COMPUTER SYSTEMS
PROJECT 0
Interposing a simple allocator

1 Library functions and interposition

We will be creating our own library to contain our compiled allocator code. 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.

We will see how these functions can be written, and how to interpose them when running a program.

2 A simple starter program

In order to work with C code and make it do what we want it to do, we are going to have to become more familiar with some standard C programming tools.

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] To see how this compiler works, we will start with a simple, example C program.

[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 the command-line, 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.

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

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

```
$ wget -nv -i https://goo.gl/hwL8eV
```

We will begin with the program, `starter.c`, which is a simple, starter[3] program to monkey around with. To whit, open that source code file with the editor of your choice. Then, having seen it, try compiling it, like so:

```
$ gcc -o starter starter.c
```

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

```
$ ./starter 13
```

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

One of the benefits of C is the easy availability of a *source level debugger*, this case, `gdb`. To try it on your starter program, first recompile with *debugging symbols* included:

```
$ gcc -g -o starter starter.c
```

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

```
$ gdb ./starter
```

...or, if you are using *Emacs*[4], you can start `gdb` within the editor by typing:[5]

```
M-x gdb
```

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

```
gdb --annotate=3 starter+.
```

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

```
(gdb) b main
```

Now run the program:

```
(gdb) run 13
```

You will see the program stop at the first line of `main()`, waiting for instruction. You should try the `help` command, as well as Use The Google to find `gdb` documentation and tutorials.

[3] Duh.

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

[5] 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.

3 Your assignment

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 with `gdb`. Try compiling it and running it:

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

You should feel free to change `main()` to more heavily test the allocator's functions.

3.2 Compiling a shared library

Next, we want to make `pb-alloc` work on other programs. To do so, we need to compile a *shared library*, like so:

```
$ gcc -std=gnu99 -g -fPIC -shared -DPB_NO_MAIN -o libpb.so pb-alloc.c
```

There's a lot of goop there. `-fPIC -shared` tells the compiler that we want a *position independent shared library*. The `-DPB_NO_MAIN` defines the symbol `PB_NO_MAIN`, which signals the compiler to skip over the `main()` function when compiling (since libraries don't have their own `main()`).

3.3 Interposing the shared library

Now for the moment of truth. We can tell the shell to load **our** library first, thus making all uses of the allocator functions[6] link to our versions:

```
$ setenv LD_PRELOAD ./libpb.so
```

Now, any command that creates a new process will be loaded and linked with our library:

```
$ ls
pb!
pb-alloc.c pb-alloc libpb.so starter starter.c
```

3.4 What you must do

Document it. Wrap your head around the `pb-alloc.c` code, and write clear and complete documentation about how the allocator works overall, and how each function specifically does its thing. Expect to need to look things up and to ask questions to figure out all of the details.

[6]Not quite all. Internal calls from within the standard C library to its own allocator functions are already linked, and cannot be intercepted (easily).

4 Submitting your work

Submit your documentation (which can be either a commented `pb-malloc.c` or a separate document entirely) with the *CS submission system*, using one of the two methods:

- **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 Sunday, Sep-24, 11:59 pm.