CS 3013 - Operating Systems

Project 3 (90 points)
Assigned: Friday, February 19, 2021
Checkpoint: Tuesday, March 2, 2021
Due: Friday, March 5, 2021

Project: Memory Allocation

In this project, you will get experience with user-space memory management by implementing the Goat memory allocator. From the perspective of the calling application, the Goat allocator behaves much like libc's `malloc` and implements the following functions:

1. `init(size_t size)`: This function is used to initialize your memory allocator.
2. `walloc(size_t size)`: This function is called by application code to request a new memory allocation of `size` bytes.
3. `wfree(void *ptr)`: This function is called by application code to free up existing memory chunks.
4. `destroy()`: This function will return the arena's memory to the OS and reset all internal state variables.

These functions are prototyped in the provided `goatmalloc.h` header file.

There three critical restrictions for your implementation. First, you cannot use the libc implementation of `malloc` for anything in your code. Second, the chunk list must be embedded into the memory arena (described below). Finally, you must not modify goatmalloc.h. Any projects failing to adhere to these restrictions are likely to receive a score of 0 points.

Initialization and Destruction

The following requirements are encoded in test cases `test_init_destroy()`.

The `init()` function must make a call to the `mmap` system call to request a region of memory from the OS. This region, called an *arena*, will serve as the source of all memory allocated by `walloc`. Specifically, `init()` should include code similar to the following.

```
int fd=open("/dev/zero",O_RDWR);
_arena_start = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
```

In the above example, `_arena_start` is a `void *` global variable pointing to the start of the arena.

The `init()` function must return the number of bytes mapped for the arena, if the mapping was successful. In the event of an error, the function will return one of the error codes defined in `goatmalloc.h`. For example, initialization function must return `ERR_BAD_ARGUMENTS` if the input size is negative.

The size of the arena is determined by the input parameter `size` and the page size. The `size` parameter denotes the minimum size of the arena. However, your allocator may have to increase the arena's size so that the arena is a multiple of the page size. For example, the call `init(1)` should result in an arena of 4096 bytes, where 4096 bytes is the page size on most modern machines.

Your allocator cannot hardcode the page size. Instead, it must use the libc function `getpagesize()` to determine the page size of the current machine.

The initialization function should also initialize the chunk list. See later sections for more details on the chunk list.

The `destory()` function should use the `munmap()` system call to return the arena's memory to the OS. The function must also reset any variables used to track internal state.

Calls to `init()` should produce output similar to the following and match the output seen in the provided `output_ref.txt` file.

```
Initializing arena:
...requested size 1 bytes
...pagesize is 4096 bytes
...adjusting size with page boundaries
...adjusted size is 4096 bytes
...mapping arena with mmap()
...arena starts at 0x7f8e5f4f9000
...arena ends at 0x7f8e5f4fa000
...initializing header for initial free chunk
...header size is 32 bytes
```

Calls to `destroy()` should produce output similar to the following and match the output seen in the provided `output_ref.txt` file.

```
Destroying Arena:
...unmapping arena with munmap()
```

Basic Allocation

The following requirements are encoded in test cases `test_allocation_basic()` and `test_free_basic()`.

The `walloc(size_t size)` function is called by application code to request a new memory allocation of `size` bytes. The allocated memory, called a *chunk*, must reside inside of the arena created by `init()`. The application code should be able to write to and read from the allocated chunk. Further, that chunk should remain available to the application until the application code explicitly frees the chunk.

Each chunk is composed of two components placed contiguously *inside* of the arena. The first component is a *header* containing metadata about the chunks and is of type `node_t`. The second component is the area of memory reserved for use by the application (i.e., the allocation itself). `node_t` is defined in `goatmalloc.h`. Because the chunk header is embedded into the arena, your allocator must account for the memory used by the header when allocating chunks.

The `wfree(void *ptr)` function is called by application code to free up existing memory chunks, i.e., make them available for future allocations. Here `ptr` is a pointer to the start of application-usable component of the chunk.

Your memory allocator must *embed* metadata into the arena to keep track of each allocated and free chunk. This metadata should take the form of a doubly-linked list of memory chunk headers, called the *chunk list*, and use the `node_t` defined in `goatmalloc.h`. Both free and allocated chunks will be placed in the same list. As mentioned above, the chunk header must reside immediately before each chunk. For example, if the chunk is at address (a), then the metadata for that chunk must be placed at address `((void *)a) - sizeof(node_t)`.

If the application code requests more memory than can be satisfied by the remaining free space in the arena, then `walloc()` should return null and set `statusno` to the error code `ERR_OUT_OF_MEMORY`.

Calls to `walloc()` should produce output similar to the following and match the output seen in the provided `output_ref.txt` file.

```
Allocating memory:
...looking for free chunk of >= 4064 bytes
```

```
...found free chunk of 4064 bytes with header at 0x7f9655b92000
...free chunk->fwd currently points to (nil)
...free chunk->bwd currently points to (nil)
...checking if splitting is required
...splitting not required
...updating chunk header at 0x7f9655b92000
...being careful with my pointer arthimetic and void pointer casting
...allocation starts at 0x7f9655b92020
```

Calls to `wfree()` should produce output similar to the following and match the output seen in the provided `output_ref.txt` file.

```
Freeing allocated memory:
...supplied pointer 0x7fd515872120:
...being careful with my pointer arthimetic and void pointer casting
...accessing chunk header at 0x7fd515872100
...chunk of size 3808
...checking if coalescing is needed
...coalescing not needed.
```

Free Chunk Splitting

The following requirements are encoded in test cases `test_allocation_withsplits()`.

When the requested allocation size is smaller than the size of the free chunk, the memory allocator must split the selected free chunk such that the remaining memory can be allocated in a future request.

There is one exception to the above splitting requirement: The allocator must only split the free chunk if there is enough available memory in the arena to account for metadata of the new chunk. Further, the resulting split free chunk must have a size of at least 1 byte.

Placement of Allocations

The following requirements are encoded in test cases `test_free_basic()` and `test_allocationfree_placement()`.

The chunk list must be logically ordered by the address of each chunk, in increasing order. This means that the first chunk in the list will always be the chunk with the lowest address and the last chunk will always be the chunk with the highest address. Note that this ordering should be naturally maintained when allocating and deallocating memory. In other words, you should not need to implement any special sorting functions.

Your allocator must assign the first free chunk from the chunk list that is large enough to satisfy the requested allocation. For example, the following sequence should result in `buff3` being placed at the previous location of `buff1`.

```
buff1 = walloc(64);
buff2 = walloc(64);
wfree(buff1);
buff3 = walloc(64);
```

Free Chunk Coalescing

The following requirements are encoded in test cases `test_coalescing_*`.

If freeing a chunk resulting in two or more adjacent free chunks in the chunk list, then your allocator should *coalesce* those chunks into a single larger free chunk. For example, the following sequence should result in the memory allocator coalescing all three chunks into a single free chunk when `wfree(buff2)` is called.

```
buff1 = walloc(64);
buff2 = walloc(64);
buff3 = walloc(64);

wfree(buff1);
wfree(buff3);
wfree(buff2);
```

Test Cases

We have provided a suite of test cases that define and exercise the critical functionality of your memory allocator. These test cases are all defined in `test_goatmalloc.c` and the result of running those test cases on a reference implementation is given in `output_ref.txt`. Your allocator must pass all provided test cases and produce output equivalent to the reference implementation to receive full credit for the assignment. Note that the grader may run additional test cases that will not be disclosed to the students, so it may be necessary to write test cases of your own.

To simplify development, we recommend that you use the test cases to guide your development. For example, first write your code to pass the initialization and destruction test cases (`test_init_destroy()`), and then focus on passing the basic allocation tests (`test_allocation_basic()`).

Finally, the behavior implied by these test cases supersedes any perceived contradictions or ambiguity in the written specifications. In other words, if anything is unclear in the specifications, please refer to the test cases for the intended behavior.

Hints

Some concepts that might be useful to review before starting this project:

- Read OSTEP Chapter 17 before starting this project. It introduces many of the concepts that you will need to understand to implement the Goat memory allocator. For example, the chapter talks at length about embedding chunk information.
- While OSTEP Chapter 17 is useful, the Goat allocator works a little differently than the schemes described in the book. Take note of the differences and think about how that will change your implementation.
- Void pointers (i.e., `void*`), structs, and pointer arithmetic. Many of your allocators interactions with the arena will be through the use of pointer arithmetic. Look to the provided test cases for some useful pointer examples.
- Read about `size_t` and its usage. Also remember that it is unsigned. This information will be useful for understanding and checking arguments.
- Always check the return value of system and library calls. Do not assume that they returned without error. Note that your Goat allocator must set and return error codes in a manner that is similar to libc.
- Always check that a pointer is not null before dereferencing.

Checkpoint Contributions

Students must submit work that demonstrates substantial progress towards completing the project on the checkpoint date. Substantial progress is judged at the discretion of the grader to allow students flexibility in prioritizing their efforts. However, as an example, any checkpoint assignment which passes the initialization, destruction, and basic allocation test cases will receive full credit towards the checkpoint. **Projects that fail to submit a checkpoint demonstrating significant progress will incur a 10% penalty during final project grading.**

Deliverables and Grading

When submitting the project, please include the following:

- The source code of your goat memory allocator in a file named `goatmalloc.c`.
- The unmodified header file (`goatmalloc.h`).
- A `Makefile` that compiles the executable.
- The output of running the provided test cases on your memory allocator in a file labeled `output.txt`.
- Optional: Any additional test cases you developed. Place these in a file called `mytests.c`.
- A document called `README.txt` explaining the project, any defects, and anything that the team feels the grader should know when grading the project. Only plaintext write-ups are accepted.

Please compress all the files together as a single .zip archive for submission. As with all projects, please only submit standard zip files for compression; **.rar, .7z, and other custom file formats will not be accepted**.

The project programming is only a portion of the project. Students should use the following checklist in turning in their projects to avoid forgetting any deliverables:

1. Sign up for a project partner or have one assigned (URL: `https://cerebro.cs.wpi.edu/cs3013-shue/request_teammate.php`),
2. Submit the project code and documentation via InstructAssist (URL: `https://cerebro.cs.wpi.edu/cs3013-shue/files.php`), and
3. Complete a Partner Evaluation (URL: `https://cerebro.cs.wpi.edu/cs3013-shue/evals.php`).

A grading rubric has been provided at the end of this specification to give a guide for how the project will be graded. No points can be earned for a task that has a prerequisite unless that prerequisite is working well enough to support the dependent task. Students will receive a scanned markup of this rubric as part of their project grading feedback. Students are expected to contribute equally on their teams; unequal contributions may result in different scores for the students on a team.

Project 3 – Memory Allocator – Grading Sheet/Rubric

Evaluation?

Grader: _____ Student Name: _____ _____

Date/Time: _____ Student Name: _____ _____

Team ID: _____ Student Name: _____ _____

Late?: _____

Checkpoint?: _____ Project Score: / 90

Earned	Points	Task ID	Description
_____	6	1	Passed all asserts in “test_init_destroy()”
_____	12	2	Passed all asserts in “test_allocaton_basic()”
_____	5	3	Passed all asserts in “test_free_basic()”
_____	25	4	Passed all asserts in “test_allocation_withsplits()”
_____	6	5	Passed all asserts in “test_allocationfree_placement()”
_____	10	6	Passed all asserts in “test_free_coalescing_case1()”
_____	7	7	Passed all asserts in “test_free_coalescing_case2()”
_____	7	8	Passed all asserts in “test_free_coalescing_case3()”
_____	6	9	Passed all asserts in “test_free_coalescing_chains_fwd()”
_____	6	10	Passed all asserts in “test_free_coalescing_chains_bwd()”

Each test case will be run independently, so it is possible to earn credit for test case 10 even if test case 6 did not complete successfully.

For each test case, you may earn credit for the first X assert statements passed. For example, if your code fails on the sixth assert statement in a test, you will earn credit for asserts 1-5, but you cannot earn credit for any of the remaining asserts in that test.

Notice: passing all tests are not sufficient to earn all the points. The correctness of the implementation is judged at the discretion of the grader. As an example, any assignment in which the policy is hardcoded to produce the expected output will lead to point deductions.

Grader Notes: