COMP 321: Introduction to Computer Systems

Project 5: Malloc Dynamic Memory Allocator
Assigned: 3/23/15, **Due: 4/1/15**

Important: This project may be done individually or in pairs. Be sure to carefully read the course policies for assignments (including the honor code policy) on the assignments page of the course web site:

`http://www.clear.rice.edu/comp321/html/assignments.html`

Overview

In this lab you will be writing a dynamic memory 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.

Project Description

Your dynamic memory allocator will consist of the following four 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);
```

The `mm.c` file that we have given you implements a simple memory allocator based on an implicit free list, first fit placement, and boundary tag coalescing, as described in the CS:APP2e text. Using this as a starting place, modify these functions (and possibly define other private `static` functions), so that they obey the following semantics:

- `mm_init`: 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` to perform 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, and 0 otherwise.

  The driver will call `mm_init` before running each trace (and after resetting the `brk` pointer). Therefore, your `mm_init` function should be able to reinitialize all state in your allocator each time it is called. In other words, you should not assume that it will only be called once.

- `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.

We will compare your implementation to the version of `malloc` supplied in the standard C library (`libc`). Since the `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` or `mm_realloc` and has not yet been freed.

- `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);`
  - if `ptr` is not NULL, it must have been returned by an earlier call to `mm_malloc` or `mm_realloc`. 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. The address of the new block might be the same as the old block, or it might be different, depending on your implementation, the amount of internal fragmentation in the old block, and the size of the `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.

These semantics match the the semantics of the corresponding `libc` `malloc`, `realloc`, and `free` routines. Type `man malloc` for complete documentation.

Heap Consistency Checker

Dynamic memory allocators are notoriously tricky to program correctly and efficiently. They are difficult to program correctly because they involve a lot of untyped pointer manipulation. You will find it very helpful to write a heap checker that scans the heap and checks it for consistency.

Some examples of what a heap checker might check are:

- 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?

Your heap checker will consist of the function `void checkheap(bool verbose)` in `mm.c`. This function should check any invariants or consistency conditions that you consider prudent. It should print out a descriptive error message when it discovers an inconsistency in the heap. You are not limited to the listed suggestions nor are you required to check all of them.

This consistency checker is intended to help you with debugging your memory allocator during development. However, the provided implementation of `checkheap` is only suited to a memory allocator that is based on an implicit free list. So, as you replace parts of the provided memory allocator, you should update the implementation of `checkheap`. Style points will be given for your `checkheap` function. Make sure to put in comments and document what you are checking.

When you submit `mm.c`, make sure to remove any calls to `checkheap` as they would likely reduce your throughput score!

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(intptr_t 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. If there is an error, it returns (void *)-1. 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 on x86-64 Linux systems).

Provided Files

The provided files are all available in

```
/clear/www/htdocs/comp321/assignments/malloc/
```

To begin working on the project, do the following:

- Copy all of the files in this directory to the directory in which you plan to do you work.
- Type your team member names in the header comment at the top of `mm.c`.
- Type the command `make` to compile and link a basic memory allocator, the support routines, and the test driver.

Looking at the `mm.c` file, you will see that it contains a functionally correct (but very poorly performing) memory allocator. Your assignment is to modify this file to implement the best memory allocator that you can.

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 available in the comp321 course directory (`config.h` indicates their location). Each trace file contains a sequence of allocate, reallocate, and free directions that instruct the driver to call your `mm_malloc`, `mm_realloc`, and `mm_free` routines in some sequence. The driver and the trace files are the same ones that we will use when we grade your `mm.c` file.

The driver `mdriver.c` accepts the following command line arguments:

- `-t <tracedir>`: Look for the trace files in directory `tracedir` instead of the default directory defined in `config.h`.
- `-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 your 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.

Programming Rules

- You should not change any of the interfaces in `mm.c`.
- You should not invoke any memory-management related library calls or system calls. Therefore, you may not use `malloc`, `calloc`, `free`, `realloc`, `sbrk`, `brk` or any variants of these calls in your code.
- You are not allowed to define any global or `static` variables that are arrays or structs in your `mm.c` program. However, this does not mean that you are prohibited from using arrays and structs, only that the memory for holding them must come from your heap. You *are* allowed to declare a small number of scalar global variables such as integers, floats, and pointers in `mm.c`.
- 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.

Hints

- *Use the* `mdriver -f` *option.* During initial development, using tiny trace files will simplify debugging and testing. We have included two such trace files (`short{1,2}-bal.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.

- *Compile with* `gcc -g` *and use a debugger.* A debugger will help you isolate and identify out of bounds memory references.
- *Understand every line of the provided malloc implementation.* The lecture notes and the textbook describe how this simple implicit free list allocator works. Don't start working on your own allocator until you understand everything about this simple allocator.
- *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 `malloc` and `free` routines working correctly and efficiently on the first 9 traces. Only then should you turn your attention to the `realloc` implementation. The provided `realloc` implementation works by simply calling your `malloc` and `free` routines. But, to get really good performance, you will need to build a smarter `realloc` that calls `malloc` and `free` less often.
- *Don't forget what you've learned before.* There are many ways to write the code to manipulate pointers to insert and remove free blocks from a free list. The most complex and error-prone way would be to use the provided macros to try and manipulate raw memory as pointers. A better way would be to define a `struct` that matches the structure of your free blocks and cast your heap pointers into pointers to that `struct`. This is a common and important idiom in C. Where have you done something like that before?
- *Use a profiler.* You may find the `gprof` tool helpful for optimizing performance. We are using `gcc` for this assignment precisely to enable you to do this, as the current version of `clang` on CLEAR has some incompatibilities with `gprof`.
- *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!

Evaluation

To turnin your code, be sure that your code and writeup are contained in files named `mm.c` and `writeup.txt`. Note that we will only use your `mm.c` along with the original files that were handed out to test your code, so you should not add or modify code in any other file (except during testing, as necessary).

To turnin your solution, you should use the `turnin` process on CLEAR (`https://docs.rice.edu/confluence/x/qAiUAQ`). First, create a `malloc` directory within your `comp321` directory. Then, be sure that your solution is contained in a file named `mm.c` and your writeup is contained in a file named `writeup.txt` inside this `malloc` directory. Finally, submit your solution by running the following command from inside your `comp321` directory:

```
UNIX% turnin comp321-S15:malloc
```

After which, you should receive an e-mail confirming the submission of your solution. Assignments can be submitted multiple times. Grading will always be done on the most recent submission before the deadline, unless slip days are used. If you would like, you can also upload additional files with testing code. **However, remember that we will *only* compile your mm.c file and nothing else!**

The project will be graded as follows:

- *Performance (70 points).* You will receive **zero points** for performance if your solution fails any of the correctness tests performed by the driver program, if you break any of the programming rules, or if your code is buggy and crashes the driver program.

  Otherwise, your performance score will be based on the following two metrics:

  - *Space utilization*: The peak ratio between the aggregate amount of memory used by the driver (i.e., allocated via `mm_malloc` or `mm_realloc` but not yet freed via `mm_free`) and the size of the heap used by your allocator. The optimal ratio equals to 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*, P, which is a weighted sum of the space utilization and throughput

  $$P = wU + (1 - w) \min\left(1, \frac{T}{T_{target}}\right)$$

  where U is your space utilization, T is your throughput, and T_{target} is the throughput of a reasonable malloc implementation on CLEAR on the default traces.[1] The performance index favors throughput over space utilization, with a default of $w = 0.4$.

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

  Note that your utilization score will remain the same on all CLEAR machines, but your throughput score may vary with the load on the particular machine that you are using (we will use an otherwise unloaded machine for grading).

  The provided implementation already achieves a performance index of $30/100$. Since your assignment is to build a better memory allocator than we provided to you, your performance score will be the performance index of your allocator minus 30.

  Do not expect to receive all 70 performance points. While it is relatively easy to achieve high throughput, it is much more difficult to achieve high utilization. Further, achieving higher utilization typically means that you will lower your throughput. The best solution that we have would receive 69 points. A less aggressive solution of ours would receive 56 points.

- Style (20 points). This includes general program style and the thoroughness and documentation of your heap consistency checker `checkheap`.

- Writeup (10 points).

[1] The value for T_{target} is a constant in the driver (22,500 Kops/s).