COMPUTER SYSTEMS
PROJECT 3
A simple heap allocator

1 Writing an allocator

We will be creating our own heap allocator. Specifically, we will define the following standard allocator functions:[1]

- `void* malloc (size_t size)`
  Allocate a block of at least `size` bytes and return a pointer to it.
- `void free (void* ptr)`
  Deallocate the block at `ptr`.
- `void* calloc (size_t nmemb, size_t size_each)`
  Allocate and clear a block of `nmemb` items of `size_each` bytes per item. That is, allocate and zero the bytes of `nmemb * size_each` bytes.
- `void* realloc (void* ptr, size_t size)`
  Change the size of the block at `ptr` to be `size` bytes in length (instead of whatever it was). Return a pointer to the newly resized block (which may be in the same location as the old one, or which may have been moved to a new location and the data from the old block copied into the new one).

These are standard functions, and the *standard C library* (a.k.a., `libc`) contains them. We would like to make programs that we run use **our** version of these functions instead of the ones in `libc`. That is, we would like to *interpose* our functions between the program and the `libc` functions. For now, we will simply write the functions and test them; later, we will find out how to make *other* programs use our allocator.

2 Introduction to *C*

Since you have not formally written, compiled, debugged, and run *C* source code before, we begin with a simple introduction to it. Here, we get more familiar with some standard *C* programming tools.

[1] The definitions of these functions given here are not rigorously complete. For a fuller definition of each, you should read its *manual page*. That is, at a shell prompt, use the `man` command to see the documentation of that funcion, e.g., `$ man malloc` will show a page of technical description of that function and other, related functions.

2.1 The compiler

The C compiler that we will use is `gcc`, the *GNU Compiler Collection* (or, once upon a time, the *GNU C Compiler*).[2] Although we have used this compiler to link some assembly code to existing `libc` procedures, we now want to use it to compile some actual *C* code:

The `sample` program: Grab the source code you will need for this part of the project (as well as later parts):

```
$ wget -nv -i https://bit.ly/AMHCS-2019F-171-p3
```

We will begin with the program, `sample.c`, which is a simple, starter program to monkey around with. Open that source code file with the editor of your choice. Then, having seen it, try compiling it, like so:

```
$ gcc -o sample sample.c
```

Assuming a successful compilation—no news is good news—then try running the program:

```
$ ./sample 3 7
2187
```

Ta da! You've compiled and run a simple C program. Toy with it to see how it works, change things around, etc.

2.2 The debugger

Another tool that we have used, and that will be even more valuable now that we are using *C* source code is the *source level debugger*, this case, `gdb`. To try it on your sample program, first recompile with *debugging symbols* included:

```
$ gcc -ggdb -o sample sample.c
```

Now, start the debugger. You can do so either at the command line...

```
$ gdb sample
```

...or, if you are using *Emacs*[3], you can start `gdb` within the editor by typing:[4]

```
M-x gdb
```

...which will be followed with a prompt to run the debugger like so:

[2] What's GNU? *GNU's Not UNIX*. Welcome to the world of recursive acronyms.

[3] Which you should. It's really handy for this kind of stuff.

[4] Note that `M-x` is Emacsish for *alt-x* or, for some computers, *ESC x*, where the *ESC* and the *x* are two separate keystrokes.

```
gdb --annotate=3 sample
```

From witin `gdb`, first *set a breakpoint* at the `main()` procedure:

```
(gdb) b main
```

Now run the program:

```
(gdb) run 2 8
```

You will see the program stop at the first line of `main()`, waiting for instruction. You can then `step` (or `s`) by a single line of code, or `continue` (or `c`) to let the program run without interruption.

Learn to use `gdb`! *Seriously.* You will resist, because you can get going with this assignment without really paying attention to learning more about how to use `gdb`. Get past your resistence. Try the `help` command; Use The Google to find `gdb` documentation and tutorials. Work through them, try stuff. Disassemble the `exp()` procedure and compare it to what you wrote by hand to calculate the same thing. Examine variables. Mess around with breakpoints and watchpoints (look it up!). As the projects get more involved, you will save yourself a *lot* of time and agony by having `gdb` to use.

3 Making a heap allocator

3.1 The allocator source code

Open, with your favorite editor, `pb-alloc.c`. There's plenty there, but most of all, there are the standard allocator functions described in Section 1. Additionally, there is a `main()` function at the bottom of the source code. For testing purposes, we will first compile this allocator as a standalone program, allowing for easier debugging. But first, some observations about this source code:

- `init()` is a procedure that initializes the heap, calling on the OS kernel to map a big space (2 GB), and setting up some pointers to keep track of that space. **This procedure must be called by the standard allocator procedures** in order to ensure that the heap is ready to be used. Notice that this procedure only performs its actions once; subsequent calls do nothing, and are therefore safe.

- `malloc()` and `free()` have empty procedure bodies—they don't do anything yet. More on completing those below, in Section 3.2.

- `calloc()` and `realloc()` *do* have complete bodies. They depend on `malloc()` and `free()`, as well as the helper procedures. They should need no modification in this version.

3.2 What you must do

Implement a *non-reclaiming, pointer-bumping heap allocator*. Here, `malloc()` will need to allocate requested blocks by moving forward (address-wise) through the heap. `free()` will do nothing, leaving dead blocks of memory unreclaimed.

Thinking ahead: The next assignment will allow blocks to be deallocated with `free()`. Thus, your allocator will need some way to keep track of blocks that have been freed, so it can find and then re-use them. What changes might that imply for your allocator?

3.3 How to compile and test your code:

When you have something ready to test, start by compiling it, again heeding any warning and errors:

```
$ gcc --std=gnu99 -ggdb -o pb-alloc pb-alloc.c
```

You should feel free to change `main()` to more heavily test the allocator's functions. Once compilation is successful, you can run it in `gdb` and, ultimately, run it on the command-line.

More motivation to use `gdb`: Printing debugging messages is a time-honored and effective way to debug code. However, *it won't work here!* The standard *C* procedure `printf()` (and its relatives) call on `malloc()` as they run. **You cannot debug `malloc()` by calling a procedure that will itself call `malloc()`.** If you do so, the infinite mutual recursion will get in your way. The *right* way to debug systems-level code like this is to use a proper debugger like `gdb`. So for the last time, **learn `gdb`**. Seriously. Just do it.

4 How to submit your work

Submit your `pb-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, Sep-26, 11:59 pm.