cs154 Autumn 2019 p5malloc: Writing a Dynamic Storage Allocator
Assigned: November 24th, Due: December 6th at 11:59am (for graduating students; note am, not pm); December 11th at 11:59pm (all others)

November 24, 2019

1 Introduction

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

2 Warning

Bugs can be especially pernicious and difficult to track down in an allocator, and you may easily spend more time debugging than coding in this assignment. We urge you to start immediately.

3 Logistics

You must do this lab on one of the CSIL Linux machines, and not the machine named "machomp."

Start by updating your svn repository to get the `p5malloc` distribution.

The only file you will be modifying is `mm.c`*, which contains your solution.* The `mdriver.c` program is a driver program that allows you to evaluate the performance of your solution. Use the command `make` to generate the driver code and run it with the command `./mdriver`.

4 How to Work on the project

Your dynamic storage allocator will consist of the following functions, which are declared in mm.h and defined in mm.c.

```
int   mm_init(void);
void *mm_malloc(size_t size);
void mm_ free(void *ptr);
void *mm_realloc(void *ptr, size_t size);
void *mm_calloc (size_t nmemb, size_t size);
void  mm_heapcheck(void);
```

The mm.c file we have given you implements nothing. However, we have also provided you with a file called mm-naive.c, which implements everything correctly, but naively. We have also provided you with a working 64-bit version of the implicit list allocator described in your textbook, called mm-implicit.c.

You may use either of these examples as starting points for your own mm.c file. Because we are running on 64-bit machines, your allocator must be coded accordingly, with one exception: the size of the heap will never be greater than or equal to 2^{32} bytes.

Implement the functions (and possibly define other private static functions), so that they obey the following semantics:

- mm_init: Performs any necessary initializations, such as allocating the initial heap area. The return value should be -1 if there was a problem in performing the initialization, 0 otherwise.

  Every time the driver executes a new trace, it resets your heap to the empty heap by calling your mm_init function.

- mm_malloc: The mm_malloc routine returns a pointer to an allocated block payload of at least size bytes. The entire allocated block should lie within the heap region and should not overlap with any other allocated chunk.

  Since the standard C library (libc) malloc always returns payload pointers that are aligned to 8 bytes, your malloc implementation should do likewise and always return 8-byte aligned pointers.

- mm_free: The mm_free routine frees the block pointed to by ptr. It returns nothing. This routine is only guaranteed to work when the passed pointer (ptr) was returned by an earlier call to mm_malloc, mm_calloc, or mm_realloc and has not yet been freed. mm_free(NULL) has no effect.

- mm_realloc: The mm_realloc routine returns a pointer to an allocated region of at least size bytes with the following constraints.

  - 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), and should return NULL;

  - if `ptr` is not NULL, it must have been returned by an earlier call to `mm_malloc` or `mm_realloc`, and not yet have been freed. The call to `mm_realloc` changes the size of the memory block pointed to by `ptr` (the *old block*) to `size` bytes and returns the address of the new block. Note that the address of the new block might be the same as the old block (perhaps there was free space after the old block and it could just be extended, or the new size was smaller than the old size), or it might be different, depending on your implementation, the amount of internal fragmentation in the old block, and the size of the `mm_realloc` request.

    The contents of the new block are the same as those of the old `ptr` block, up to the minimum of the old and new sizes. Everything else is uninitialized. For example, if the old block is 8 bytes and the new block is 12 bytes, then the first 8 bytes of the new block are identical to the first 8 bytes of the old block and the last 4 bytes are uninitialized. Similarly, if the old block is 8 bytes and the new block is 4 bytes, then the contents of the new block are identical to the first 4 bytes of the old block.

- `mm_calloc`: Allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory. The memory is set to zero before returning.

  Note: Your `mm_calloc` will not be graded on throughput or performance. Therefore a correct simple implementation will suffice.

- `mm_checkheap`: The `mm_checkheap` function scans the heap and checks it for consistency. This function will be very useful in debugging your malloc implementation. Some malloc bugs are very hard to debug using conventional gdb techniques. The only effective technique for some of these bugs is to use a heap consistency checker. When you encounter a bug, you can isolate it with repeated calls to the consistency checker until you find the instruction that corrupted your heap.

These semantics match the semantics of the corresponding `libc` routines. (Note that `mm_checkheap` does not have a corresponding routine in `libc`.) Type `man malloc` to the shell for complete documentation.

5 Support Routines

The `memlib.c` package 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 system's page size in bytes (4K)

6 The Trace-driven Driver Program

The driver program `mdriver.c` tests your `mm.c` package for correctness, space utilization, and throughput. The driver program is controlled by a set of *trace files* that are included in the distribution. Each trace file contains a sequence of allocate and free directions, traced from a real program, that instruct the driver to call your `mm_malloc` and `mm_free` routines in some sequence. The driver and the trace files are the same ones we will use when we grade your handin `mm.c` file.

When the driver program is run, it will run each trace file 12 times: once to make sure your implementation is correct, once to determine the space utilization, and 10 times to determine the performance.

The driver `mdriver.c` accepts the following command line arguments. The normal operation is to run it with no arguments, but you may find it useful to use the arguments during development.

- `-t <tracedir>`: Look for the default trace files in directory `tracedir` instead of the default directory defined in `config.h`.
- `-f <tracefile>`: Use one particular `tracefile` instead of the default set of tracefiles for testing correctness and performance.
- `-c <tracefile>`: Run a particular `tracefile` exactly once, testing only for correctness. This option is extremely useful if you want to print out debugging messages.
- `-h`: Print a summary of the command line arguments.
- `-l`: Run and measure `libc` malloc in addition to your malloc package. This is interesting mainly to see how slow the libc malloc package is.
- `-V`: 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.
- `-v <verbose level>`: This optional feature lets you set your verbose level manually to a particular integer.
- `-d <i>`: At debug level 0, very little validity checking is done. This is useful if you're mostly done but just tweaking performance.

  At debug level 1, every array the driver allocates is filled with random bits. When the array is freed or reallocated, we check to make sure the bits haven't been changed. This is the default.

  At debug level 2, every time any operation is done, all arrays are checked. This is very slow, but useful to discover problems very quickly.
- `-D`: Equivalent to `-d2`.
- `-s <s>`: Timeout after s seconds. The default is never to timeout.

7 Programming Rules

- The minimal size to increase the heap by through mem_sbrk must be 4KB, which can be conveniently defined in a macro such as CHUNKSIZE as is done in mm-implicit.c. In other words, every time you need to request more space from the heap through a call to mem_sbrk, the size argument must be at least 4KB, and our grading script will check and enforce this condition. The only exception is the first call to mem_sbrk in mm_init: you are free to use any size to only accommodate for special prologue, epilogue, etc., as is done in mm-implicit.c.

- You should not change any of the interfaces in mm.h, or, indeed, any code outside of mm.c. However, we strongly encourage you to use static functions in mm.c to break up your code into small, easy-to-understand segments.

- You should not invoke any external memory-management related library calls or system calls. This excludes the use of the libc malloc, calloc, free, realloc, sbrk, brk or any other memory management packages in your code. This rule applies not only to the allocations you are being asked to do (i.e., you cannot simply pass through the requests to the system libraries), but also to any accounting information you keep (i.e., you must store all your information in your own heap).

- You are not allowed to define any global data structures such as arrays, structs, trees, or lists in your mm.c program. However, you *are* allowed to declare global scalar variables such as integers, floats, and pointers in mm.c.

  If you need space for large data structures, you can put them at the beginning of the heap.

- You are not allowed to simply hand in the code for the allocators from the CS:APP or K&R books. If you do so you will receive no credit.

  However, we encourage you to study these codes and to use them as starting points. For example, you might modify the CS:APP code to use an explicit list with constant time coalescing. Or you might modify the K&R code to use constant time coalescing. Please remember, however, that your allocator must be for 64-bit machines.

- It is **not** acceptable to copy any code of malloc implementations found online or in other sources, except for the implicit list allocator described in your book.

- We encourage you to study the trace files and optimize for them, but your code must be correct on any trace. The score you get is averaged over all traces marked '*'. The utilization score weights all traces equally, whereas the performance score weights by the number of operations. In other words, if you are worried about speed, optimize for the largest traces.

- 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 check this requirement.

- Your code should compile without warnings. Warnings often point to subtle errors in your code; whenever you get a warning, you should double-check the offending line to make sure the code is really doing what you intended.

8 Evaluation

There are a total of 100 points. 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:

Two metrics will be used to evaluate your solution:

- *Space utilization*: The peak ratio between the aggregate amount of memory used by the driver (i.e., allocated via `mm_malloc` but not yet freed via `mm_free`) and the size of the heap used by your allocator. The optimal ratio equals 1. You should find good policies to minimize fragmentation in order to make this ratio as close as possible to the optimal.
- *Throughput*: The average number of operations completed per second.

The driver program summarizes the performance of your allocator by computing a *performance index*, $0 \leq P \leq 100$, which is a weighted sum of the space utilization and throughput

$$P = 100 * \left(w \min\left(1, \frac{U}{U_{thresh}}\right) + (1 - w) \min\left(1, \frac{T}{T_{thresh}}\right)\right)$$

where U is your space utilization, T is your throughput, and U_{thresh} and T_{thresh} are the estimated space utilization and throughput of an optimized malloc package.[1] The performance index favors space utilization over throughput: $w = 0.6$.

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. 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 only. To receive a good score, you must achieve a balance between utilization and throughput.

The 100 performance points (`$perfpoints`) will be allocated as a function of the performance index (`$perfindex`):

```
if ($perfindex < 60) {
    $perfpoints = 0;
}
else {
    $perfpoints = (40 + ((3 * $perfindex)/5));
}
```

We chose this function so that, when run on a CSIL Linux machine, the CS:APP implicit list allocator receives 0/100 points, a good explicit list allocator can get 100/100 points.

You must run your tests on a CSIL Linux machine (other than "machomp") for the performance indices you see to have any hope of matching the ones we will get when we grade your code. Since

[1] The values for U_{thresh} and T_{thresh} are constants in the driver (0.85 and 5,500 Kops/s) that we established when we configured the program. This means that once you beat 85% utilization and 5,500 Kops/s, your performance index is perfect.

performance varies from hardware configuration to hardware configuration, the performance you might observe on any other machine is **absolutely meaningless** in predicting your score. Any work you do to assess or optimize performance on any machine other than a CSIL Linux machine is, at best, based on misleading data. Note that "machomp" has a different processor than the rest of the machines, and thus cannot be used either.

9 Handin Instructions

Hand in your `mm.c` file by committing it in the `p5malloc` directory of your `CNET-cs154-2017` repository in the same manner you have done for all prior projects. You may submit your solution as many times as you wish up until the due date.

Only the last version you submit will be graded.

10 Hints

- *Use the* `mdriver -c` *option or* `-f` *option.* During initial development, using tiny trace files will simplify debugging and testing. The first several traces that `mdriver` runs are such small trace files.
- *Use the* `mdriver -V` *options.* The `-V` option will also indicate when each trace file is processed, which will help you isolate errors.
- *Use the* `mdriver -D` *option.* This does a lot of checking to quickly find errors.
- *Compile with* `gcc -g` *and use a debugger.* A debugger will help you isolate and identify out of bounds memory references. You may want to modify the Makefile and remove the -O2 option during initial testing. But don't forget to restore the original optimizations when doing performance testing.
- *Use gdb's* `watch` *command* to find out what changed some value you didn't expect to have changed.
- *Understand every line of the implicit list malloc implementation in the textbook.* A working 64-bit version of this allocator is in `mm-implicit.c`.
- *Encapsulate your pointer arithmetic in C preprocessor macros or inline functions.* Pointer arithmetic in memory managers is confusing and error-prone because of all the casting that is necessary. You can reduce the complexity significantly by writing macros for your pointer operations. See the text for examples.
- *Remember we are working with 64-bit machines.* All pointers are 8 bytes, as is `size_t`.
- *Use your heap consistency checker.* A good heap consistency checker will save you hours and hours when debugging your malloc package. You can use your heap checker to find out exactly where things are going wrong in your implementation. Make sure that your heap checker is detailed. Your heap checker should scan the heap, performing sanity checks and possibly printing out useful debugging information. Every time you change your implementation, one of the first things you should do is think about how your `mm_heapcheck` will change, what sort of tests need to be performed, etc.

- *Use a profiler.* You may find the `gprof` tool helpful for optimizing performance.
- *Versioning your implementation.* There is nothing worse than making a change that you think will improve your score, discovering that it makes things worse, and then having trouble backing out those changes. You could end up spending considerable time and effort just trying to get bak to where you were before. This is exactly the kind of problem that svn was designed to solve. Commit early, often, and whenever you have a working allocator, particularly before you embark on a new initiative.
- *Start 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 in your career. So start early, and good luck!

11 More Hints

Basically, you want to design an algorithm and data structure for managing free blocks that achieves the right balance of space utilization and speed. Note that this involves a tradeoff. For space, you want to keep the internal data structures small. Also, while allocating a free block, you want to do a thorough (and hence slow) scan of the free blocks, to extract a block that best fits our needs. For speed, you want fast (and hence complicated) data structures that consume more space. Here are some of the design options available to you:

- Data structures to organize free blocks:
  - Implicit free list (the free blocks are part of the list of all blocks, and you need to scan over all blocks to find the free ones)
  - Explicit free list (the free blocks are in their own list, separate from the allocated blocks; finding a free block to use for an allocation need only involve scanning this shorter list)
  - If you like a challenge, you can try to implement a segregated list allocator which usually outperforms an explicit list allocator. Segregated free lists (explicit free lists that are segregated by size of block, so that an allocation only considers free blocks that are approximately the needed size, rather than a large number of irrelevant free blocks)
- Algorithms to scan free blocks:
  - First fit/Next fit
  - Blocks sorted by address with first fit
  - Best fit

You can pick (almost) any combination from the two. For example, you can implement an explicit free list with next fit, a segregated list with best fit, and so on. Also, you can build on a working implementation of a simple data structure to a more complicated one.

In general, we suggest that you start with an implicit free list, then change this to an explicit list, and experiment with different policies. A good explicit list implementation should be good enough to achive 100 perfpoints, but you are welcome to optimize the implementation even further by using segragated lists.

12 Acknowledgement

This project is based on one developed by the authors of our textbook.