CS 213, Fall 2000
Lab Assignment L4: Dynamic Storage Allocator
Assigned: Nov. 2, Due: Wed. Nov. 15, 11:59PM

Logistics

Tiankai Tu (`ttu@andrew.cmu.edu`) is the lead person for this assignment.

The tar file for this assignment can be retrieved from

`/afs/cs.cmu.edu/academic/class/15213-f00/L4/L4.tar`

Once you've copied this file into a (private) directory, run the command `tar xvf L4.tar` and fill in your team information in the structure at the beginning of the file `malloc.c`. When you have completed the lab, you will hand in only one file (`malloc.c`) that contains your solution. As usual, you may work in a group of up to 2 people.

Introduction

In this lab 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.

Your dynamic storage allocator will consist of the following four functions, which are declared in `malloc.h` and defined in `malloc.c` with empty function bodies.

```
int   mm_init(void);
char *mm_malloc(size_t size);
void  mm_free(void *block);
void *mm_realloc(void *ptr, size_t size);
```

You should fill in these empty function bodies (and possibly define other private functions) so that they obey the following semantics:

- `mm_init`: Before calling `mm_malloc` `mm_realloc` or `mm_free`, the application program calls `mm_init` to perform any necessary initializations, including the allocation of the initial heap area. The return value should be -1 if there was a problem in performing the initialization, 0 otherwise.

- `mm_malloc:` The `mm_malloc` routine returns a pointer to an allocated chunk of memory of at least `size` bytes. The pointer must be aligned to 8 bytes. The entire allocated chunk should lie within the heap region and should not overlap with any other allocated chunk.
- `mm_free:` The `mm_free` routine frees the block pointed to by `block`. It returns nothing. This routine is only guaranteed to work when the passed pointer (`block`) 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 Linux `malloc`, `realloc`, and `free` routines. Type `man malloc` to the shell for complete documentation.

Support routines

The `memlib.c` package is provided to facilitate your implementation of a dynamic memory allocator. You can invoke the following three functions in `memlib.c`.

- `mem_sbrk:` expands the heap area. The lower and upper boundaries of the heap area are contained in `dseg_lo` and `dseg_hi` respectively. You are allowed to read these variables, but you should not modify them in any way. You must call `mem_sbrk` in order to change the upper bound. This function accepts a positive integer argument, which is the amount of bytes by which the upper bound should be expanded. The return value is the beginning of the newly allocated heap area, or `NULL` if there wasn't any memory left. You cannot decrease the heap area in size, only increase it, so be careful how you call `mem_sbrk`.
- `mem_pagesize:` returns the system page size.
- `mem_usage:` returns the current size of the heap in bytes.

Driver and traces

The `driver.c` program in the L4.tar distribution contains the actual driver program we will use to test your allocator. Feel free to use any other testing method you wish while developing your code. The test driver should provide you with some useful information for debugging your program. The command line options it accepts are as follows:

`-f` *tracefile* ...	Use a particular tracefile for testing; can repeat this option to load multiple tracefiles. If no tracefiles are specified, the default set of tracefiles is used.
`-v`	Verbose mode; print out some detailed debugging info (default).
`-q`	Quiet mode.
`-h`	Print a help (usage) message.
`-c`	Run the system (`libc`) malloc in addition to yours and report the throughput stats for both.
`-C`	Run the system malloc by itself and report its throughput stats.
`-d`	Generate a dump of your allocator's internal operation (e.g. the pointers it returns and the `mem_sbrk` calls it initiates) into a text file.
`-t` *tolerance*	Specify a relative error tolerance for the time measurements (default: 0.05)

Please avoid excessive use of the `-c` and `-C` flags especially when a lot of users are sharing the same machine for development. These options are implemented to provide you with a basis for comparison and should be avoided under high load. Because they cause the program to consume a significant amount of memory, their excessive use can make machine sharing inconvenient when the number of concurrent users is high.

We are also providing you with all the traces that we will use to evaluate your allocator. These can be found at `/afs/cs/academic/class/15213-f00/L4/traces/`.

Rules

- You are not allowed to change the interfaces.
- You should not invoke 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.
- You should not declare any global or static data in your `malloc.c`. Any control data you need should be stored on the heap area.
- You should not make use of any other functions in `memlib.c` except for the three functions listed in the "Support routines" section.

Hints

- *Write debugging functions.* Dynamic memory allocators are notoriously tricky beasts to program correctly and efficiently. They are difficult to program correctly because they involve a lot of `(void *)` pointer references. It may be helpful to write functions that print the state of your allocator's data structures which you can use when debugging your program.

- : *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 standalone `realloc`.
- *Use the* `-d` *driver option.* For debugging purposes, you may find the `-d` option helpful. Feel free to modify the dump routines to report more information if you like.
- *Use the* `-f` *driver option.* During initial development, using shorter traces may simplify debugging and testing. We have placed two short trace files in `L4/traces/short/short{1,2}.rep` that you can use by invoking the `-f` option of the driver.
- *Use a profiler.* When optimizing performance, you may find the `gprof` tool helpful. This tool produces an execution profile of your program. It calculates the amount of time spent in each routine. To use `gprof`, you will need to turn on the `gprof` flags when compiling your program:

  ```
  bass>gmake clean
  bass>gmake GPROF=-pg
  bass>gmake GPROF=-pg
  ```

  When you run your executable, say `malloc`, a file named `gmon.out` is created in your current working current directory. To view the profile information in this file, you can invoke `gprof` as follows:

  ```
  bass>gprof malloc gmon.out
  ```

Evaluation

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

- Correctness (20 points). You will receive full points if your solution satisfies the semantic requirements. You will receive partial for correct implementations of `malloc` and `free`. The driver program will check these correctness criteria for you.
- Performance (35 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 close to the optimal.

- *throughput*: average number of operations finished per second. (In this lab, higher throughput translates to shorter response time, but generally this is not the case.)

The performance of your allocator is summarized by a performance index, P, which is a weighted sum of the utilization and throughput metrics. The index favors space utilization over throughput, with a default $w = 0.65$:

$$P = w\frac{U}{U_{opt}} + (1 - w)Min(1, \frac{Throughput}{Throughput_{libc}}) = wU + (1 - w)Min(1, \frac{Throughput}{Throughput_{libc}})$$

where U is your space utilization, $Throughput$ is your throughput, U_{opt} is 1, and $Throughput_{libc}$ is the throughput of `libc` malloc.

Observing that both memory space and CPU cycles are expensive system resources, we adopt this formula to encourage balanced optimization of both. 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 extremes to optimize either the memory utilization or the throughput only.

- Style (5 points). Your code should be readable and well commented. Define macros or subroutines where necessary to make the code more understandable.

Handin

See the web page for this lab for how to hand in your solution to the web server.