Assignment 3: Dynamic Memory Allocator

C S 429, Spring 2022
Unique IDs: 51215, 51220, 51225, 51230, 51235, 51240
Lead TA: Srikar Ganti

Assigned: Saturday, 19 February 2022 12:00 CT
Checkpoint: Friday, 25 February 2022 23:59 CT
Due: Friday, 4 March 2022 23:59 CT
Last possible hand-in: Sunday, 6 March 2022 23:59 CT

1 Introduction

In this lab you will be writing a dynamic storage allocator for C programs, i.e., your own version of the `malloc` and `free` routines. Explore the design space creatively. You must implement an allocator that is correct, space-efficient, and high-throughput.

2 Logistics

You are required to complete two submissions: a checkpoint, due Friday, 25 February 2022; and the final hand-in, due Friday, 4 March 2022. The grading details for each submission are given in Section 6.

Start early enough to get things done before the due dates. Assume things will not go according to plan, and so you must allow extra time for heavily loaded systems, dropped internet connections, corrupted files, traffic delays, minor health problems, *force majeure*, etc.

Unless otherwise stated, every project in C S 429 is an individual project. Before you begin, please take the time to review and abide by the course policy on academic integrity at: `https://www.cs.utexas.edu/academics/conduct`.

All hand-ins are electronic. The testing for your code is automated, using OS and compiler configurations similar to those on the UTCS Linux machines. *Test your code thoroughly on a UTCS Linux machine before submitting it.*

Any clarifications or corrections for this lab will be posted on Piazza. You should also ask your questions on that forum. Unless it's an implementation-specific question (i.e., private to instructors), please post it on Piazza publicly so that students with similar questions can benefit as well.

3 Download and setup

First, go to our GitHub Classroom page and create a copy of the assignment: `https://classroom.github.com/a/cUn5yF_C`. This will automatically create an assignment repository specific to you on GitHub. Now clone this created repository to `~/cs429/projects/proj3` on a UTCS lab machine using the command `git clone your-repo-url`. Ask a TA for help if you are having trouble with these steps.

Before you forget, run `./install.sh` in this directory. This will install some Python libraries required to successfully run the test script. And also update the file `umalloc.c` with your name and UT EID as specified, and save your changes.

The only files you will modify and submit are `umalloc.h`, `umalloc.c`, and `check_heap.c`. You will do this as in Assignment 2, by linking your GitHub repository to Gradescope.

The file `runner.c` contains source for a driver program that allows you to evaluate the performance of your solution. Use the command `make` to generate the driver program, and run it with the command `./runner`. The `-h` flag displays the various command-line arguments that you can present to the driver.

Important: Do this assignment **only on the UTCS public machines**. Do not try to do any development on your local desktop or laptop.

4 Details of the assignment

Your dynamic storage allocator will consist of the following three functions declared in `umalloc.h` and implemented in `umalloc.c`.

```
int   uinit(void);
void* umalloc(size_t size);
void  ufree(void* ptr);
```

The supplied file `umalloc.c` provides a handful of useful data structure declarations and utility methods, plus placeholders for these three functions. Using this file as a starting template, modify these functions (and optionally define other `static` functions), so that they obey the following semantics. For the checkpoint, you are required to comment each of the function headers in `umalloc.h` explaining what they do.

- `uinit`: Before calling `umalloc` or `ufree`, the application program (i.e., the trace-driven driver program that you will use to evaluate your implementation) calls `uinit` 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, 0 otherwise.

- `umalloc`: The `umalloc` 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 standard `malloc` on x64/GNU/Linux systems returns payload pointers that are aligned to 16 bytes, your `umalloc` implementation must do likewise and always return 16-byte aligned pointers.

- `ufree`: The `ufree` routine frees the block pointed to by `ptr`. It returns nothing. This routine is guaranteed to work only when the passed pointer `ptr` was returned by an earlier call to `umalloc` and has not yet been freed.

These semantics match the the semantics of the corresponding standard `malloc` and `free` routines. Type `man malloc` to the Linux shell for complete documentation.

4.1 Heap consistency checker

Dynamic memory allocators are notoriously tricky beasts to program correctly and efficiently. They are difficult to program correctly because they can involve a lot of untyped pointer manipulation. We therefore require you to write a heap checker that scans the heap and checks it for consistency. The interface declared in the file `check_heap.h` is `int check_heap(void);` you will implement this routine in the file `check_heap.c`.

4.1.1 Required checks

You must implement the following three checks.

1. Check that pointers in the free list point to valid free blocks. Blocks should be within the valid heap addresses: look at `csbrk.h` for some clues. They should also be allocated as free.
2. Check if any memory blocks (free and allocated) overlap with each other. (*Hint*: Examine the heap sequentially and check that for every memory block b, the "next" memory block of b has a sensible block size and is within the valid heap addresses.)
3. Check that each memory block is 16-byte aligned.

Your implementation of `check_heap()` must check for the *three* invariants or consistency conditions listed above. It should return a zero value if and only if your heap is consistent, and a non-zero value if not. You may use `assert()` macros to check for inconsistent heaps. **However, it is not acceptable to simply print out all the blocks in your free list as your implementation of the heap consistency checker.** Comment your implementation to identify what conditions you are checking to conclude that the heap is consistent. This consistency checker must be completed in order for you to pass the checkpoint.

4.1.2 Optional checks

Here is a list of additional checks you could implement if you wanted to.

- Is every block in the free list marked as free?
- Is every free block on the free list?
- Are all bytes obtained from the operating system through `sbrk()` calls accounted for in either an allocated block or a free block?
- If you implement coalescing: are there contiguous free blocks that somehow escaped coalescing?
- If you maintain the free list in memory order: are you maintaining that order after inserting a freed block into the free list?

If you would like to implement more checks, you are not limited to the listed suggestions, nor are you required to check all of them. Many of the suggestions, especially those that require checking allocated blocks, may require additional variables or helper functions to implement. **Note that you will not earn any extra credit for implementing additional checks. They are purely intended to help you put your heap through more stringent tests for structural integrity.**

4.2 Support routine

The `csbrk.c` file provides a thin wrapper (aka a "shim") around the standard Linux system call `sbrk` that provides a chunk of memory to your allocator. You can do a `man sbrk` to learn more about this routine. However, you are not to call it directly, but only through the provided wrapper.

The declared interface for `csbrk` is `void* csbrk(intptr_t increment);`. The argument `increment` is the amount of memory your allocator is requesting from the operating system. The returned value is a pointer to the starting address of a contiguous range of memory of at least this size. A return value of `NULL` indicates an error condition. Keep a few things in mind as you use this call.

- This is a wrapper around a *system call*. Therefore, do not call it frequently.
- Ask for a decent amount of space each time, e.g., a small multiple of `PAGESIZE` (a macro defined in `csbrk.h`).
- While each call to `csbrk` returns a contiguous chunk of memory, successive calls to `csbrk` are not guaranteed to return adjacent chunks of memory. *This is a significant change to the underlying assumptions of many memory allocators available through a web search; in particular, any allocator using an implicit free list will almost certainly fail.*

4.3 The trace-driven driver program

The driver program `runner` created from the source file `runner.c` in the `MM-handout.tar` distribution tests your `umalloc.c` package for correctness, space utilization, and throughput. The driver program is controlled by a set of *trace files* that are included in the `traces` subdirectory of the `MM-handout.tar` distribution. Each trace file contains a sequence of allocate and free operations of various sizes that instruct the driver to call your `umalloc` and `ufree` routines. The driver and the trace files are the same ones we will use when we grade your final submission. The `runner -h` command will display the various command-line arguments that you can present to the driver. These include several useful facilities, such as the ability to single-step through the trace.

4.4 The unit testing framework

We have created a simple-to-use unit testing framework for you.[1] To compile the unit test driver, type `make unittest`. This will make a binary called `unittest` that you can call from the command line. You will need to provide a path to the test case you want to run with the `-i` flag. See the example test case in the file `unittests/example.txt`. This file also contains instructions on how to create your own unit test. There is also a guide on how to individually test functions like `find`, `extend`, `split`, and `coalesce`.

Specify the `-s` flag if `block_size_alloc` stores the total block size; omit it if it stores the payload size. You can implement it either way, so we need to know this implementation detail here so you don't get wonky outputs.

Here is an example of running the unit test case framework.

```
$ unittest -i unittests/example.txt # payload size
$ unittest -is unittests/example.txt # total block size
```

[1] This is a new feature for this semester. In other words, it almost certainly has more bugs in it than the other well-trodden parts of the assignment.

4.5 Debugging with GDB

GDB is a great tool for debugging your code, finding segmentation faults, and stepping through your routines. The provided Makefile provides two targets to compile your code.

1. The `deploy` target uses the `-O2` flag, which tells the compiler to optimize your code. This has the side effect of optimizing out variables, making it harder to see what's going on in GDB. This is the version that you will use for performance testing, and this is the target that the autograder will build and run.

2. The `debug` target uses the `-O0` flag, which tells the compiler not to optimize your code. This makes it possible to debug using GDB. Use this version at the beginning to firm up your code.

Because the traces are so long, it can be useful to use **conditional breakpoints** to break at a specific line in the trace. For example, if your `umalloc` fails the correctness check after `umalloc(2048)`, you can use the command `(gdb) b umalloc if size == 2048` to set a conditional breakpoint for this case.

4.6 Programming rules

- You must not change any of the interfaces in `umalloc.c`.
- You may not implement a bump allocator. You may not implement the allocator described in §8.7 of K&R2e.
- You must not invoke any memory-management related library calls or system calls. This forbids the use of the standard `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 of derived data types such as arrays, structs, trees, or lists in your program. However, you *are* allowed to declare global or `static` variables of the basic data types and of pointers.
- For consistency with the system `malloc` package, which returns blocks aligned on 16-byte boundaries, your allocator must always return pointers that are aligned to 16-byte boundaries. The driver will enforce this requirement for you.

5 Handing in your assignment

Upload the files `umalloc.h`, `umalloc.c`, and `check_heap.c` on Gradescope. **These files must be at the top level of your git repository.** The autograder for this assignment will look for `./umalloc.h`, not for `./MM-lab/umalloc.h`.

6 Evaluation

This assignment is worth 6% of the total course grade.

- Checkpoint 1 will contribute 3%, and you will receive these points if:
    (a) you comment 1-2 sentences per function in `umalloc.h` explaining what it does;

   (b) your uploaded `check_heap.c` file implements a heap consistency checker following the instructions given in Section 4.1; and
   (c) your uploaded `umalloc.c` file passes the tests in the four files `short1.rep`, `short2.rep`, `short1-bal.rep`, and `short2-bal.rep`. Note that passing means correctness only.

- The final hand-in will contribute 3%. Your code must compile without error. The correctness of your solution will be determined by the supplied traces. You must also write 3-4 sentences describing your block selection strategy (above `find()`), how you split a free block (above `split()`), and your free block insertion policy (above `ufree()`).

  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:

   (a) *Correctness (20%).* You will receive full points for correctness if your solution passes the correctness tests performed by the driver program. You will receive partial credit for each correct trace. If you only receive partial credit for correctness, your utilization and throughput scores will be adjusted accordingly.
   (b) *Space utilization (50%).* The autograder will determine the average space utilization of your solution. An average utilization of 60% will earn full credit for space utilization. An average utilization over 60% can earn up to 5 extra points, for a maximum of 55 points from space utilization. **An average utilization under 35% will earn zero credit for utilization.**
   (c) *Throughput (20%).* The autograder will determine the average throughput of your solution. An average throughput of 1400 will earn full credit for throughput.[2] An average throughput over 1400 can earn up to 5 extra points, for a maximum of 25 points from throughput.
   (d) *Style (10%).*
      (i) Your code must be modularized into functions and use as few global variables as possible. You will need to use some global variables (such as a pointer to the head of your free list), but only add more if they are absolutely necessary to implement your solution.ction does.
      (ii) Each of the routines `find()`, `split()`, and `ufree()` must have a header comment that describes what it does and how it does it.
      (iii) Your heap consistency checker `check_heap()` must implement the required checks and be well-documented.
      (iv) Any changes to the provided data structures or helper functions must be documented.
      (v) Any new data structures or helper functions you introduce must be documented.
      (vi) Outside of the above guidelines, you just need to use a consistent style throughout your code. (For example, we don't enforce any specific variable name convention, but you should be consistent with whatever convention you choose.)

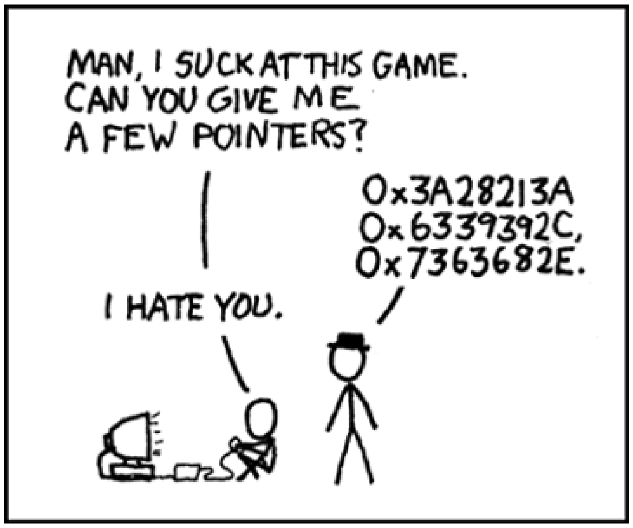
   Half of the style points will be for the heap consistency checker; the other half will be for good program structure and comments.

7 Hints

- Comment `umalloc.h` *first.* It will help familiarize you with the functions that we give to you and how they fit in with what you have to implement.

[2]The unit for throughput are unspecified. All that matters is that higher values are better.

- Don't wait to implement `check_heap.c`. It is an essential tool that will catch bugs as they arise. Don't forget to use the flag that runs the heap checker.
- *Use the command-line options to* `runner`. During initial development, using tiny trace files will simplify debugging and testing. We have included several such trace files (look for file names starting with `short`) that you can use for initial debugging.
- *Compile the* `debug` *target first and use a debugger.* A debugger will help you isolate and identify out-of-bounds memory references.
- *Understand and use the data structures and getters/setters provided to you.* You are, of course, free to modify the data structures and functions, or ignore them entirely. However, they have been provided to assist you in your design and implementation, and it is certainly possible to get full credit without modifying the given helper functions and data structures.
- *Do your implementation in stages.* Start by doing a detailed system design and making sure that all of the pieces will interact as intended. **Draw a picture of your heap.** Next, implement a heap consistency checker—at least a barebones version. Only then should you work on coding `umalloc` and `ufree` and making sure that they work correctly on the test traces. You can even defer coalescing until you have got the other pieces working. Performance and space optimization are always the final step; make sure you don't sacrifice correctness to gain efficiency.
- *Use a profiler.* You may find the `gprof` tool helpful for optimizing performance.
- *Start early!* It is possible to write an efficient memory allocator package in under 200 lines of code, if you do a good design up front. 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!



From `https://xkcd.com/138/`. Licensed under `https://xkcd.com/license.html`.