COMPUTER SYSTEMS
PROJECT 4
A space re-using allocator

1 A more complete allocator

Moving forward from Project 3, we will now implement a *recency-ordered, first-fit, non-splitting, non-coalescing allocator* that re-uses heap blocks that have been freed.

More specifically, you are going to implement a heap allocator that:

- Maintains a list of blocks that were previously allocated and then freed.
- Includes a *header* with each block that stores the block's size, and that has pointer space to link the block into a linked list.
- Preferrentially uses a block from the free list when possible, creating new blocks only when necessary.

2 The code

2.1 Getting it

Get started by creating yourself a directory for this project and grabbing new source code:

```
$ wget -nv -i https://bit.ly/AMHCS-2019F-171-p4
```

You will end up with a new source code file, `neo-alloc.c`, that is quite similar to your previous `pb-alloc.c`, but is changed in a number of critical ways.

2.2 Grokking the changes

- **The `header_s` structure:** We will use *C structures* (or *structs*, for short) to represent our headers. A *struct* is something like a Java object that has only instance data members—no class variables, no methods. If we take a *struct* pointer to a memory location, we will then treat that location as having space for each of its *fields*.

  You will see that the `header_s` struct contains two fields: `size`, to hold the size of the block; and, `next` to link to the next free block in a linked list of them. If we have a pointer to a header (e.g., `header_s* header_ptr`), we can then access the size of the block by writing `header_ptr->size`, and access its linked-list pointer by writing `header_ptr->next`.

- **The `WORD_SIZE` and `DOUBLE_WORD_SIZE` symbols:** These are constants that can be used in checking *word-* and (particularly) *double-word* alignment of the addresses returned by `malloc()`.

- **The head of the free block list:** The static variable `free_list_head` is declared, and is intended to be the beginning of a linked list of free blocks.
- **The `debug()` and `debug_int()` functions:** To make it easier to emit debugging messages in your code, you can call these functions to print a fixed string or a given integer value to the terminal. Note that these functions avoid calling `malloc()` itself, making them safe to use on this project (unlike `printf()`, which does call `malloc()`).
- `main()` **itself:** It now does more allocating and deallocating, and printing of the addresses at which those allocations occur. This expanded `main()` is not a comprehensive testing of an allocator, but it does serve as an example of ways in which you may more thoroughly test your own solution.

3 Your assignment

3.1 Required elements

In order to complete this assignment, you must find the incomplete portions of the following methods (marked with a comment that begins, `TO DO`), and complete them:

1. `malloc()`: There are two portions to be completed here. The first requires you to add code that traverses the linked list of free blocks (using `free_list_head` as the entry-point to the list), searching for one that is sufficiently large for the request. If it finds such a block, it should be removed from the free list, and then returned.

   If no such block is found, then the code that follows in this function will pad the requested size up to the nearest multiple of 16, insuring double-word alignment, and then check that the allocation will not exhaust the available space for the heap. After this point, you will find the next segment that you must complete, in which a new block is allocated at the end of the used portion of the heap (thus moving `free_ptr`). Your work from the previous project may be useful, although you should note the need to add and initialize a header for the new block before returning it.

2. `free()`: Add code to find the header of the block (which should immediately precede the given `ptr`), and then insert that block to the front of the free list.

3. `realloc()`: This function, which can re-size an allocated block, must determine how large the existing block is, thus allowing it to compare the requested new size to the existing size. Thus, you must complete the code the finds the header on the block, and grabs it `size` field for the rest of the function's code to use.

When you complete these portions in a self-consistent way, the allocation should preferrentially use a sufficiently large free block for each allocation request, creating new blocks only when necessary.

3.2 Optional challenges

You may wish to enhance the capabilities of this allocator—something you are welcome to do to more deeply develop your understanding of how this abstraction can be fully implemented. **These are optional, and need not be completed to receive full credit for the assignment.** You have a number of choices for ways to improve your allocator:

- **Implement a *best-fit* search:** Don't just search for the first block in the free list that is has a sufficient capacity; search the entire list and use the best fit (if any).
- **Spilt re-allocated blocks:** When you find a free block, if it is larger than the request, split that block into two. Doing so will require adding a header to the remaining portion of the free block, and linking it back into the list as a new (and smaller) free block.
- **Coalesce free blocks:** Examine the previous and next blocks in the address space; if either or both are also free, coalesce these into a single, larger free block. Note that you may wish to add fields to the header that will help your code find the previous and next blocks efficiently.
- **Change the structure to be a segregated-fits allocator:** This is fundamentally different way of creating and organizing blocks. We will discuss these in class next week, and you may wish to try to implement this efficient allocator.

4 How to submit your work

Submit your `neo-alloc.c` file using one of the two usual tools:

- **Web-based:** Visit the submission system web page.
- **Command-line based:** Use the `cssubmit` command at the shell prompt on `remus/romulus`.

This assignment is due on Thursday, Oct-10, 11:59 pm.