Project #3: Dynamic Memory Allocator

Today: June 1, 2023
Submission Due: June 23, 2023
(Late submission allowed until June 25, 2023)

1   Introduction

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

2   Materials

You can start the assignment by downloading `prj3-malloc.tar` from cyber campus to a directory in which you plan to do your work. Then give the command `tar -xvf prj3-malloc.tar`. This will cause a number of files to be unpacked into the directory.

- **Requirements:** The only file you will be modifying and turning in is `mm.c`, which contains your solution.

- **Test Driver Program:** The `mdriver.c` program is a driver program that allows you to locally evaluate the performance of your solution in the same way that TA will evaluate your final submission. Use the command `make` to generate the driver code and run it with the command `./mdriver`.

3   Project Description

Your dynamic memory 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);
```

The `mm.c` file we have given you implements everything correctly but naively. 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 initialization, such as allocating 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 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 be comparing 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.

  1. if **ptr** is NULL, the call is equivalent to **mm_malloc(size)**;

  2. if **size** is equal to zero, the call is equivalent to **mm_free(ptr)**.

  3. 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. Type **man malloc** to the shell for complete documentation. Lastly, read the README file carefully, since it contains lots of useful information

4   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 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?

One of your possible heap checkers will consist of the function **int mm_check(void)** in **mm.c**. It will check any invariants or consistency conditions you consider prudent. It returns 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** as they will slow down your throughput.

5   Supporting 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).

6   The Trace-driven Driver Program

The driver program `mdriver.c` in the `prj3-malloc.tar` 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 `prj3-malloc.tar` 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 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.

7   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.

- You are not allowed to define any global or `static` compound 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`.

- 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.

8   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** (20 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. (determined based on the validity of the -V option.)

- **Performance** (35 points). Two performance metrics will be used to evaluate your solution:

  1. *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.

2. *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, T_{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 only. To receive a good score, **you must achieve a balance between utilization and throughput.** Note that you don't have to achieve P = 1 to get the full 35 points. Also, even if you achieve P=0.5, it doesn't mean you will get half of 35 points. The exact formula for your performance score will be announced later.

- **Document** (10 points): You are required to write a document about your `malloc` package. The document will be including the following items at least.

  - Design of your allocator with any type of figures, i.e. flow chart, depiction of an architecture, ...

  - Description of subroutines, structs, any global variables (use global variables as few as possible) that you newly define in your code

9 Submission Guideline

9.1 Put your name on `mm.c`

```
/**********************************************************
 * NOTE TO STUDENTS: Before you do anything else, please
 * provide your information in the following struct.
 **********************************************************/
team_t team = {
    /* Your student ID */
    "20XXXXXX",
    /* Your full name*/
    "Gildong Hong",
    /* Your email address */
    "example@sogang.ac.kr",
};
```

Figure 1: Declaration of `struct team_t team` in `mm.c`

In `mm.c`, there will be `struct team_t team` where you are required to put your ID, name, email address as shown in Figure 1. If you don't update this struct with your information, you will get some penalty for that.

9.2 How to Submit

You are required to submit only your `mm.c` with the document. The name and extension of a document should be `document.pdf`. All materials to be submitted should be in the directory with the name of your **student id** and archived with linux `tar` command. The name of a tarball should be `prj3_<your_student_id>`.tar.gz. You should hand-in the project through the assignment tab on the cyber campus.

   Please strictly follow the submission instruction below. If you not, you may get a penalty on your overall project score.

1. You have to submit one archive file, and the archive file **(.tar.gz)** to be submitted should be named as prj3_*your_student_id*.tar.gz.

2. If you unarchive the file, the name of the root directory of the tar file should be your student id.

Following is the example of a submission.

```
$~> ls
20201234
$~> tar -zcvf prj3_20201234.tar.gz 20201234
$~> ls
prj3_20201234.tar.gz
$~> tar -zxvf prj3_20201234.tar.gz
$~> ls
20201234   prj3_20201234.tar.gz
$~> cd 20201234
$~>20201234> ls
document.pdf   mm.c
```

There will be 5% off penalty on your grade if you don't strictly follow the instruction.

Note:Please make sure to compile your source code on cspro(NOT cspro1˜10) server, as the TA will build and compile your submitted project on that server. If the submitted code does not compile it cannot be scored!!!

10   Miscellaneous

10.1   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 every line of the `malloc` implementation in the textbook.** The textbook has a detailed example of a simple allocator based on an implicit free list. Use this is a point of departure. Don't start working on your allocator until you understand everything about the 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. See the text for examples.

- **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. (The order of the trace is the sequence of the test that is performed when mdriver -V is executed.) 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!*