ICS 53, Fall 2022

Assignment 4: A Memory Allocator

You will write a program which maintains an implicit heap. Your program will allow a user to allocate memory, free memory, and see the current state of the heap. Your program will accept user commands and execute them.

I. Assumptions about the heap

The heap is 127 bytes long and memory is byte-addressable. The first address of the heap is address 0, so the last address of the heap is address 126. When we refer to a "pointer" in this assignment we mean an address in memory. All pointers should therefore be values between 0 and 126.

The heap will be organized as an implicit free list. The heap is initially completely unallocated, so it should contain a single free block which is as big as the entire heap. Memory is initialized so that all addresses (other than the header and the footer of the initial free block) contain 0. Each block should have a header and a footer which is a single byte, and the header and footer byte should be contained in memory, just before the payload of the block(header) and after payload of a block (footer). **The most-significant 7 bits of the header and footer** ? ? ? **should indicate the size of the block, including the header and footer itself. The least significant bit of the header and footer should indicate the allocation of the block: 0 for free, 1 for allocated**. The header for the first block (the initial single free block) must be placed at address 0 in memory and the footer of the first block must be placed at address 126.

You cannot assume that the blocks are aligned. This means that the start address of a block can be any address in the heap.

II. Operations

Your program should provide a prompt to the user (">") and accept the following commands (Do not print any excessive message). Your program should repeatedly accept commands until the user enters "quit". You only need to support these commands and can safely assume they will be entered and formatted in the manner shown below.

1. **malloc <int size>** - This operation allows the user to allocate a block of memory from your heap. This operation should take one argument, the number of bytes which the user wants in the payload of the allocated block. This operation should **print out a pointer** which is the **first address of the payload** of the allocated block.

   Example:

   ```
   >malloc 10 // Comment: header at 0, payload from 1-10, footer at 11
   ```

```
1
>malloc 5 // Comment: header at 12, payload from 13-17, footer at 18
13
>malloc 2 // Comment: header at 19, payload from 20-21, footer at 22
20
```

2. **free <int index>**- This operation allows the user to free a block of memory. This operation takes one argument, the pointer to the start of the payload of a previously allocated block of memory. You can assume that the argument is a correct pointer to the payload of an allocated block.

   Example:

   ```
   >malloc 10
   1
   >malloc 5
   13
   >free 13
   >free 1
   ```

3. **blocklist** - This operation prints out information about all the blocks in your heap. The information about blocks should be printed in the order of payload size in a descending format like the example below. (if payload sizes are equal, the lower address first.) The following information should be printed about each block: the **payload size** , pointer to the **start of the payload**, and the **allocation status** **(allocated or free)**. All three items of information about a single block should be **printed on a single line and should be separated by hyphens.**

   Example:

   ```
   >malloc 10
   1
   >malloc 5
   13
   >blocklist
   106-20-free
   ```

```
10-1-allocated

5-13-allocated
```

4. **writemem <int index>, <char * str>** – This operation writes alpha-numeric characters into memory. The operation takes two arguments. The first argument is a pointer to the location in memory and the second argument is a sequence of alpha-numeric characters which will be written into memory, starting at the address indicated by the pointer. The first character will be written into the address indicated by the pointer, and each character thereafter will be written into the neighboring addresses sequentially. For example, the operation "`writemem 3 abc`" will write an 'a' into address 3, a 'b' into address '4', and a 'c' into address 5. Additionally, if a block is freed, you must ensure that whatever was written to the block is reset back to 0. **The index can be any location in the heap. Since C does not do explicit bounds-checking, the writmem command for your simulator can very well overwrite parts of another blocks even headers and footers and payloads of other allocated/free blocks. You are not required to do bounds checking for your heap as well. Operation like these will destroy the heap structure and are permitted operations. This can then corrupt subsequent operations as well.**

so even the first argument can be anywhere of the heap, which can overwrite anywhere???

5. **printmem <int index>, <int number_of_characters_to_print>** – This operation prints out a segment of memory in **decimal**. The operation takes two arguments. The first argument is a pointer to the first location in memory to print, and the second argument is an integer indicating how many addresses to print. The contents of all addresses will be printed on a single line and separated by a single hyphen. **The index along with number of characters can very well exceed block sizes. Like writemem, you are not required to do bounds checking.**

what to print for header and footer??? 0??? size of block??? the allcoated status??? 0 or 1???

Example:

```
>writemem 5 ABC

>printmem 5 3

65-66-67
```

so when we store in, we store 'A', 'B', 'C' in each address. Then print for ASCII???

Notice that the values 65, 66, and 67 are the decimal representations of the ASCII values of the characters 'A', 'B', and 'C'.

6. **quit** – This quits your program.

III. Requirements about allocation and freeing of memory.

When a block is requested which is smaller than any existing block in the heap, then **your code must perform splitting** to create a new block of the appropriate size. If splitting would create a new block whose size is smaller than 3 bytes, then splitting should not be performed. Instead, the entire block should be allocated to satisfy the request, even though the allocated block will be 1 or 2 bytes larger than is necessary to satisfy the request.

When a block is freed, it must be coalesced with the next block if the next block is free and the previous block if the previous block is free. (Forward and Backward Coalescing needs to be supported by your code)

When searching for a block to allocate, be sure to use one of the following strategies.

IV. Memory Allocation Strategies:

When we need to allocate memory, storing the blocklist allows us to implement various strategies.

- **First Fit**: This algorithm searches along the list, looking for the first block that is large enough to accommodate the payload. The block is then split into two blocks if the size of each is not smaller than 3 bytes; otherwise, splitting should not be performed. This method is fast as the first available block that is large enough to accommodate the payload is found. Remember that your blocklist was not in the order of block addresses, but in this algorithm, you need to find the first block in ascending order of block address.
- **Best Fit**: Best fit searches the entire list and uses the smallest block that is large enough to accommodate the payload. The idea is that it is better not to split up a larger block that might be needed later. Best fit is slower than first fit as it must search the entire list every time.

Your simulator should accept EITHER "FirstFit" or "BestFit" as the command-line argument OR no command-line argument, which by default selects the First Fit algorithm. The following examples show how your simulator should launch the system. We assume that the Linux prompt is the symbol "$ "and the executable is named "a.out".

$ ./a.out	The system, by default, adopts the First Fit algorithm.
$ ./a.out FirstFit	The system adopts the First Fit algorithm.
$ ./a.out BestFit	The system adopts the Best Fit algorithm

V. Submission Instructions

Your source code must be a single c file named 'hw4.c' containing your solution. Make sure not to print any excessive messages or text in your final code ; otherwise, it will make grading very difficult. Be sure that your program must compile on openlab.ics.uci.edu using gcc version 11.2.0.

Submissions will be done through Gradescope. If you are working with a partner, you must indicate that through Gradescope. The first line of your submitted file should be a comment which includes the name and ID number of you and your partner (if you are working with a partner).

VI. Sample Execution

```
>malloc 10
1
>malloc 5
13
>blocklist
106-20-free
10-1-allocated
5-13-allocated
>free 1
>blocklist
106-20-free
10-1-free
5-13-allocated
>malloc 5
1
>blocklist
106-20-free
5-1-allocated
```

```
5-13-allocated

3-8-free

>writemem 1 HELLO

>printmem 1 5

72-69-76-76-79

>free 13

>blocklist // Comment: You can see here that there was a block
allocated between 2 free blocks, on freeing that allocated block, both the
free blocks were coalesced into one free block.

118-8-free

5-1-allocated

>quit

$ // Comment: back to bash prompt.
```