Programming Assignment #5

There are two facets to this assignment. The first one gives you a chance to implement a rudimentary memory manager for dynamically allocated memory. The second one is that it will be implemented as a shared object library. The memory manager will make use of a singly-linked list to keep track of allocated and unallocated blocks of memory. The algorithm for memory management was discussed in class and is on the website as is the information regarding shared objects.

1 Interface to the Memory Manager

The interface for the memory manager is simple and consists of 4 functions, listed below:

```
typedef enum MMPolicy {mmpFirstFit, mmpBestFit} MMPolicy;

block_info *mm_create(size_t bytes, enum MMPolicy policy);
void *mm_allocate(block_info *heap, size_t bytes, char *label);
int mm_deallocate(block_info *heap, void *address);
void mm_destroy(block_info *heap);
```

Each node of the linked list is an instance of a `block_info` structure as follows:

```
#define LABEL_SIZE 16
typedef struct block_info
{
  int allocated;           /* flag if block is allocated (1) or not (0)        */
  size_t size;             /* size (in bytes) of the block                     */
  char *address;           /* address of the block                             */
  char label[LABEL_SIZE];  /* text label for the block (useful for debugging)  */
  struct block_info *next; /* pointer to next block                            */
}block_info;
```

1.1 Functions

`mm_create` — creates a memory manager instance of a specified heap size (in bytes). Your memory manager should use `malloc` *only once* to create the entire heap of memory that it will use for memory allocation. You will also need to use `malloc` to create new nodes for the list. The parameter `total_bytes` gives the total size of the heap in bytes. For example, the code fragment

```
block_info *heap = mm_create(10000, mmpFirstFit);
```

creates a block of memory, named `heap`, with a heap size of 10,000 bytes. Each subsequent call to the `mm_allocate` function will return a pointer to a chunk of memory within this 10K block. The second parameter is the search policy employed when searching for an available block. This was discussed in detail in class and is in the notes.

`mm_destroy` — destroys the memory manager by freeing the block of memory allocated in the `mm_create` function, as well as all of the nodes in the list that are tracking the blocks.

`mm_allocate` — get a block of memory of a specified size (in bytes) from the heap. On success, the function returns a pointer to a memory block of the requested size; on failure, the NULL pointer is returned. For example, the code fragment

```
char *block = mm_allocate(heap, 50, "fred");
```

will allocate space from the heap for an array of 50 chars and label the block *fred*. Be sure to use the C library function `strncpy` to copy the label into the block. This will ensure that you don't overwrite memory.

`mm_deallocate` — frees a block of memory (that was allocated using `mm_allocate`) from the heap. The value of `address` should be a pointer that was returned by the `mm_allocate` function. The code fragment

```
mm_deallocate(heap, address);
```

frees the space on the heap used by the array of characters allocated above. Upon successfully freeing the block, `SUCCESS` is returned. If `address` has a value of NULL, then nothing is freed and `SUCCESS` is returned. If `address` refers to an address that was not returned by `mm_allocate` (i.e. the block is not in the list), or refers to the address of a block that was previously deallocated, then nothing is freed and `FAILURE` is returned.

1.2 Details on `allocate` and `deallocate`

The `mm_allocate` member function should search for a block of memory on the heap of the requested size. You must implement both the *first fit* and *best fit* algorithm as was described in class.

Not only should the `mm_deallocate` function mark the specified block as free (deallocated), but should also consolidate adjacent free blocks into a single free block as was discussed in class. Refer to the web site and your notes for more information.

2 Implementing with shared objects

The second "feature" of this assignment is to implement the memory manager functions in a shared object (library). This means that the user won't link the functions at link-time, but instead at run-time. This was demonstrated in class and is in the notes on the website for reference. Note that you don't have to do anything special in your functions for them to be used in a shared library. The shared library is built by supplying command line options to the compiler (and linker).

Technically, the burden of using shared libraries is on the user (e.g. the driver) of the library. It is the responsibility of the user to load the library (using `dlopen`) and to retrieve the necessary function pointers (using `dlsym`). However, since one of the goals of this assignment is to have the student practice these tasks, you will be responsible for implementing the code to do this. The driver will simply call your code to load the library and retrieve the addresses of the functions.

There are multiple test drivers for you to use during your development. Most of the drivers work like all of the other drivers you've seen, meaning, they link to your code statically (at link-time). There is also a test driver that uses the shared library to link to your code dynamically (at run-time). Use the statically-linked driver for development and testing. Once you have everything related to the memory management working, move on to the dynamically-linked test driver and make sure you've got that working as well. Both drivers produce the exact same output. Keep in mind that the static version is to help you get the memory management working. I will not be using it to grade, so if you can't get the dynamic version to work, you won't get a good grade. As always, see the website for more details.

3 What to hand in

Your submission should consist of four files: the Doxygenized implementation files `memmgr.c` and `shared.c`, the `typescript` file, and the PDF file `refman.pdf`.