CS 367 Spring 2017 – Lab 3
Heap Memory Management

Due: Tuesday, May 2, 11:59 PM

In class, we discussed the heap management algorithms . For this assignment, you are going to use C to implement a *simulation* of one of the algorithms that we discussed: **segregated lists**. The segregated list technique, as we discussed, is among the most efficient heap management algorithms in terms of search time and memory utilization.

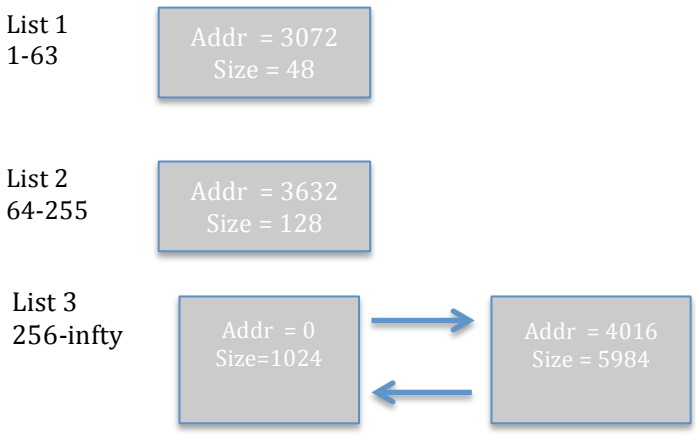
You can review the slides of *Run-Time Memory Management* lecture and the corresponding textbook chapters to get more details about the scheme to be implemented. Your simulator should be able to manipulate the heap properly after each `malloc(), free()` or `realloc()` request, and print out the resulting memory map when requested.

Using C (and the data structures described below), you will maintain and properly represent the current segregated lists after calls to `malloc()`, `free()`, and `realloc()`.

For example, let us assume that in a specific run we have three segregated lists. List 1 contains free blocks of size 1-63 bytes, List 2 contains those of size 64-255 bytes, and finally, List 3 contains all free blocks of size 256 bytes or larger. Assuming a heap size of 10000, initially, the entire heap is available in a single block of size 10000 in List 3. List 1 and List 2 are empty.

Now consider six back-to-back malloc calls, with requested sizes of 1024, 2048, 48, 512, 128 and 256 bytes, respectively. At that point, List 1 and List 2 are still empty, while List 3 has only one block of size 5984, starting at the address 4016.

Consider what the heap would look like after de-allocations of blocks of size 1024, 48, and 128. The three linked lists that illustrate this particular situation using segregated lists would be as follows:



You are going to manipulate the data structures that represent the segregated lists based on a sequence of calls to `malloc()`, `free()` and `realloc()`.You will be assuming an 8-byte alignment requirement.

Starting Code

The first thing you need to do is get the handout code from Blackboard, adjacent to this specification. Once you un-tar the handout (`tar xvf lab3_handout.tar`), you will have the following files:

- `memory.h`: This file, shown below, contains definitions and typedefs that you will use for this assignment. The **`HEAPSIZE`** gives the overall size of the heap. When a **`malloc`** (or **`realloc`**) request is made, you are going to align your block on 8-byte boundaries - the next two definitions are used for this aspect. **`ALIGN`** is a macro that will convert the given size to the next number divisible by 8.

```
#define HEAPSIZE 10000
#define ALIGNMENT 8
#define ALIGN(size) (((size)+(ALIGNMENT-1)) & ~0x7)


typedef struct m{
    int size;
    int address;
    struct m *next;
    struct m * previous;
 }mem_rec, *mem_ptr;
```

  The file also contains the prototypes for the **`mm_malloc, mm_free, mm_realloc`** functions. You can add additional definitions and data types to this data file and even add fields to the struct defined here. **Do not remove or rename anything or remove any fields**.

- `mm_malloc.c, mm_free.c, mm_realloc.c`: (stubs for your code) These files contains stub definitions for the three allocation functions you are writing. The three functions are:

```
mem_ptr mm_malloc(int size) {
   * Input: size of the block needed
     Output:
       Return a pointer to a mem_rec of the appropriate size (new_size).
       The function should identify the smallest nominal size segregated list that is
       potentially large enough to satisfy the new request. Within that segregated list,
       the block will be identified using the first-fit heuristic.

      If that segregated list does not have any free block that can accommodate the
      request, the segregated list of the next largest size should be checked, and so on.

     If none of the segregated lists has place for the block of the given size, call
     error_msg(1) and return a NULL pointer
  */


  int new_size = ALIGN(size);
  return NULL;
}
```

```
void mm_free(mem_ptr m) {
/* Input: pointer to a mem_rec
     Output:  None

        You must coalesce this block with adjacent blocks if appropriate. The new free block
        (if possible, coalesced with the neighbors) must be added to the appropriate
        segregated list. The appropriate segregated list is defined as the smallest nominal
        size list that is still large enough to hold the new free block.

        Before returning from mm_free, make sure that all free blocks, in all segregated
        lists, are address-ordered (lower addresses first).

        If the input pointer is null, call error_msg(2) and return.
  */
}


mem_ptr mm_realloc(mem_ptr m, int size) {
/* Input: pointer to a mem_rec, new size of the block needed
     Output:
       If the input pointer is null, call error_msg(2) and return

       Return a pointer to a mem_rec of the appropriate size (new_size).

       This block should be chosen as follows:
       if the new size is less than the current size of the block,
        use the current block after moving the excess back to the appropriate segregated
       free  list.

       if the new block size is larger than the current size,
       first see if there is enough space after the current block
       to expand (possibly using the neighboring free block).
        Any excess partial free block that may result should be put to the appropriate
        segregated list.

       If this will not work either, you will need to free the current block by calling
       mm_free(),  and find a location for the requested larger block using mm_alloc().


       If there is nowhere to place a block of the given size,
       call error_msg(1) and return a NULL pointer.

       Remember that the free blocks in each segregated list must be address-ordered
        at all times.

  */


   int new_size = ALIGN(size);
   return NULL;
}
```

You are going to create a static library using these functions and then linking this library into the executable. The Makefile will do this automatically for you but you might want to look at this file to see how it is done.

- **mdriver.c**: This file is the main program for your memory allocation system. It reads in traces and calls the appropriate functions. It also contains the initialization function and print function that will be used. You will need to add the implementation of the "get_seglist_info()" function that will

interact with the user to get the information about the number and sizes of the individual segregated lists. See below the paragraph titled "getting the segregated list configuration information" for more details.

The same file (mdriver.c) also defines a global variable "probes" initialized to zero. That variable will be used to count the number of free nodes visited during the mm_malloc() calls. You must update that variable properly in your mm_malloc.c file. After that, within mdriver.c, uncomment the line which reads "`printf("Total number of visited free nodes:%d\n", probes);`"

In the same file, there are two functions mm_init() and mm_dump(), that initializes the heap, and that prints the free blocks in the heap, respectively. The current implementation of those functions (in the provided mdriver.c) assumes a single free list, and the current "Heap" variable is a pointer to the head of that single free list. You will need to modify mm_init() and mm_dump() to initialize all segregated lists and to print the contents of the free blocks in all the segregated lists (in increasing address order).

Do not modify other existing parts of mdriver.c, otherwise your program may not compile or successfully run.

- **`trace1 – trace4`**: Sample traces that you can use to test your implementation

- **`trace1output – trace4output`**: The expected output of sample traces when run with only one segregated list (the traditional explicit free list). You can use these files for an initial test of your **`free, alloc, realloc`** implementations. Even though it is a good idea to develop and test your program first with only one list , keep in mind that during grading we will run your program also with multiple segregated lists of various sizes.

- **`Makefile`**: makes the library and the executable.

After using **`make`** to build the **`mdriver`** executable, it can be used from the command line with a single tracefile argument. Usage:

```
prompt$ ./mdriver <tracefile>
```

Getting the Segregated List Configuration Information

A key feature of the segregated list technique is that it is based on having *multiple* free lists. Each free list accommodates free blocks of of a distinct size range. Your program, at the beginning, must interact with the user (through stdin) to get the number and configuration of the segregated lists for that specific run.

If "n" is the number of the segregated lists, the program should read exactly 2*n + 1 integers from stdin. The first integer will give the number of the segregated lists, and the following 2n numbers will be the smallest and largest size in the range of List 1, List 2, etc.

For example, if there are only three lists, and
List 1 has free blocks of size 1 – 32;
List 2 has free blocks of size 33 – 128;
List 3 has free blocks of size 129 or larger

Then the user will enter: 3 1 32 33 128 129 0

The first integer will be the number of the segregated lists (3 in this example). All the numbers must be separated by whitespace and end with "0" – the range of the last segregated list must be large enough to accept a block of ANY size, if needed. "0" gives that information – in some sense it represents "infinity".

The function "get_seglist_info()" that you will implement in mdriver.c will implement that functionality. You can assume that the user will enter only non-negative integers; but you should include some basic error checking to make sure that there are exactly 2n integers after the segregated list number information ("n") and that those integers, except the last "0", are given in non-decreasing order.

Feel free to determine the return type and parameters of seg_list_info as appropriate. You will also probably need to define a (dynamic) array of pointers, where each element points to the head of a separate segregated list.

Implementation Notes

- **Linked Lists.** You will notice from the `memory.h` file and from the pictures that the each list is doubly linked. This is not a requirement of the assignment and the auxiliary functions in `mdriver` will work fine even if you do not use the `previous` link.

- **Segregated Lists.** All free blocks in a given segregated list must be ordered according to their addresses (lower addresses first). A free block must appear only in the segregated list of appropriate size range (implying that any excess size block that may arise while executing mm_malloc() should be moved to the segregated list of appropriate size range).

- **No arrays.** Except for possibly reading the input from the user, your programs should not use any statically defined arrays. If you need an array, dynamically allocate space for it (through malloc()) and use it that way. Your list manipulation must be based on the pointers exclusively.

- **The number of probes.** An important requirement is to accurately keep track of, and then print, the total number of visited free nodes in all mm_malloc() calls during the execution of your program. Use the global "probes" variable defined for that purpose in mdriver.c.

- **Creating a library:** The Makefile will automatically create a library with the three functions you are implementing. If you want or need to write auxiliary functions for these main functions you have two options.

    a. Add the function to the file for the function that uses it - this works well if only one of your functions needs this new function.
    b. Add a new file to the library itself. This is the recommended technique if more than one of your main functions use the same auxiliary function. To do this:
        i. Create a new file with the name of the function.
        ii. In the Makefile, add an entry to compile the function to an object file and add the new function to the 'ar' command (as well as the dependencies). You can use one of the basic functions (mm_free for example) as a pattern.

- **Trace Files.** If needed, you can modify the trace files or even create your own trace files easily. Each line in a trace file has three integer values: the operation, an index, and a size. Look at some traces and examine this chart.

Operation	Description
1: malloc	Each **`malloc`** has an associated **`index`** - you can think of this index as a pointer to the allocated node. To later free this node, use the same index. **`size`** is the space to be allocated. Remember that for alignment reasons, you may be allocating a larger block than requested.
2: free	The given **`index`** should correspond to an allocated block. The **`size`** field (which must be included) is ignored.
3: realloc	The given **`index`** should correspond to an allocated block. The **`size`** is the new size for the block and may be larger or smaller than the current size.
4: print out memory	Both the **`index`** and **`size`** fields are ignored (but required)
0: terminate program	Both **`index`** and **`size`** fields are ignored (but required)

Submission

- In this project, you are allowed to work with a partner enrolled in your own CS 367 Section. (Maximum group size is two). GMU and Computer Science Department Honor Codes will be enforced.
- Prepare a short (around 2-3 pages) project report that explains the overall design of your program, and the data structures you created for the representation of the segregated lists. Also include a brief "self-assessment" paragraph to include your own evaluation of the project: what are the strengths and limitations of your design? Are there any known problems in your implementation? What type of extensions can you think of? Make sure to include the names and G numbers of all team members in the report.
- Type 'make clean' and then zip or tar all the given files (make sure to include the versions that you modified, not the partially empty source files that are originally handed out!), and any new file that you may have created (including your project report in pdf).
- Upload this single compressed file to Blackboard. Before submitting your compressed file through the Blackboard, you should rename it to reflect your name(s), lab and section numbers. For example, if Mike Smith and Alice Parker from Section 001 worked together, and they are submitting a *tar* file, their submission file should be named as: lab3-s001-m-smith-a-parker.tar. Both members of the group must make separate submissions to the Blackboard, and their submissions must be identical. Make sure to plan in advance.
- In addition, submit the hard (paper) copies of all the source files, Makefile, and your project report at the beginning of the first class after the due date. A group can submit a single set of hard copy deliverables in class. Both soft and hard copies are needed for grading.

Grading

Submissions that are late will be penalized by 20% per day. Submissions that are late by 3 days or more will not be accepted. If you submit the soft copies after the class meeting following the due date, you are responsible for submitting also the paper copies of the source files to the instructor on the same date.

75 points of your grade will be determined by the correct implementation. 15 points will be for your overall design and report, and 10 points for coding style and commenting. **Make sure that your code runs on zeus as the grading will be done on zeus (no exceptions).** We will be testing each program with various trace files and with segregated lists of various configurations.

Recommendations

We recommend that you identify your partner and decide on the responsibilities of each team member as soon as possible. In addition, implementing first the case with single free list will enable you to understand and address the main intricacies of mm_malloc, mm_free, and mm_realloc functions. In that way, you will be able to also compare the output of your program against the trace outputs (which also assume a single list). If you decide to first implement and debug the single free list case, then you do not need to modify mdriver.c while implementing that – just focus on mm_malloc.c mm_free.c, and mm_realloc.c. Note that that initial implementation can implicitly assume that there is one free list that can hold any block of any size; so you do not need to write the get_seglist_info() function either.

Once you make sure that your program works reasonably well with the single free list case, you can extend your program to multiple segregated lists with arbitrary configurations. That will require further modifications in mm_malloc, mm_free, and mm_realloc, and implementing the changes required in mdriver.c. Make sure to leave sufficient time for that extension!