Writing a Dynamic Storage Allocator

Project 3

Writing a Dynamic Storage Allocator

Introduction

Your task is to create a dynamic storage allocator for use on a mobile phone platform that Snailspeed Ltd. is sure will dethrone the iPhone. They have generated a number of memory allocation traces from some in-house mobile applications; since these traces are representative of the sort of work that the dynamic storage allocator will be asked to do, you should make sure that your allocator performs well on these traces. Since resources are limited on mobile devices, a good solution should be fast and should have as little memory overhead as possible.

1 Getting started

As before, you will be obtaining the project files using git. Use the following command to clone the git repository that has been set up for your group for the assignment.

```
git clone /afs/csail/proj/courses/6.172/student-repos/project3/groupname/
project3
```

Remember that we expect groups to be practicing pair programming, where partners are working together with one person at the keyboard and the other person serves as watchful eyes. This style of programming will lead to bugs being caught earlier, and both programmers always having familiarity with the code. You will find that it'll be difficult to split this project into two individually-completed chunks, so don't plan to divide work in this manner.

2 Heap memory allocator interface

Your dynamic storage allocator will consist of the following four functions, which are declared in `mm.h` and defined in `mm.c`. The `mm.c` file we have given you implements the simplest functionally correct `malloc` package that we could think of. Using this as a starting place, modify these functions (and possibly define other private static functions), so that they obey the proper semantics.

- `int mm_init(void);`

  Before calling `mm_malloc`, `mm_realloc` or `mm_free`, the application program (i.e., the trace-driven driver program that you will use to evaluate your implementation) calls `mm_init`. You may use this function to perform any necessary initialization, such as allocating the initial heap area. The return value should be -1 if there was a problem in performing the initialization and 0 if everything went smoothly.

- `void* mm_malloc(size_t size);`

  This call must return a pointer to a contiguous block of newly allocated memory which is at least `size` bytes long. This entire block must lie within the heap region and must not overlap any other currently allocated chunk. The pointers returned by `mm_malloc` must always be aligned to 8-byte boundaries; you'll notice that the `libc` implementation of `malloc` does the same. If the requested size is zero or an error occurs and the requested block cannot be allocated, a `NULL` pointer must be returned.

- `void mm_free(void* ptr);`

  This call notifies your storage allocator that a currently allocated block of memory should be deallocated. The argument must be a pointer previously returned by `mm_malloc` or `mm_realloc`, and not previously freed. You are not required to detect or handle either of these error cases. However, you should handle freeing a `NULL` pointer – it is defined to have no effect.

- `void* mm_realloc(void* ptr, size_t size);`

  This call returns a pointer to an allocated region, similarly to how `mm_malloc` behaves. There are two special cases you should be aware of.

  - If `ptr` is `NULL`, the call is equivalent to `mm_malloc(size);`.
  - If `size` is equal to zero, the call is equivalent to `mm_free(ptr);`.

  Otherwise, `ptr` must meet the same constraints as the argument to `mm_free`; it must point to a previously allocated block and it must have been previously returned by either `mm_malloc` or `mm_realloc`. You do *not* need to defend against frees to invalid pointers. The return value of `mm_realloc` must meet all of the same constraints as the return value of `mm_malloc`; namely, it be 8-byte aligned and must point to a block of memory of at least `size` bytes.

  There is one additional constraint on the behavior of `mm_realloc`. Any data in the old block must be copied over to the new block. If the new block is smaller, the old values are truncated; if the new block is larger, the value of each of the bytes at the end of the block is undefined.

  A naïve implementation of `mm_realloc` might consist of nothing more than a call to `mm_malloc`, a memory copy, and a call to `mm_free`. This is, in fact, how the reference implementation works; leaving this solution in place is probably a good way to get started. Once you've made progress on `mm_malloc` and `mm_free`, you will want to consider ways of improving the performance of `mm_realloc`.

All of this behavior matches the semantics of the corresponding `libc` routines. Type `man malloc` at the shell to see additional documentation, if you're curious.

3 Checking the consistency of the heap

Dynamic memory allocators are notoriously tricky beasts to program correctly and efficiently. One reason why they can be difficult to program correctly is because they involve a lot of untyped pointer manipulation. Corruption introduced by mishandling pointers might not show up until several operations later, making it extremely difficult to diagnose the root cause for a crash or incorrect output. For this reason, among others, you will find it very helpful to write a heap checker that scans the heap and checks it for consistency. Naturally, exactly what can or should check for will depend on how you choose to implement your storage allocator. However, here are some examples of questions that the heap checker might ask.

- Is every block in the free list marked as free?
- Are there any contiguous free blocks that somehow escaped coalescing?
- Is every free block actually in the free list?
- Do the pointers in the free list point to valid free blocks?
- Do any allocated blocks overlap?
- Do the pointers in a heap block point to valid heap addresses?

Generally, as you design the data structures you will use to solve this problem, you should make a note of any relevant invariants. If your heap checker does *not* verify an invariant, you should have a good reason (explained in your writeup) for the omission.

Your heap checker will consist of the function `int mm_check(void)` in `mm.c`. It must return a nonzero value if and only if your heap is consistent. You are not limited to the listed suggestions nor are you required to check all of them. You are encouraged to print out error messages when `mm_check` fails. A well-implemented (and well-commented) `mm_check` function will form a large component of the grade we will assign after looking at your code.

You can tell the driver to check the heap after every operation by passing the `-c` option to the driver, and looking at the "checked" column. You should also sprinkle heap check assertions in your code when you feel the heap might go from uncorrupted to corrupted (e.g. before and after major operations on your internal data structures). Remember that assertions are only checked in debug mode, and are not executed in release mode. The heap checker is like your own internal suite of unit tests. We will not run it against anyone else's code, and we will not look at it during cross testing or performance testing. If we notice it failing when evaluating your code, we may take points out of your code quality grade for failing your own internal tests.

Along the same lines, you may find it helpful to write some debugging functions that outputs your key data structures to some easy-to-read format on your screen.

4 Support routines

The code in `memlib.c` simulates the memory system for your dynamic memory allocator. You can invoke the following functions in `memlib.c`:

- `void* mem_sbrk(int incr);`

  Expands the heap by `incr` bytes, where `incr` is a positive non-zero integer and returns a generic pointer to the first byte of the newly allocated heap area. The semantics are identical to the Unix `sbrk` function, except that `mem_sbrk` accepts only a positive non-zero integer argument.

- `void* mem_heap_lo(void);`

  Returns a pointer to the first byte in the heap.

- `void* mem_heap_hi(void);`

  Returns a generic pointer to the last byte in the heap.

- `size_t mem_heapsize(void);`

  Returns the current size of the heap in bytes.

- `size_t mem_pagesize(void);`

  Returns the systems page size in bytes (4 KiB on Linux systems). It is unlikely that you will need this.

5 Writing the external validator

For the cross testing component of this project, you will be writing a blackbox (external) validator that can test any malloc implementation. Your validator will be used to test your peers' implementations. Look in `validator.c` for the skeleton of our validator, which lists all of the invariants we want you to check. They are reproduced here, for reference.

- For the given traces, neither `malloc` or `realloc` should return `NULL`.
- Allocated ranges returned by the allocator must be aligned to 8 bytes.
- Allocated ranges returned by the allocator must be within the heap.
- Allocated ranges returned by the allocator must not overlap.
- When calling realloc on an existing allocation, the original data must be intact (up to the reallocated size).

We've provided a linked list representation for the ranges, but you must provide the add and remove operations for this data structure.

6 The trace-based driver

The driver program `mdriver.c` tests your `mm.c` package for correctness, space utilization, and throughput. The driver program is controlled by a trace file. Each trace file contains a sequence of allocate, reallocate, and free commands that instruct the driver to call your `mm_malloc`, `mm_realloc`, and `mm_free` routines in some sequence. The driver `mdriver` accepts the following command line arguments:

- `-t <tracedir>`: Look for the default trace files in directory `<tracedir>` instead of the default directory (`../traces`).
- `-f <tracefile>`: Use one particular tracefile for testing instead of the default set of tracefiles.
- `-h`: Print a summary of the command line arguments.
- `-l`: Run and measure `libc malloc` in addition to the students `malloc` package.
- `-v`: Verbose output. Print a performance breakdown for each tracefile in a compact table.
- `-V`: More verbose output. Prints additional diagnostic information as each trace file is processed. Useful during debugging for determining which trace file is causing your `malloc` package to fail.

The simple implementation given to you will run out of memory on the two `realloc` traces and throw an error since it does not utilize freed space appropriately. Your implementation will be expected to pass all of the traces.

7 Rules and reminders

- You should not change any of the sources in the distribution except for the `Makefile`, `mm.c`, `validator.c`, and `bad_malloc.c`. You are free to add new files and update the `Makefile` appropriately if you wish. All of the other files will be overwritten with fresh copies during cross testing.
- You should not invoke any memory-management related library calls or system calls. This excludes the use of `malloc, calloc, free, realloc, sbrk, brk, mmap` or any variants of these calls in your code.
- The total size of all defined global and static scalar variables and compound data structures must not exceed 256 bytes.

8 Evaluation

You will receive zero points if you break any of the rules or your code is buggy and crashes the driver. Otherwise, your grade will be calculated as follows:

- **Correctness.** You will receive partial credit for each trace that your allocator successfully handles, as evaluated by our validator. If you pass every test trace, you will receive full credit.
- **Performance.** Two performance metrics will be used to evaluate your solution. This is a little bit different from what we've done in previous projects.
  - Space utilization: The peak ratio between the aggregate amount of currently allocated memory (i.e., allocated via `mm_malloc` or `mm_realloc` and not yet freed via `mm_free`) and the size of the heap used by your allocator. The optimal ratio is, of course, 1. You should find good policies to minimize fragmentation in order to make this ratio as high as possible.

– Throughput: The average number of operations completed per second.

The driver program summarizes the performance of your allocator by computing a performance index, P, which is a weighted sum of the space utilization and throughput:

$$P = wU + (1-w)\,\texttt{min}(1, T/T_{\texttt{libc}})$$

where U is your space utilization, T is your throughput, and $T_{\texttt{libc}}$ is the estimated throughput of `libc malloc` on your system on the default traces. The performance index favors space utilization over throughput, with a default of $w = 0.6$.

Since both memory and CPU cycles are expensive system resources, we adopt this formula to encourage balanced optimization of both memory utilization and throughput. Ideally, the performance index will reach $P = w + (1-w) = 1$ or 100%. Since each metric will contribute at most w and $1-w$ to the performance index, respectively, you should not go to extremes to optimize either the memory utilization or the throughput at the expense of the other. To receive a good score, you must perform well in both categories.

- **Validation.** Your malloc validator should catch all violations of the invariants we listed. We will run it on every other students malloc implementations and compare your results with our own to determine your grade.

- **Code quality.** When we look at the code portion of your submission, it should be easy to follow. This will require that it be decomposed into functions, where appropriate; that it use as few global variables as possible; that functions and relevant chunks of code be well-commented; and that functions and variables have relevant, easy-to-interpret names.

  This includes your heap consistency checker, which will also be evaluated on the basis of how comprehensive it is.

- **Written submission.** You will notice that we have not provided you with a list of questions to guide your exploration of this problem. Now that you have several projects under your belts, we will expect you to be able to produce well-documented code and accompanying design materials without prompting.

  To supplement the documentation present in your code, you should submit some additional materials; they should be as concise as possible while still doing an effective job of explaining how your allocator works. Diagrams may be useful (and you should probably include some)!

  Your written materials should describe the data structures you have chosen and how each of your calls manipulates those data structures to accomplish its goals. While a lot of this material will probably overlap the comments present in your code, your write-up should be sufficiently detailed as a stand-alone document that we can completely understand what you are doing without looking at your code.

  As always, be sure to include a discussion of any possibilities that you examined and discarded. If you were forced to entertain any trade offs, be sure to discuss the possible advantages and disadvantages of each choice, and explain why you made the decision that you did.

9 Hints and tips

- Spend plenty of time writing your internal consistency checker, heap visualizer, and other debugging tools. This will save you time in the long run.
- Use the `mdriver -f` option. During initial development, using tiny trace files will simplify debugging and testing. We have included two such trace files.
- Use the `mdriver -v` and `-V` options. The `-v` option will give you a detailed summary for each trace file. The `-V` will also indicate when each trace file is read, which will help you isolate errors.
- Use a debugger. A debugger will help you isolate and identify out-of-bounds memory references.
- Use assertions. When debugging a failure or a crash, sprinkle assertions in your code before and around the crash and rebuild it in DEBUG mode. When a program crashes due to an assertion failure, it prints out the failed condition and the line number of the failed assertion, which is more helpful than "Segmentation Fault."
- You may want to explore encapsulating your pointer arithmetic in static inline functions. Pointer arithmetic in memory managers is confusing and error-prone because of all of the casting which is necessary. You can reduce this complexity significantly by writing helper functions (or macros if appropriate) for your pointer operations.
- Do your implementation in stages. The first 9 traces contain requests to `malloc` and `free`. The last 2 traces contain requests for `realloc`, `malloc`, and `free`. We recommend that you start by getting your `mm_malloc` and `mm_free` routines working correctly and efficiently on the first 9 traces. Additionally, remember that the reference implementation of `mm_realloc` is built on top of `mm_malloc` and `mm_free`. This can provide a place to start working from when you move on to the last 2 traces.
- Use a profiler like `perf` or `gprof` to identify hot spots. Remember all of the techniques we've discussed so far!
- **Start very early**! It is possible to write an efficient `malloc` package with a few pages of code. However, we can guarantee that it will be some of the most difficult and sophisticated code you have written so far, and probably also the most difficult to debug. If you wait until the last minute, you may find that you do not have enough time to produce a worthwhile product. Good luck!

6.172 Performance Engineering of Software Systems

Fall 2010