CS 367 – Project 3
Heap Memory Management

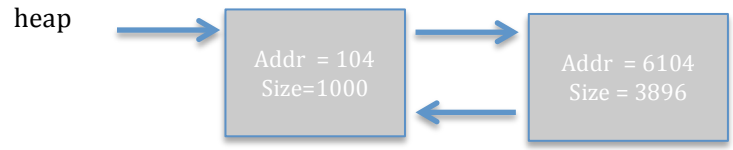
Due: Tuesday Nov 22, 11:59pm

This is an individual effort – no partners allowed.

Start by downloading p3handout.tar to your account on zeus.

In class, we discussed management of free/allocated information in the heap for the `malloc()`, `free()`, and `realloc()` functions. For this assignment, you are going to use C to implement a *simulation* of one of the algorithms that we discussed: **explicit lists**. (It will also use the **first-fit** approach). You will want to review the slides to get more details on the algorithm than provided in this document.
Using C (and the data structures described below), you will represent the current free list after calls to `malloc()`, `free()`, and `realloc()`. For example, initially, the entire heap is available in one block. You can represent this with the single-element linked list:

Consider what the heap would look like after some **allocations** (requested sizes of `100`, `1000` and `5000`) followed by a **de-allocation** (of the `Addr=104` block). A C linked list that illustrates this particular situation would be:



In both drawings, only the free nodes are in the explicit list – the 'missing' blocks are all allocated. (Question: why are the nodes not the exact same size as the allocation requests? Because we must align to multiples-of-8 addresses). You are going to manipulate this data structure based on a sequence of calls to `malloc()`, `free()` and `realloc()`.

- *Note that the header/footer information about address/size is part of these C-struct list nodes, and don't take up space in the heap directly; this is the main aspect of our assignment that doesn't quite match a direct implementation.*

Starting Code

Once you un-tar the handout (`tar xvf p3handout.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.
    - *NOTE: the size/alignment information shows up in the mem_rec struct, not as actual bytes of space in the heap.*

```
#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;
```

  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).
       This block should be found using first-fit.
       If there is nowhere to place a block of the given size, call error_msg(1)
          and return a NULL pointer
  */
  int new_size = ALIGN(size); // you'll likely use this somewhere here
  return NULL;
}



void mm_free(mem_ptr m) {
  /* Input: pointer to a mem_rec
     Output:  None
        You need to be sure the status of the given block is FREE.
        You must coalesce this block with adjacent blocks as appropriate.
        If the input pointer is null, call error_msg(2) and return
        If the given pointer points at a free node, call error_msg(3) 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
       If the given pointer points at a free node, call error_msg(3) 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 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.
            If this will not work, you will need to free the current block
              and find a location for this larger block using first-fit.
       If there is nowhere to place a block of the given size, print
          call error_msg(1) and return a NULL pointer
  */
   int new_size = ALIGN(size);
   return NULL;
}
```

You are going to be creating 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 from **mm.c**. It also contains the initialization function and print function that will be used.

- **trace1 – trace5:** The five progressively more advanced traces that you can use to test out the assignment.

- **mdriver_ref**: You can use this (zeus-compatible) executable to help you understand what your output **should** be for the given traces (or traces you create). Usage:

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

- **run_test**: Unix shell script that will automatically run your implementation against the driver and let you know where or not your answers are completely correct.

- **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>
```

Implementation Notes

- **Linked Lists.** You will notice from the `memory.h` file and from the pictures that the linked 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. It is set up this way because I like to use this structure.
- **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.
    - 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.
    - Add a new file to the library itself. This is what I would do if more than one of your main functions use the same auxiliary function. To do this:
        - Create a new file with the name of the function.
        - 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 (e.g. `mm_free`) as a pattern.

- **Trace Files.** As described earlier, you were given some trace files for testing. If needed, you can modify these 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` (name)- you can think of this index as similar to a pointer to the allocated node. To later free this node, use the same index. `size` is the space to be allocated. Remember: 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)

The trace files start out easy and get more difficult. For example, the first trace file only has `malloc` calls. Be sure you are getting the correct output for each trace before you move on to the next!

We will use different trace files for testin; these are just guides. Test your own program sufficiently!

Submitting via BlackBoard

- On zeus, type `make clean`. Zip or tar all of the given files (in case you changed anything). Upload.

Grading

Your grade will be determined as follows:

- **72 points** – equally-weighted among multiple traces. *These won't be the sample traces!*
    - *Your code will be run on zeus; we're diffing the results, so remove any debugging printf's.*
- **12 points** – runtime behavior (running on zeus with `valgrind`). You code should have no more leaks than the reference version.
- **16 points** – code & comments. Document your design clearly through commenting, etc.