Malloc

Part 1 due **2019-02-25 23:59**
Graded files:

- alloc.c

Part 2 due **2019-03-04 23:59**
Graded files:

- alloc.c

Content

Learning Objectives

The learning objectives for Malloc are:

- Memory Allocation and Management
- Performance Optimization
- Developing in a Restricted Environment

Backstory

Well, color me impressed! Your shell was so fancy that you actually received that pay raise! However, you *may* have gone too far with your shell. Your boss is now so impressed at your skills that they sent you to the nth Inter-Company Turbo Malloc Contest - even though you're just a newhire! But hey, a business trip doesn't sound that bad, right?

Upon arriving at the competition venue, you realize that all your peers from CS 241 are in the contest! Apparently, all of them went too far with their shells as well, and ended up in the same scenario as you. Just as you were reminiscing about your time in CS 241, you received an email from your senpai in the company about the contest, and your face turns pale immediately.

Turns out, the Inter-Company Turbo Malloc Contest is the official™ way tech companies compete with each other these days - winning the competition guarantees fame and glory for the company, and the losing companies will suffer shame and embarassment. What's more, any company that cannot beat the baseline will incur severe stock drops! Needless to say, losing the competition will ruin all your efforts in the past weeks to impress your boss, and if you fail to beat the baseline, you will probably lose your job, again (and this time, everyone will know your failures, since this contest is heavily publicized).

The competition lasts for two weeks, and you get total freedom on your implementation. This time, you've decided not to procrastinate, since your entire livelihood hinges on how well you perform in this competition. Can you out-think every other contestant's implementations and secure that epic victory royale?

Overview

You should write your implementations of `calloc` (https://linux.die.net/man/3/calloc), `malloc` (https://linux.die.net/man/3/malloc), `realloc` (https://linux.die.net/man/3/realloc) and `free` (https://linux.die.net/man/3/free) in alloc.c. alloc.c will be the only file we test.

Don't modify `mcontest.c` , `contest.h` , or `contest-alloc.so` . Those files create the environment that replaces the standard glibc malloc with your malloc. These files will be used for testing.

Your `malloc` (https://linux.die.net/man/3/malloc) must allocate heap memory using `sbrk` (https://linux.die.net/man/2/sbrk). You may not use files, pipes, system shared memory, `mmap` (https://linux.die.net/man/2/mmap), a chunk of predefined stack memory, other external memory libraries found on the Internet, or any of the various other external sources of memory that exist on modern operating systems.

A Bad Example

Memory allocation seems like a mystery, but in actuality, we are making a wrapper around the system call `sbrk` (https://linux.die.net/man/2/sbrk). Here's a really simple implementation of `malloc` (https://l

```
void *malloc(size_t size) {
    return sbrk(size);
}
```

As you can see, when we request `size` (https://linux.die.net/man/1/size) bytes of memory, we call sbrk(size) to increase the heap by `size` (https://linux.die.net/man/1/size) bytes. Then, we return a pointer to this memory, and we're done. Simple!

Here is our implementation of `free` (https://linux.die.net/man/3/free)

```
void free(void *ptr) {
}
```

This is a “correct” way to implement `free` (https://linux.die.net/man/3/free). However, the obvious drawback with our implementation is that we can’t reuse memory after we are done with it. Also, we have not checked for errors when we call `sbrk` (https://linux.die.net/man/2/sbrk), and we have not implemented `realloc` (https://linux.die.net/man/3/realloc) or `calloc` (https://linux.die.net/man/3/calloc).

Despite all of this, this is still a “working” implementation of `malloc` (https://linux.die.net/man/3/malloc). So, the job of `malloc` (https://linux.die.net/man/3/malloc) is really to allocate memory, but to keep track of the memory we’ve allocated so that we can reuse it. You will use methods that you’ve learned in class and practiced in the Mini Valgrind lab to do this.

Testing Your Code

In order to run your solution on the testers, run `./mcontest` with the tester you want. You **must** do this, or your code will be run with the glibc implementation!

Example:

```
./mcontest testers_exe/tester-1
Memory failed to allocate!
[mcontest]: STATUS: FAILED=(256)
[mcontest]: MAX: 0
[mcontest]: AVG: 0.000000
[mcontest]: TIME: 0.000000
```

We’ve also distributed a bash script `run_all_mcontest.sh` to run all testers. We design the script so that you can easily test your malloc implementation. Here is how you use the script:

```
./run_all_mcontest.sh
```

It will automatically make clean and make again, and then run each test case in the *testers* folder. If you want to skip some test cases, you can do:

```
./run_all_mcontest.sh -s 1 2 3
```

where 1, 2, and 3 are the tests you want to skip. You can skip as many as you like.

Here is what some of our error codes mean:

```
11: (SIGSEGV) Segmentation fault
15: (SIGTERM) Timed out
65, 66: Dynamic linking error
67: Failed to collect memory info
68: Exceeded memory limit
91: Data allocated outside of heap
92: Data allocated exceeds heap limit
```

Good Practices

Since you can implement your malloc in whatever way you want, you may end up with a huge chunk of messy code that’s hard to debug. Here are some suggestions for organizing and maintaining your code better:

- Build simple functions before you add advanced features. In other words, make sure your program does what you want it to do before moving on to optimize it.
- Separate the functionality of your program into smaller chunks of independent code. For example, if you find that you’re frequently splitting a block of memory into two blocks, you probably want to write a split function instead of copy-pasting the splitting code every time you need to split.
- Keep your code readable. This can be naming your variables appropriately or commenting your code well. This will really help you understand what your code is doing when you look back at them three days later!

As you come up with ideas it’s a good idea to try to understand what edges your algorithm might run into and try to make sure you handle them before even implementing your code. You may find this visualizer useful for this task: http://aneeshdurg.me/visual-malloc/ (http://aneeshdurg.me/visual-malloc/)

Debugging

`./mcontest` runs an optimized version of your code, so you won’t be able to debug with `gdb` (https://linux.die.net/man/1/gdb). To solve this, we have provided another version called `./mreplace` which uses a version of your malloc compiled without optimization, so you can debug with `gdb` (https://linux.die.net/man/1/gdb). Here’s an example, running tester2 with gdb:

```
gdb --args ./mreplace testers_exe/tester-2
```

Since `./mreplace` calls `fork` (https://linux.die.net/man/3/fork), you'll need to change the `follow-fork-mode` in `gdb` (https://linux.die.net/man/1/gdb) to be able to set a breakpoint in `alloc.c` :

```
(gdb) set follow-fork-mode child
(gdb) break alloc.c:322
No source file named alloc.c.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (alloc.c:322) pending.
(gdb) run
```

Note: if you terminate your running program and run it again, i.e. if you do this:

```
(gdb) run
Thread 2.1 "tester-2" hit Breakpoint 1, malloc (size=1) at alloc.c:323
323    return ptr;
(gdb) kill
Kill the program being debugged? (y or n) y
[mcontest]: STATUS: FAILED. SIGNAL=(9)
[mcontest]: MAX: 0
[mcontest]: AVG: 0.000000
[mcontest]: TIME: 0.012000
(gdb) run
Starting program: malloc/testers_exe/tester-2
Memory was allocated, used, and freed!
```

it will no longer use your own implementation, and therefore will not stop at the breakpoints you set, and will use the glibc implementations of malloc/calloc/etc. This is because of the way `gdb` (https://linux.die.net/man/1/gdb) handles dynamically loaded libraries.

Real Programs

Both `mcontest` and `mreplace` can be used to launch “real” programs (not just the testers). For example:

```
# ignore the warning about an invalid terminal, if you get it
./mreplace /usr/bin/less alloc.c
```

or

```
./mcontest /bin/ls
```

There are some programs that might not work correctly under your malloc, for a variety of reasons. You might be able to avoid this problem if you make all your global variables static. If you encounter a program for which this fix doesn't work, post on Piazza!

Grading

Here is the grading breakdown:

- Correctness (75%)
  - Part 1 (25%): tests 1-6 complete successfully.
  - Part 2 (50%): tests 1-12 complete successfully.
- Performance (25%): Points only awarded if all part 2 testers complete successfully - due with part2

There are 12 testcases in total. For part 1, you will be graded using tests 1 through 6. For part 2, you will be graded using tests 1 to 12 (tests 1 through 6 get graded twice). Tester 13 is not graded.

There are also performance points, which you are only eligible for if you pass all the testcases. Your malloc will be compared against the `glibc` version of malloc, and given a base performance score as a percentage (accounting for runtime, maximum memory usage, and average memory usage). The base score is calculated using the formula from the contest section below; higher percentages are better. Performance points are then awarded in buckets:

- Better than or equal to 85% of glibc: Full 25% awarded.
- 75-85% (include 75%, exclude 85%): 20% awarded.
- 60-75%: 15% awarded.
- 40-60%: 10% awarded.
- 40% and worse: 0% awarded.

So, let's work out some scenarios:

- Scenario 1: A student gets tests 1 through 6 working for part1 and misses 2 tests on part2. Then they get all of the correctness points for part1, 10/12 of the correctness points for part2 and none of the performance points. Thus this student will receive a `(6 / 6) * 25 + (10 / 12) * 50 + 0 = 66.67%` .
- Scenario 2: A student gets none of the tests working for part1 and gets everything working for part2 and beats `glibc` . Then they get none of the correctness points for part1, 12/12 of the correctness points for part2, and the performance points. This student will receive a `(0 / 6) * 25 + (12 / 12) * 50 + 25 = 75.00%` .
- Scenario 3: A student gets tests 1 through 6 working for part1, then they get all the tests except test 4 working for part2. Then they get all of the correctness points for part1, 11/12 of the correctness points for part2, but they will not receive any of the performance points. This student will receive a `(6 / 6) * 25 + (11 / 12) * 50 + 0 = 70.83%` .
- Scenario 4: A student gets tests 1 through 6 working for part1, then they get all of the tests working for part2, but they can only get to `65%` of `glibc` . In this case, they get all of the correctness points for part 1, all of the correctness points for part 2, but only 15% performance points. So, they get `(6 / 6) * 25 + (12 / 12) * 50 + 15 = 90.00%`

- We modify the allocation numbers slightly when we actually grade.

Contest

View the malloc contest here (https://sp19-cs241-grd-02.cs.illinois.edu/malloc/)

The malloc contest pits your memory allocator implementation against your fellow students. There are a few things to know:

- The test cases used for grading will be randomized with a different seed every day.
- There may be additional, more advanced tests added which won't count for anything but the contest.
- The memory limit is 2.500GB.
- To submit your program to the contest, you simply commit, push your code and run the distributed autograder. The contest page will be updated to reflect the results. You will have a number of runs per day, so that you can update your score as desired.
- We will assign a score to each of the three categories (max heap, average heap, and total time) based on how well your program performs memory management relative to a standard solution.
- You can pick a nickname in `nickname.txt` . You will show up as this name on the contest webpage.
- On the webpage, each test will either be green, which signifies that you passed the test, or red, which signifies that you failed the test.

Scores and ranking

Your score will be computed by the following formula:

$$100\% \times \frac{1}{3n}\sum_{i=1}^{n}\left(\log_2\left(\frac{time_{reference,i}}{time_{student,i}}+1\right)+\log_2\left(\frac{avg_{reference,i}}{avg_{student,i}}+1\right)+\log_2\left(\frac{max_{reference,i}}{max_{student,i}}+1\right)\right)$$

Where:

- n is the number of tests
- $reference$ in the subscript means reference implementation, and $student$ means student's implementation
- $time_{reference,i}$ is the time reference implementation spends on test i
- $time_{student,i}$ is the time student spends on test i
- $avg_{reference,i}$ is the average memory used by reference implementation on test i
- $avg_{student,i}$ is the average memory used by student implementation on test i
- $max_{reference,i}$ is the max memory used by the reference implementation on test i
- $max_{student,i}$ is the max memory used by the student implementation on test i

Higher scores are better.

Note: We reserve the right to slightly modify constants inside the formula to ensure fair grading and prevent gaming the system. However, the basic idea will not be changing, and whatever we use will be the same for everyone.

Example 1.

If a student implementation x performs like the reference implementation, which means it spends the same time and memory as the reference, then x's score will be:

$$\begin{aligned}score_x &= 100\% \times \frac{1}{3n}\sum_{i=1}^{n}\left(\log_2\left(\frac{time_{reference,i}}{time_{x,i}}+1\right)+\log_2\left(\frac{avg_{reference,i}}{avg_{x,i}}+1\right)+\log_2\left(\frac{max_{reference,i}}{max_{x,i}}+1\right)\right)\\ &= 100\% \times \frac{1}{3n}\sum_{i=1}^{n}\left(\log_2(2)+\log_2(2)+\log_2(2)\right)\\ &= 100\% \times \frac{1}{3n}\sum_{i=1}^{n}3\\ &= 100\%\end{aligned}$$

Example 2.

If a student implementation y performs the same as the reference implementation on memory usage, but is twice as slow (meaning $time_{y,i} = 2 \times time_{reference,i}$), then y's score will be:

$$\begin{aligned}score_y &= 100\% \times \frac{1}{3n}\sum_{i=1}^{n}\left(\log_2\left(\frac{time_{reference,i}}{time_{y,i}}+1\right)+\log_2\left(\frac{avg_{reference,i}}{avg_{y,i}}+1\right)+\log_2\left(\frac{max_{reference,i}}{max_{y,i}}+1\right)\right)\\ &= 100\% \times \frac{1}{3n}\sum_{i=1}^{n}\left(\log_2(\tfrac{1}{2}+1)+\log_2(2)+\log_2(2)\right)\\ &= 100\% \times \frac{1}{3n}\sum_{i=1}^{n}2.585\\ &= 86.2\%\end{aligned}$$

Example 3.

If a student implementation z performs three times better than the reference implementation, which means $time_{z,i} = \frac{time_{reference,i}}{3}$, $avg_{z,i} = \frac{avg_{reference,i}}{3}$, and $max_{z,i} = \frac{max_{reference,i}}{3}$, then z's score will be:

$$\begin{aligned} score_z &= 100\% \times \frac{1}{3n}\sum_{i=1}^{n}\left(\log_2\left(\frac{time_{reference,i}}{time_{z,i}}+1\right)+\log_2\left(\frac{avg_{reference,i}}{avg_{z,i}}+1\right)+\log_2\left(\frac{max_{reference,i}}{max_{z,i}}+1\right)\right) \\ &= 100\% \times \frac{1}{3n}\sum_{i=1}^{n}\left(\log_2(4)+\log_2(4)+\log_2(4)\right) \\ &= 100\% \times \frac{1}{3n}\sum_{i=1}^{n}6 \\ &= 200\% \end{aligned}$$

WARNING: As the deadline approaches, the contest page will refresh more slowly. There are 400 students, 12 test cases, and up to a minute or so. It will only retest a student's code if it has been updated, but many more students will be updating their code causing longer waits. Start early, and don't become reliant on the contest page by testing locally!