CMSC 313 — Spring 2017

Project 6 — Hybrid Memory Management: malloc() with Caching

Assigned	Thursday, April 13th
Program Due	Wednesday, April 19th by 11:59pm
Updates:	None yet.

Objectives

The objective of this programming project is to practice using pointers and dynamic memory allocation in C.

Background

In the previous project, we implemented our own form of memory allocation by creating our collection of allocatable blocks (`fraction`'s) as a static array, and linking them into a free list, handing them out one by one when requested by the caller, and collecting them back into the free list when the user deleted them. We did not utilize any operating system support for dynamic memory allocation.

For this project, we will be using real dynamic memory allocation, via calls to `malloc()`. However, as explained as part of the real-world motivation for Project 5, it is often the case that even when applications use `malloc()` and `free()`, they sometimes implement a form of intermediate caching to speed up memory management. We do this because calls to malloc() and free() can be expensive. It is cheaper to reduce calls to malloc() by pre-allocating several objects at a time, doling them out one at a time to the caller in subsequent calls, and reduce calls to free() by just adding deleted objects to a list of free objects. This is particularly true when the units of allocation are uniform, which is the case with our `fraction` structures.

(From this point on in this document, when we use the term "block", we will be referring to "space large enough to hold one fraction", i.e., a struct fraction-sized block of space. This is because the design you implement will no longer be dependent on fractions, but will in fact work with managing any kind of collection of uniform-sized blocks.)

For this project, you will again implement a system that manages a heap of `fraction`-sized blocks, but this time, instead of allocating them out of a static array, you will use `malloc()` to get the space for any and all necessary blocks. However, you will not be simply calling malloc() every time the user wants a new fraction. You wil implement a hybrid memory management scheme to make freeing and subsequently reallocating blocks much more efficient. It will do so by allocating space with malloc() in large chunks, big enough for several free blocks at once, which it will then manage and hand out one block at a time. When it has dispensed all the available blocks, if yet another block is requested by the caller,

it will call malloc() again, once again for a chunk big enough for several blocks. So, depending on how many blocks the user ends up allocating, you might have called malloc() multiple times, getting a small array of blocks each time.

Conversely, when then user frees up fractions, your package will not actually return them to the system via calls to `free()`, but will instead just link each fraction-sized block into a list of free blocks to keep around so that they can be very efficiently reallocated to the user when new allocation requests come in.

At this point, it might seem like the allocation function (new_frac()) is getting complicated: should it dispense blocks that it got from a call to malloc(), or should it dispense a block from the free list that the deallocation function (del_frac()) is buildin up? The solution is simple: if the free list is empty, then call malloc(), and take all the blocks in the new array and add them to the free list. Then, just dispense the first item from the free list.

This new scheme allows us to create as many new fractions as the user desires, up to the limits of the program's memory, which is a Good Thing. However, it also adds a complication: our collection of blocks--some allocated to the caller, the rest in our free list--will now be potentially in any of several arrays, since each call to malloc() returns a distinct array. We can no longer use a simple array index to identify any given block, so we can no longer link the free fractions together as we did in the previous project: using the "denominator" field to hold the index of the next free structure. We want to upgrade to treating the fractions as just blocks of space, anyway, which means linking the free blocks together using actual pointers. This further destroys any hope of using any of the fraction fields (like denominator) to hold our link information, because neither ints nor chars can be used to hold true pointers. We will solve this problem by using a C language feature called a `union`.

C `union`s

You have already used C `struct`s, which allow you to compose a set of members into a single data structure. There is another C structure called a `union`. You can tell C that a structure is actually serving multiple, mutually exclusive purposes at the same time, by defining a union. The syntax for a union looks like this:

```
union foo {
    int    i;
    double d;
    char   c;
};

union foo my_foo;
```

This looks very similar to a `struct` definition, but with the word "struct" replaced by "union". The above example first defines a new type of `union` called "foo", then declares a variable of that union type called "my_foo". The syntax for accessing the members is also the same: you would access the integer member of `my_foo` by using "my_foo.i". However, instead of the members("i", "d", and "c" in the above example) being laid out one after the other in sequential memory locations, ***all of the members occupy the same space*** (or at least, start at the same place), and the union takes up only as much space as its largest member. This means only one of the members can be in use at any time, and you have to remember which, but it will usually be clear from context. More later on using `union`s in this project.

Assignment

Just as in Project 5, you will be managing space for `fraction` structures. This structure remains exactly the same; this is our "block"--the unit of space we will be managing. However, instead of defining a large, fixed-size global array to disburse blocks from, you will be using `malloc()` to dynamically allocate chunks of 5 free blocks at a time. In other words, each call to `malloc()` will request space for an array of 5 blocks. You will only call malloc() when a request for a new

fraction (via a call to new_frac()) finds there are no free blocks available. Since you're malloc()'ing a bunch of blocks each time, you will set one aside to return to the user, and place the rest into a linked list of free blocks to be used for future calls to `new_frac()`.

As before, in the file frac_heap.c, you must supply four functions: `init_heap()`, `new_frac()`, `del_frac()` and `dump_heap()`. You will be writing completely new implementations for these functions, with the following modified behaviors:

- `init_heap()` must be called once by the program using your functions before calls to any other functions are made. This allows you to set up any housekeeping needed for your memory allocator. Note that it will no longer actually link together an initial list of free blocks. ***It should not call malloc(), either***. It will just initialize an empty free list. (Yes, this function is now as trivial as it sounds.)

- `new_frac()` must return a pointer to `fraction`. If the free list has any available free blocks, it removes the first one, and returns a pointer to it.
  If the free list is empty, it should allocate an array of 5 new free blocks using a single call to malloc(). (Note: the free list will be empty the very first time this function is called.) It should return one of these to the caller, saving the rest on the free list.
  To help the graders, each time malloc() is called, you must print out a debugging message saying: "called malloc(%d): returned %p\n", where the %d and %p print out the size passed to, and pointer returned from, malloc().

- `del_frac()` takes a pointer to `fraction` and adds that item to the free block list. This is identical to what it did in Project 5. Note that you never give space back to the system using free().

- `dump_heap()` is for debugging/diagnostic purposes. It changes significantly: it now only prints out the addresses of each block on the free list. If the free list is empty, it should print out "Free list is empty".

The sample main program (main6.c – almost identical to main5.c) can be used to do a simple test of your project. Additionally, you must write additional main programs that test various features of your memory allocator. Note that the different implementation will mean that the edge cases are different, so you need to test different things.

Implementation Notes

- The `fraction` struct has not changed at all from Project 5. Also, while the internal implementation of the fraction heap management functions has changed, the syntax and semantics of the function calls have not. Therefore, the file `frac_heap.h` has not changed at all. We are providing you with the file `frac_heap.h` for this project here.. However, obviously, frac_heap.c will be a completely new implementation, written entirely by you.

- Your memory allocator should no longer use a "large" global array of `fractions`. However, you will still need the head of the free list (now a pointer) to be a global variable, again declared `static` to give it file scope to ensure

  that code in other files does not access it directly. It should be declared to be a pointer to the correct type (see discussion below on `unions`.

- The internal implementation of the free list must now use pointers to link the free blocks together. This is now necessary, because you might have allocated more than one array over the life of the program (you only allocate arrays of 5 blocks at a time), which means you cannot use a simple index number. In Project 5, we dissuaded you from using pointers, because trying to store a pointer in one of the integer fields of a fraction would trigger warnings

from the compiler. However, there is a better way, using a `union`.
Once you have a definition for a fraction (and assuming you used typedef to define it as "fraction" as required in Project 5), you would do something like the following:

```
union fraction_block_u {
    union fraction_block_u *next;
    fraction frac;
};

typedef union fraction_block_u fraction_block;
```

Here, the frac_block would very likely be exactly the same size as a fraction, since the "frac" member, would be the larger of the two members of the union. When you access the "frac" member, the union will act exactly like a `fraction`.

In new_frac(), you would actually treat the chain of free blocks as a chain of fraction_block unions (so obviously your free list head would be a "fraction_block *"). Assuming you removed the first element from the free list and were pointing to it with "fbp", you would not even need a cast: you could just say:

```
return (&fbp->frac);
```

This would work because the address of any of the members of a union would be the same as the address of the union itself, but with a different associated type (in this case, "pointer to fraction" instead of "pointer to fraction_block").

Later, when the user frees up the fraction, you would cast the "fraction *" parameter to a "fraction_block *", using something like:

```
del_frac(fraction *fp) {
    fraction_block *fbp;

    fbp = (fraction_block *) fp;
    /* Now, you can access fbp->next */
```

and... Voila!... you have your fraction_block back!
Note that unlike in the previous project, you should ***not*** reset the numerator and denominator to 0--why not?

- You must use ONLY C! The graders will build your assignment using gcc, not g++, and your program must have the extension ".c", not ".cpp" or ".cxx". This means you must use C library functions like "printf" instead of C++ library functions like "std::cout".
- Please name the file that has the memory allocator functions `frac_heap.c` and name your test files: `test1.c`, `test2.c`, `test3.c`, ...
- Make sure that your test programs fully exercise your memory allocation functions. The program `main6.c` does not.
- Make sure your code builds against main6.c without any modifications to main6.c Your output does not need to exactly match that of main6.txt, depending on your allocation/deallocation strategy.

- Each time you need additional space in new_frac(), you must allocate a chunk of space large enough for exactly 5 new fractions.
- If malloc() returns a NULL, indicating you've truly run out of memory, you should print an error message and exit. (This is different from Project 5.)
- Here is a sample output from the sample main program: main6.txt. Your output does not have to look identical to this, but it gives you an idea of what should happen when you run the program.
- All of the files mentioned above (main6.c, frac_heap.h, etc.), as well as a Makefile that you can use to build your program, are available at
  `/afs/umbc.edu/users/p/a/park/pub/cmsc313/spring17/proj6/`

What to Submit

Use the UNIX `submit` command on the GL system to turn in your project.

You should NOT submit the header file `frac_heap.h`, since we provided a finished one for you, and we will be using exactly that one for testing your program.

You must submit the implementation `frac_heap.c` and your test programs `test1.c`, `test2.c`, ... You should resubmit your test programs even if you did not wind up changing them at all from what you submitted for the previous project.

In addition, submit a `typescript` file showing that your test programs compiled and ran.

You may optionally submit a README file explaining anything the graders might need to know about compiling and/or running your programs.

The UNIX command to do this should look something like:

```
submit cs313_park proj6 frac_heap.c test1.c test2.c test3.c
submit cs313_park proj6 typescript README
```