CSCE 351
Malloc Assignment : Writing a Dynamic Storage Allocator
Due: See Canvas for more information

1 Introduction

In this programming assignment 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 most importantly correct. It should also be memory efficient and fast.

2 Logistics

Since this assignment is quite large, I will allow you to work in team. However, if you choose to work alone, you can do so for a 10% bonus. (I will add 10% to the total points you've earned.) (I will detect 10% from the total points your team has earned.) You will have a choice of building two systems. One will have the limit of 2 members per team. Any clarifications and revisions to the assignment will be posted on canvas.

3 Hand Out Instructions

Start by downloading *malloc-assignment.zip* to a protected directory in CSCE.UNL.EDU (**Note that the system is CSCE and not CSE.**). Then issue the command: `unzip malloc-assignment.zip`. This will cause a number of files to be unpacked into the directory called *malloc-assignment*. The only file you will be modifying and handing in is `mm.c`. The `mdriver.c` program is a driver program that allows you to evaluate the correctness and performance of your solution. Use the command `make` to generate the driver code and run it with the command `./mdriver -V`. (The `-V` flag displays helpful summary information.) `./mdriver -lV` also reports the performance of the dynamic memory management routines from standard C library (glibc).

Looking at the file `mm.c` you'll notice a C structure `team` into which you should insert the requested identifying information about your team. **Do this right away so you don't forget.**

When you have completed the lab, you will hand in only one file (`mm.c`), which contains your solution.

4 How to Work on the Lab

Your dynamic storage 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 we are providing, implements a simple implicit list with no boundary tags. As such, traversing the list can only be done in one direction. We only implement `mm_init` and `mm_malloc`. The latter is working but poorly utilizing memory and operating slowly. Thus, when we use it with the provided memory allocation traces, it can only pass 6 of the 11 tests. It fails 3 tests due to out-of-memory and 2 tests due to lack of support for `mm_realloc`.

Using this as a starting place, implement `mm_free` and `mm_realloc`. The semantic of each function is described below.

- `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, the staring address of the heap otherwise.
- `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 (`glibc`). Since the `glibc` malloc always returns payload pointers that are aligned to 8 bytes, your malloc implementation should do likewise and always return 8-byte aligned pointers. Our *mdriver* program tests for 8-byte alignment and terminates if the alignment check fails.
- `mm_free`: The `mm_free` routine frees the block pointed to by `ptr`. It returns nothing. **This routine is only guaranteed to work correctly 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. Notice that 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. Issue `man malloc` command to a shell for complete documentation.

5 Heap Consistency Checker

Dynamic memory allocators are notoriously tricky beasts 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 basic functions that a heap checker should support 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?

We provide you with a heap checker (`mm_checkheap`) that works with the current implementation of `mm_malloc`. That is, it can check the header information to identify the size of the block and its allocation status. You can use this to help debug your implementation. When you submit `mm.c`, make sure to remove any calls to `mm_check` as they will slow down your throughput.

6 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 on Linux systems).

7 The Trace-driven Driver Program

The driver program `mdriver.c` in the `malloc-assignment.zip` distribution 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 `malloc-assignment.zip` distribution. 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 we will use when we grade your handin `mm.c` file.

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

- `-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` 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 student's 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.

8 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. This excludes the use of `malloc`, `calloc`, `free`, `realloc`, `sbrk`, `brk` or any variants of these calls in your code.

- 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.
- Over the years, we have collected implementations of similar work that have been released on the Internet. While it has been argued that your implementation may be similar to those that are publicly available, we have made several modifications to the provided implementation of `mm.c` so that your implementation would be different from those that are out there. As such, we will use MOSS (`https://theory.stanford.edu/~aiken/moss/`) to compare your work against our collection that includes those programs available on the Internet and from our prior semester. If your implementation is flagged as being similar to our collection, you will be reported to CSCE Academic Integrity Committee for Investigation without exceptions. For more information about CSCE Academic Integrity Policies, please visit: `https://cse.unl.edu/academic-integrity-policy`.

9 Evaluation

You will receive **zero points** if you break any of the rules or your code is buggy and crashes the driver. Otherwise, there are two levels of implementation that you can choose from. Since the maximum numbers of team members differ, you will need to choose when you sign up the team (if you have 4 members, you MUST build an Explicit List System).

9.1 Implicit List System (maximum marks: 75 out of 100 points)

The easier path to earn a decent amount of points is to simply extend the implicit list implementation provided to you to support the missing functions. If you choose to build this system, you would only earn at most 75 out of 100 points but the amount of time you spend to complete the project is also significantly less than that to build the more advanced system. The requirements for this system are:

- The system must not use boundary tags. Traversing the list can only go in one direction.
- You cannot change `mm_init` and `mm_malloc`. Simply add code to complete `mm_free` and `mm_realloc`.
- `mm_free` must support coalescing. Without it, your system will not pass all 11 tests.
- `mm_realloc` must support all conditions stated as part of the reallocation semantics.
- There is a limit of 2 members per team if you choose to build this system. Working alone still earns you a 10% bonus. Working in pair is allowed for no bonus.

Since this system will perform slowly, you will only be graded on correctness. To earn all 75 points, your system must pass all 11 tests. You will receive partial credit for operating correctly on each trace.

9.2 Explicit List System (maximum marks: 100 points)

You explicit list system must support boundary tags and can traverse the heap in both forward and backward directions. It must support coalescing so that you can pass all tests. Please refer to *lecture5.pptx* for more information about how to implement an explicit list system. The requirements for this system are:

- You cannot significantly change `mm_init`. Ideally, you should only need to add a boundary tag to the initial block. You can use the current empty payload of the first block for this purpose.
- You can choose a criterion to insert a free block into the list (e.g., LIFO or address ordered). The criterion should not affect correctness but can yield better performance for your system. This can lead to more performance points.
- Your `mm_realloc` implementation must follow the previously stated semantics.
- `mm_free` must support coalescing. Without it, your system will not pass all 11 tests.
- There is a limit of 2 members per team if you choose to build this system. Working alone still earns you a 10% bonus. Working in a team of 2 is allowed for no bonus.

Next, we describe the scoring rules for this system.

- *Correctness (75 points).* You will receive full points if your solution passes the correctness tests performed by the driver program. You will receive partial credit for each correct trace.
- *Performance (25 points).* Two performance 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` 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

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

  where U is your space utilization, T is your throughput, and T_{libc} is the estimated throughput of `libc` malloc on your system on the default traces.[1] 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

[1] The value for T_{libc} is a constant in the driver (600 Kops/s) that your instructor established when they configured the program.

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.

10 Programming Style

You can lose points for poorly designed and written programs. As such, adhere to the following styling rules.

- Your code should be decomposed into functions and use as few global variables as possible.
- Your code should begin with a header comment that describes the structure of your free and allocated blocks, the organization of the free list, and how your allocator manipulates the free list. You can use the current header comment section in `mm.c` as an example. Each function should be preceded by a header comment that describes what the function does.
- Each subroutine should have a header comment that describes what it does and how it does it.
- Your heap consistency checker `mm_checkheap` should be thorough and well-documented. While we will not test it, it can help your productivity during the development of your system.

11 Handin Instructions

1. You will submit your `mm.c` file via a web interface in Canvas.
2. When testing your system, make sure to use CSCE. This will insure that the performance report you get from `mdriver` is representative of the performance report you will receive when we grade your solution.

12 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 (`short1,2-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 the general concept of malloc implementations in the lecture notes.* The lecture note has a detailed example of a simple allocator based on an implicit free list (*lecture4.pptx*). The provided `mm.c` also has extensive comments. Use this is a point of departure. Don't start working on your allocator until you understand everything about the provided simple implicit list allocator.

- *Encapsulate your pointer arithmetic in C preprocessor macros.* 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. We provide you with several working macros in *memhelper.h*. Use them as the starting point to write your own macros.

- *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. For starters, build `realloc` on top of your existing `malloc` and `free` implementations. But to get really good performance, you will need to build a stand-alone `realloc`.

- *Use a profiler.* You may find the `gprof` tool helpful for optimizing performance.

- *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!