Project 3

Writing a Dynamic Storage Allocator

Out: In class on Thursday, 8 Oct 2009
Due: In class on Thursday, 22 Oct 2009

In this project, you will be writing a dynamic storage allocator for C programs, i.e., your own version of the `malloc`*,* `free` *and* `realloc` *routines. You are encouraged to explore the design space creatively and implement an allocator that is correct, efficient and fast.*

Introduction

Alyssa Hacker's boss has taken a break from his Minesweeper marathon to ask Alyssa 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 using Pin 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.

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
```

Your group name will consist of the usernames of each of the group's members, separated by dashes. Since this is a group project, we have created only one repository per group. Group members can easily work concurrently on different files or even different parts of the same file and communicate the changes by pushing to the main repository, and having others pull from it. Note that you must commit your changes to your local repository before you can push and pull. Also, remember that committing locally allows you to keep track of smaller sets of changes even if you're not ready to push those changes to your group members.

While pulling your group members' changes you may occasionally see a merge conflict. This occurs when two people edit the same portion of the same file. Normally, git is smart enough to handle these conflicts automatically, but if the changes overlap enough, it will be unable to do so.

The offending file will be modified to show you both versions of the conflicting section(s). Specifically, git will insert `<<<<<<<`, `=======`, and `>>>>>>>` markers within your code. To resolve the conflict, simply delete the markers and edit the code appropriately so that all of your changes are integrated together. Finally, commit your changes to your local repository and (if appropriate) push them.

1 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. (You don't have to explicitly call an initialization function when you use `libc malloc` because this takes place before your `main` function is executed.)

- `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 an error occurs and the requested block could not be allocated, a `NULL` pointer is 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`. The block of memory to which it points must not have been previously freed. Freeing a `NULL` pointer is allowed and has 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`. The return value of `mm_realloc` must meet all of the same constraits 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.

  However, there is one additional constraint on the behavior of `mm_realloc`. For any i between 0 and the size of the old memory block pointed to by `ptr`, byte i of the new block of memory must have the same value as byte i of the old 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.

  You will observe that 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.

2 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. 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 be checked 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.

This consistency checker is for your own debugging during development. When you submit `mm.c`, make sure to remove any calls to `mm_check` in your other calls as this will probably have a significant, negative impact on your performance. 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.

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 generic 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).

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.

Rules and reminders

- You should not change any of the sources in the distribution except for the `Makefile`, `mm.c`, and `mm.h`. You are free to add new files and update the `Makefile` appropriately if you wish. The `config.h`, `memlib.*`, `mdriver.c`, and timer related files will be overwritten with fresh copies during 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` 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.
- For consistency with the `libc malloc` package, which returns blocks aligned on 8-byte boundaries, your allocator must always return pointers that are aligned to 8-byte boundaries. The driver will enforce this requirement for you.

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. 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:

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

  where U is your space utilization, T is your throughput, and $T_{\mathtt{libc}}$ 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 $w = 0.6$. Observing that 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.

- **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 tradeoffs, be sure to discuss the possible advantages and disadvantages of each choice, and explain why you made the decision that you did.

Hints and tips

- Use the `mdriver -f` option. During initial development, using tiny trace files will simplify debugging and testing. We have included two such trace files (short1,2bal.rep) that you can use for initial debugging.
- 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.
- You may want to explore encapsulating your pointer arithmetic in C preprocessor macros. 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 macros 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 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. 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 2009