ASSIGNMENT 3

CS 361: Systems Programming / Spring 2023

Description

In this assignment, you will implement a set of utilities for heaps containing memory allocated dynamically with the function `malloc`.

API

You will have to implement the following API:

```
long distance_to_end_of_heap(void * ptr);

long chunk_size(void * ptr);

int is_chunk_free(void * ptr);

void * next_used_mem(void * ptr);

void * next_free_mem(void * ptr);

void free_everything(void * start, void * end, int size, long * stats);
```

distance_to_end_of_heap

Function malloc allocates memory inside a heap that grows as needed. It is possible to learn the size of the heap currently in use.

This function takes a pointer to memory allocated with malloc, and returns how far from the end of the heap that pointer is, in bytes.

chunk_size

Function malloc allocates memory in chunks. Each piece of memory it returns is part of a chunk. Each chunk has a header, and that header has the size of the chunk.

This function takes a pointer to memory allocated with malloc, and returns the size of the chunk allocated, in bytes. Note that the chunk may be larger than the memory requested with malloc. For instance:

```
void * mem = malloc(1);
printf("%d\n", chunk_size(mem)); // Prints 32
```

is_chunk_free

Chunk headers also contain information about the status of the chunk: whether they are free or used.

This function takes a pointer to memory allocated with malloc, and returns non-zero if the chunk that contains that memory is free. Otherwise, when the chunk is used (not free), this function returns zero. For instance:

```
void * mem = malloc(100);
printf("%d\n", is_chunk_free(mem)); // Prints zero
free(mem);
printf("%d\n", is_chunk_free(mem)); // Prints non-zero
```

next_used_mem

Function malloc allocates chunks sequentially in memory: each new chunk is allocated immediately after the previous.

This function takes a pointer to memory allocated with malloc, and returns a pointer to the next memory allocated with malloc in the same heap and not yet free'd. For instance:

```
void * mem1 = malloc(100);
void * mem2 = malloc(100);
void * mem3 = malloc(100);
if (mem2 == next_used_mem(mem1))
  printf("This line should be printed\n");
free(mem2);
if (mem2 == next_used_mem(mem1))
  printf("This line should not be printed\n");
if (mem3 == next_used_mem(mem1))
  printf("This line should be printed\n");
```

next_free_mem

This function takes a pointer to memory allocated with malloc, and returns a pointer to the next memory allocated with malloc in the same heap and free'd. For instance:

```
void * mem1 = malloc(100);
void * mem2 = malloc(100);
void * mem3 = malloc(100);
free(mem3);
if (mem3 == next_free_mem(mem1))
  printf("This line should be printed\n");
if (mem3 == next_free_mem(mem2))
  printf("This line should be printed\n");
free(mem2);
if (mem2 == next_free_mem(mem1))
  printf("This line should be printed\n");
```

free_everything

This function frees all memory allocated between (but not including) `start` and `end`. Only free memory chunks larger than the provided `size`. For instance:

```
void * start = malloc(100);
void * mem1  = malloc(100);
void * mem2  = malloc(9999);
void * mem3  = malloc(100);
void * end   = malloc(100);

free_everything(start, end, 500, NULL);

printf("%d\n", is_chunk_free(start)); // Should print zero
printf("%d\n", is_chunk_free(mem1));  // Should print zero
printf("%d\n", is_chunk_free(mem2));  // Should print non-zero
printf("%d\n", is_chunk_free(mem3));  // Should print zero
printf("%d\n", is_chunk_free(end));   // Should print zero

free_everything(start, end, 100, NULL);

printf("%d\n", is_chunk_free(start)); // Should print zero
printf("%d\n", is_chunk_free(mem1));  // Should print non-zero
printf("%d\n", is_chunk_free(mem2));  // Should print non-zero
printf("%d\n", is_chunk_free(mem3));  // Should print non-zero
printf("%d\n", is_chunk_free(end));   // Should print zero
```

The last argument is an optional pointer to an array of two longs. This function should populate the array with the number of free'd chunks in index 0, and the total size of free'd memory in index 1. For instance:

```
void * start = malloc(100);
void * mem1  = malloc(100);
void * mem2  = malloc(9999);
void * mem3  = malloc(100);
void * end   = malloc(100);

long stats[2];
free_everything(start, end, 500, stats);

printf("%d\n", stats[0]);  // Should print 1
printf("%d\n", stats[1]);  // Should a number slightly larger than 9999

free_everything(start, end, 100, stats);

printf("%d\n", stats[0]);  // Should print 2
printf("%d\n", stats[1]);  // Should a number slightly larger than 200
```

Entry Point

You will modify file `a3.c` with your solution and implement each function as defined above.

Due Date and Resubmission Policy

This assignment is due on **March 4th 2023** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

To resubmit this assignment, you are required to have attendance to Lab Sessions 5, and 6. You can resubmit your assignment until **March 11th 2023** (following Saturday) at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

Bonus Points

This assignment has a total of **20% bonus points**:

- 10% can be earned by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment3` label, and non-anonymous to the instructors to count towards the bonus. You can claim bonus points through **a Gradescope quiz**.
- 10% can be earned by completing Lab Sessions 5 and 6.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 100%. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 10 tests, that will check if your submission outputs the expected result. Each test is worth 10%.

Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your submission with `printf("expected result").`

The academic integrity policy described in the syllabus applies to this assignment. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.