Computer Systems
Project 3
A pointer-bumping heap allocator

1 Writing an allocator

We will be creating our own heap allocator. Specifically, we will implement 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 `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` itself.

2 Getting started

2.1 Creating the repository

1. **Login to the server:** Connect to the course server.
2. **Login to GitLab:** From your browser, login to
   `https://gitlab.amherst.edu`

[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. Indeed, the `man` command is generally quite helpful for providing documentation for all sorts of C functions and shell commands.

3. **Start a new project:** On the top toolbar of the GitLab window, click the little drop-down menu marked by a plus-sign. Select `New project`. Set the *Project name* to be `sysproj-3`, and leave the other default values. Click on the `Create project` button at the bottom.

4. **Clone the repository onto the course server:**

```
$ git clone git@gitlab.amherst.edu:yourusername/sysproj-3.git
$ cd sysproj-3
```

5. **Download the source code:** After you download the files, use `ls -l` to list the directory and see what you have.

```
$ wget -nv -i https://bit.ly/cosc-171-21f-p3
$ ls -l
```

6. **Add/commit/push the source code to the repository:**

```
$ git add *
$ git commit -m "Starting code."
$ git push
```

3 The pointer-bumping heap allocator

3.1 The allocator tester

Open, with your favorite editor, `memtest.c`, which is a very simple program that calls on `malloc()` and prints some results. Begin by compiling this code and running it:

```
$ make memtest
$ ./memtest
```

You will see the output, which is the program displaying the *addresses* that were returned for the calls to `malloc()` and held in the pointer variables `x`, `y`, and `z`. Here, the `malloc()` function that is providing these blocks is the `libc` version, since we have not yet done anything to trigger the use of our own allocator.

Notice that the addresses are shown in *hexadecimal* (a.k.a., *base 16*), with digits from 0 to 9, and a to f. You can see, in the least significant digits, how far apart each block of allocated memory is. You can use your computer's calculator app, which likely has a *programmer mode*, to do arithmetic on hex values and to convert them to and from decimal and binary.

3.2 The pointer-bumping allocator

Now open `pb-alloc.c`. Some observations about this source code:

- `init()` is a procedure that initializes the heap, calling on the OS kernel to *map*[2] a big space (2 GB), and keeping some static variables that keep track of that space. **This procedure is called by `malloc()`** to prepare the heap. Notice that this funtion only performs its actions once; subsequent calls do nothing, and are therefore safe.
- `malloc()` is fully written, but without any comments. More about this in Section 3.4.
- `free()` does nothing; it doesn't need to.
- `calloc()` is fully written and commented, and does what it needs to do. Check it out.
- `realloc()` is fully written, but without any comments. Again, see Section 3.4.

3.3 Compiling, using, and debugging pb-alloc

Compiling To compile the allocator:

```
$ make libpb
```

This command will compile the source code, and then link it into a *shared library*, which is code that can be loaded into a process and used by multiple programs. Specifically, it will generate the file `libpb.so`, which is that shared library. This file is **not** an executable program itself; it has no `main()` function.

Using To use the allocator, you must tell the OS to use functions in `libpb.so` *before* it uses any functions in `libc`, making programs link to our implementation of these functions instead of the standard implementations. To do that, at the shell and from within your `sysproj-3` directory:

```
$ export LD_PRELOAD=${PWD}/libpb.so
```

At this point, **all commands** will use our allocator. For now, let's just use `memtest` itself, and then turn off the use of our allocator with the `unset` command:

```
$ ./memtest
$ unset LD_PRELOAD
```

You will likely notice that the specific addresses shown by `memtest` are a bit different when it uses our allocator.

[2] Mark as valid for the process to use.

Debugging with output What if we want to see debugging output messages from within our code? You may have noticed, in the allocator code, a number of `DEBUG()` calls, but they don't seem to produce anything. To change that:

1. Open the `Makefile` file.
2. At the top, where `SPECIAL_FLAGS` is defined remove the comment marker (`#`) so enable the line that ends with `-DDEBUG_ALLOC`.
3. Just below it, insert the comment marker (`#`) to disable the line that defines `SPECIAL_FLAGS` without the trailing `-DDEBUG_ALLOC`.
4. Save and close `Makefile`
5. Recompile and link:

```
$ make clean
$ make libpb memtest
```

Now, when you enable the use of `libpb` and run `memtest`, you will see a whole stream of debugging messages. You can change these in your code however you like to show yourself what is happening inside the code.

Debugging with `gdb` You can start `gdb` on `memtest`, and **then** enable the use of `libpb`:

```
$ gdb memtest
[...]
(gdb) set environment LD_PRELOAD /home/yourusername/sysproj-3/libpb.so
(gdb) b main
(gdb) run
(gdb) b malloc
(gdb) c
```

So what is happening here? Once `gdb` is running, we set `LD_PRELOAD` to use our library. However, it won't actually load it until the program actually starts. So, we set a breakpoint at `main()`, stopping the program just as it is getting started. Now we can set a breakpoint at one of our functions (here, `malloc()`), and then instruct `gdb` to *continue* (`c`).

The program will then stop within `gdb` as soon as it reaches `malloc()`. You can then step through one line of code to the next (with the `n` command), you can print variables (`p` or `p/x`) to see what's happening, and watch the progression. If your code has a bug that causes a *segmentation fault*, `gdb` will catch the error and then allow you to see the *backtrace* (`bt`), which is a printing of the stack to see where the program is, and how it got there.[3]

[3] Suffice it to say that you should search the web for tutorials and documentation on `gdb`. It's amazingly handy and worth learning how to use.

3.4 Your assignment

You have **three tasks** to perform:

1. **Add comments:** The `malloc()` and `realloc()` functions are implemented, but they're not commented. Add comments to the code to describe what is happening. **Work in groups** to discuss and figure out what each part of these functions is doing, and provide comments that would help someone exactly like you to open this code and understand it.

2. **Fix the alignment problem:** This allocator does its job, but it **does not return double-word aligned blocks**, which it should.[4] Update the code in `malloc()` to pad the blocks such that they will always at addresses that are multiples of 16.

3. **Enhance the tester:** Modify `memtest.c` so that it tests the behavior of `realloc()`, which should allocate and copy the contents into a new block *only if* the new requested size is larger than the old. Does it allocate a new block only when it should? Does it copy the contents correctly?

   Also modify it to test whether blocks are being returned at double-word aligned addresses.

4 How to submit your work

First, be sure that the most recent versions of your work are up-to-date on the GitLab server by performing an *add/commit/push* with `git`. Then, go to GitLab with your browser, and add me (*sfkaplan*) as a *Developer* to your repository.

This assignment is due on Sunday, Sep-26, 11:59 pm.

[4] This requirement exists for the same reason that stack frames needs to be double-word aligned.