile** (2 points) We expect your code to compile cleanly without warnings.

Code quality (buckets weighted to contribute ~20 points)

- We expect your allocator code to be clean and readable. We will look for descriptive names, helpful comments, and consistent layout. We expect your code to show thoughtful design and appropriate decomposition. Control flow should be clear and direct. We expect common code to be factored out and unified. Complex tasks or tricky expressions that are repeated should be decomposed into shared helpers.
- The code review will also assess the thoroughness and quality of your `validate_heap` routine.

On-time bonus (+5%)

The on-time bonus for this assignment is 5%. Submissions received by the due date earn the on-time bonus. The bonus is calculated as a percentage of the point score earned by the submission.

Special attention to the Honor Code

The textbook contains good foundation material which you are encouraged to review as a starting point. You may also find it helpful to talk over implementation options and trade-offs with classmates. Your readme must properly cite any external resources or discussions that influenced your design. When it comes to code-level specifics, your allocator must be your independent work. Under no circumstances should you be studying code from outside sources

nor incorporating such code. If you accidentally stumble across allocator code written by someone else, we expect you to immediately about-face from it and get back to designing and writing your own allocator. We have a **zero-tolerance policy** for submissions containing code that has been adopted/borrowed from others. This act constitutes plagiarism and will handled as a violation of the Honor Code.

A student who is retaking the course may not re-use the design, code, or documentation from any previous effort. The prohibition against re-use applies regardless of whether the original submission was partnered or individual. The best approach to avoiding problems is to discard all work from the past, as it can be a temptation that you're better off without.

Finish and submit

When finished, be sure to submit (/class/cs107/submit.html) your files for grading.

Absolutely no submissions will be accepted after the grace period ends, nor will there be any option to make up from a missed/poor submission. Cutting it too close can land you on the wrong side of the deadline with no recourse other than to discard all your hard work into the rubbish bin. Do not let this happen to you -- submit early, submit often, so you'll have an insurance policy against any unexpected last-minute mishap.

Check in with post-task self-check (/class/cs107/selfcheck.html) and it's definitely time to celebrate-- congratulations on an awesome quarter!

> Here I am at the end of the road and at the top of the heap. — Pope John XXIII