CSE 109: Systems Programming

Fall 2017

Program 4: **Due on Wednesday, October 11th at 9pm on CourseSite.**

Collaboration Reminder:

1. You must submit your own work.
2. In particular, you may not:
   (a) Show your code to any of your classmates
   (b) Look at or copy anyone else's code
   (c) Copy material found on the internet
   (d) Work together on an assignment

Assignment: Preparation

1. Make a *Prog4* directory in your class folder.
2. Create the files *Alloc.c* and *Alloc.h*
3. The linkable object, *MemTester.o*, will be provided at some point at ~jloew/CSE109/prog4student/

   It will not necessarily be obvious what is wrong when/if it fails.
4. All source code files must have the comment block as shown at the end of this document. All files must be contained in your *Prog4* directory.

Assignment:

You will be creating both a header and an implementation file for this assignment.

You will be creating a basic memory allocator, non-optimized, that can fulfill memory requests made by users.

1. *Alloc.h* and *Alloc.c*. Make sure you put the appropriate material into each file.
   (a) Define the *Alloc* structure.

    i. It must contain a *void ** called *memory* that points to the chunk of memory that the *Alloc* structure is using.

    ii. It must also contain a size_t called *capacity* that indicates the maximum capacity of the structure. This amount may be slightly more than the user actually requested (see construction).

    iii. You can (and will) include additional instance variables in order to satisfy the assignment.

(b) struct Alloc* constructAlloc(size_t size): Creates an *Alloc* object with the desired size and returns it to the user. The user is expected to call *destructAlloc* when they are done with it. If the *size* provided is invalid, return a *NULL* pointer instead - *size* is invalid if it less than 0. *size* is also invalid if it causes our attempt at allocation to fail. *size* must be rounded up to the nearest *double word* size (multiple of 16).

    Do not use *calloc*. The tester may disable *calloc*.

(c) struct Alloc* destroyAlloc(struct Alloc*): Destroys the Alloc object that is given to it. This is where you do all the appropriate *free*s that are required to not leak memory. Returns a *NULL* pointer when successful, will crash otherwise (you don't have to make it crash, it will crash on its own).

(d) void *allocate(struct Alloc*, size_t size): Requests *size* number of bytes from the *Alloc* object. The *Alloc* object returns a pointer of *size* bytes (where *size* is rounded up to the nearest *double word* size). The *Alloc* object must not reuse this space until the user returns it back to the *Alloc* object. If we can not provide sufficient space, return *NULL* instead. You may not use *malloc* or related calls to create the space returned, you must space within the *memory* pointer as what you return to the caller.

    You may end up using *malloc* in regards to how you keep track of what is used and what isn't. You must use a size/capacity idiom to minimize how many times you call *malloc*.

(e) void deallocate(struct Alloc*, void *ptr): This returns data back to the *Alloc* object, such data can now be reused for allocations. If *ptr* is *NULL*, do nothing. If *ptr* is not something that we provided to the user print ”corruption in free” to *stderr* and call *exit(2);* to terminate the program.

(f) size_t getCapacity(struct Alloc*): Returns the maximum capacity, in bytes, of the *Alloc* object.

(g) size_t getSize(struct Alloc*): Returns the amount of data currently allocated by the *Alloc* object, in bytes.

(h) void* getBase(struct Alloc*): Returns a *void ** that is the base of the *Alloc* object's data, namely *memory*.

(i) size_t getNumAllocations(struct Alloc*): Returns the number of allocations currently held by the *Alloc* object.

(j) void** getAllocations(struct Alloc*): Returns an list of *void **s that represent all of the locations that the *Alloc* object has given to users which has not yet been returned.

You may be able to simply return a *void *** cast of whatever instance variable you are using to keep of allocations.

(k) int needDeallocate(struct Alloc*): Returns 1 if the user needs to deallocate the pointer returned by *getAllocations*. Returns 0 if the user should not try to deallocate that pointer.

(l) void *riskyAlloc(struct Alloc*, size_t size): Same as *allocate* except in the case that we don't have enough memory available, use *realloc* to get more memory. This is completely unsafe in some cases. If the reallocation was *safe*, you now have a larger capacity and need to adjust accordingly - you can then allocate as normal. If the reallocation was not safe, meaning that the pointers given to the user are now all invalid, print "Bad realloc" to *stderr* and then return NULL.

Testing:

1. You will need to use multiple steps to compile your code since you will have to link the compiled Alloc code with some other compiled code that will use Alloc.

   You can provide a *Makefile* if you want, it will not be used during our testing.

2. Make sure to test cases where you run out of memory - note that you may memory leak in that case without penalty, but only in that case.

Submission:

1. Once ready to submit, you can package up the assignment as a .tgz file

   tar -czvf Prog4.tgz Prog4

   You must use this command in the directory that contains the *Prog4* folder, not within the directory.

2. Transfer *Prog4.tgz* to the Program 4 submission area of CourseSite.

Comment Block:

```
/*
    CSE 109: Fall 2017
    <Your Name>
    <Your user id (Email ID)>
    <Program Description>
    Program #4
*/
```