CSE 109: Systems Programming

Fall 2018

Program 4: **Due on Sunday, October 21st at 9pm on CourseSite.**

Checkpoint Due: **Due on Monday, October 15th at 9pm via checkpointer.**

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 *Allocator.c*, *Allocator.h*, *Allocation.c* and *Allocation.h*
3. You can use whatever tester you want for this. We will provide a *prog4.o* that you can link to your code for testing purposes.
4. We also will provide an object and header file that you must link to your code to provide the Allocator::printAllocations(...) function (see *printAllocations.h* and *printAllocations.o*).
5. 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 a basic memory allocator, **non-optimized**, that can fulfill memory requests made by users.

1. *Allocation.h* and *Allocation.c*. Make sure you put the appropriate material into each file.

    (a) Each *Allocation_t* object will contain a size_t to represent the starting offset of the allocation as well as the size of the allocation.

    (b) void makeAllocation(struct Allocation_t* it, size_t start, size_t size): Constructs the allocation object with the given values. Note that the *Allocation_t* objects are **not aware** of the constraints of the allocator and must not care either. We won't return anything - just modify that space you are pointed to. Nothing should stop the user from making an invalid allocation, they must use *doesOverlap* to ensure this does not happen.

    (c) void freeAllocation(struct Allocation_t* it): Destroys the contents of the *Allocation_t* object. User is responsible for assigning to NULL afterwards, therefore, we return nothing and do not free the actual *Allocation_t* object.

    (d) size_t getStart(struct Allocation_t* it): Returns the starting location of the allocation.

    (e) size_t getEnd(struct Allocation_t* it): Returns the ending location of the allocation (not inclusive).

    (f) size_t getSize(struct Allocation_t* it): Returns the size of the allocation.

    (g) int doesOverlap(struct Allocation_t* it, size_t start, size_t size): Returns 1 if the*Allocation_t* object would overlap with the given range. Returns 0 otherwise. Use this before creating a new *Allocation_t* object to validate that *start* and *size* do not conflict with any existing *Allocation_t* object.

2. *Allocator.h* and *Allocator.c*. Make sure you put the appropriate material into each file.

    (a) Define the *Allocator_t* structure.

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

        ii. A size_t to represent the capacity of the allocator.
        This must be a multiple of 16.
        This reflects alignment for *long double* on a 64-bit Intel machine.

iii. A dynamically resizable list of *Allocation_t* objects as well as size_t fields to represent size and capacity of this list.

   When expanding, expand by doubling capacity and then adding 1.

iv. The amount of memory used by the *Allocator_t* object to track allocations must scale linearly with the number of active allocations, not the amount of total memory.

(b) void makeAllocator(struct Allocator_t* it, size_t capacity): Constructs an *Allocator_t* object using the given capacity. Since the capacity must be a multiple of 16, round up to the nearest multiple of 16, if necessary. This will call malloc once, to create the memory space specified by the capacity. The list of *Allocation_t* objects will initially be empty and there shall be no mallocs associated with it at this point. Make sure that you only allocate memory once at this point: this may be explicitly tested. You will also keep track of the sum of the allocations made in a size_t.

(c) void freeAllocator(struct Allocator_t* it): Destroys the given *Allocator_t* object. User is responsible for freeing and assigning to NULL afterwards, therefore, we return nothing. This must handle freeing *memory* as well as any *Allocation_t* objects we may have and the list that contained them.

(d) void* allocate(struct Allocator_t* it, size_t amt): Requests an allocation from the allocator. The allocator will determine a region of space within *memory* that satisfies the request but has not been allocated already. Note that *amt* must be rounded up to the nearest multiple of 16. Track the allocation of this space and then return a pointer to the region of space within *memory* that satisfies this request. If no such space exists, return NULL. The memory location you return **must** be that with the **lowest possible address** within your allocation - if you always scan from the beginning of memory to find where to allocate, you will do this implicitly.

   You are **required** to use *Allocation_t* objects to track the usage of memory within the allocator.

(e) void deallocate(struct Allocator_t* it, void* ptr): Deallocates the given allocation. If we are given NULL, ignore the request. Otherwise, search through our list of allocations and remove the matching allocation. If there is no matching allocation, print an error to standard error (a bunch of garbage with numbers would be appropriate and funny but not necessary, "Corruption in free" is sufficient.) and call exit(1) to terminate the program. When the user tries to deallocate something that we didn't allocate for them, we punish them severely.

(f) void* getBase(struct Allocator_t* it): Returns a pointer to the allocator's memory. You won't use this for anything in particular but the tester expects it.

(g) size_t getUsed(struct Allocator_t* it): Returns the amount of space used by the allocations in the allocator.

(h) size_t getCapacity(struct Allocator_t* it): Returns the capacity of the allocator.

(i) void printAllocations(struct Allocator_t* it, FILE* fd): This is given to you, precompiled. This prints all of the allocations made by the allocator in a "nice" format to the FILE* specified by *fd*. This requires the following two functions (getAllocation, numAllocations) which would be *private* if that was allowable in C.

(j) struct Allocation_t* getAllocation(struct Allocator_t* it, size_t index): Returns the allocation specified by index. Returns a NULL if out of bounds.

(k) size_t numAllocations(struct Allocator_t* it): Returns the number of allocations that are currently tracked by the allocator.

(l) void* riskyAlloc(struct Allocator_t*, 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. If the reallocation was not safe, meaning that the pointers that had been given to the user in the past are now all invalid, print "Bad realloc" to standard error and then return NULL.

Checkpointing:

1. The Allocation object, in its entirety (*Allocation.h* and *Allocation.c*) are due for the checkpoint.

2. You may call other functions, etc, as long as everything is included in your checkpoint submission.

3. To submit your checkpoint:

```
cp Allocation.h checkpoint4.c
cat Allocation.c >> checkpoint4.c
~jloew/CSE109/submitCheckpoint.pl 4
```

That is lowercase PL, followed by the number 4.

4. You may submit your Checkpoint up to ten times total, this includes after the checkpoint is due as well.

5. Ideally, it will tell you which functions are incorrect. It is possible that the functions are incorrect but pass the checkpoint. Although, they should be mostly correct or completely correct if they pass the checkpoint.

6. The checkpoint will not check for memory corruption or leaks. You will need to handle that yourself.

Style:

For assignments, we follow the Allman style of braces and indentation.

1. **Review the Style document on Coursesite**

Testing:

1. You will need to use multiple steps to compile your code since you will have more than one .c source file.

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

   module load gcc-7.1.0

   gcc -Werror -Wall -g -c Allocation.c

   gcc -Werror -Wall -g -c Allocator.c

   gcc -Werror -Wall -g -o prog4 Allocation.o Allocator.o prog4.o

2. Your final executable will be called *prog4*.

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

4. For your own testing you will need to link printAllocations:

    module load gcc-7.1.0

    gcc -Werror -Wall -g -c Allocation.c

    gcc -Werror -Wall -g -c Allocator.c

    gcc -Werror -Wall -g -c prog4.c

    gcc -Werror -Wall -g -o prog4 Allocation.o Allocator.o printAllocations.o prog4.o

Submission:

1. Your code **must** have the functionality as specified by the assignment and you absolutely must **not** break encapsulation unless it is otherwise not possible to do so. This is because we can replace your *Allocator* and/or *Allocation* code with our own and everything should still work.

2. 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.

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

Comment Block:

```
/*
    CSE 109: Fall 2018
    <Your Name>
    <Your user id (Email ID)>
    <Program Description>
    Program #4
*/
```