Assignment 03

If any of this is unclear, please get in touch with me ASAP. There are definitely things in the course I want you to struggle with. I absolutely do NOT want you struggling with understanding my expectations.

In this assignment, you will implement replacements for `malloc(3)` and `free(3)`, along with variants of the former, `calloc(3)` and `realloc(3)`, and a related function, `malloc_usable_size(3)`. As discussed in class, for efficiency reasons, these are *not* system calls; instead, they use the `brk(2)` and `sbrk(2)` system calls to request large chunks of memory from the kernel, out of which `malloc(3)` itself satisfies the individual requests. (In this respect, it's more than a little similar to myread.) On the flip-side, `free(3)` reclaims chunks of memory that are no longer in use.

You will create a dynamically-linkable library that contains your implementations, and use the `LD_PRELOAD` mechanism (use described below) to cause it to be loaded and linked *before* `libc`, which contains the official implementations of `malloc(3)` and `free(3)`. By linking it before `libc`, the implementations in your library will be the ones whose addresses get put in the global offset table, and hence whenever any program run in that manner calls `malloc`, it will get your implementation and *not* the offical one.

This presents an interesting problem: functions like `printf(3)` use `malloc(3)` to do their thing and, during development, if your implementation isn't yet correct, then `printf(3)` isn't going to work! Two complementary suggestions: a) use `write(2)` instead (recall that the file descriptor corresponding to stdout is 1); 2) make extensive use of `gdb`. (How to use `gdb` with `LD_PRELOAD` is described below.)

Without `printf(3)`, though, we lose the ability to format pointers and integers as human-readable text (recall the difference between how numerical values are represented in memory and the ASCII characters that humans use to interpret such a value). I have written a function to help you with this; both the function and driver code are available here: pointer_to_hex_le.c. I have also written a similar function (along with driver code) for unsigned integers: uint64_to_string.c.

Your implementations need not be terribly clever, but they do need to work, even when real programs like `ls` and `cat` use them. I am being intentionally vague about what it means to "work" because I want you to think about your use of `malloc(3)` in the past, the behavior of the relevant system calls, and how you can unify these ideas.

NOTE: The pointers returned by your implementation of `malloc` *must* be evenly divisible by 16. (And it is probably a good idea if the bookkeeping struct is also on a 16-byte boundary.)

For context, curiosity, and light bedtime reading, you might be interested in how the real thing works. There are actually many different implementations running around; some of the biggies (at least in the UNIX world) are/were:

- Doug Lea's malloc ("dlmalloc"), probably best described in this Phrack paper;
- jemalloc, originally presented in this paper; and
- ptmalloc, currently used in glibc, and for which I could not find a good description (the typical response being, "Use the Source, Luke!").

New Mechanics

You might find these useful in this assignment.

New system calls/functions

- `brk(2)`
- `sbrk(2)`
- `memset(3)`

Language features

You might be tempted to declare a global variable in your library. Under normal circumstances, such a variable would become a symbol that can be linked by a process that loads your library (ie, it is *exported*). This might be harmful if the name of your variable happens to match the name of another variable already in the process. To get around this, you can precede the declaration of your variable with the `static` keyword, which restricts the symbol to the file in which it is declared—ie, it is not exported.

Building, running, and debugging

You need a few extra arguments to `gcc` to create a shared library. Therefore, to compile `my-malloc.c` as a library called `my-malloc.so` that can be linked at runtime, use a command line this:

```
$ gcc -g -Wall -pedantic -rdynamic -shared -fPIC -o my-malloc.so my-malloc.c
```

If that `my-malloc.so` library defines functions like `malloc` and `free` (just to pick two completely random examples), you can cause your implementations to override the `libc` ones in a single process, start said process with a command like this:

```
$ LD_PRELOAD=./my-malloc.so ls
```

As I said in class, you will likely find `gdb` really helpful for this assignment. If you use the `LD_PRELOAD` trick above with `gdb`, however, anytime `gdb` wants to use `malloc`, it will use your implementation, which might be unhelpful if your implementation is buggy. Therefore, to cause `gdb` to use the special library only for the program to be debugged, use a command like this:

```
$ gdb --args env LD_PRELOAD=./my-malloc.so ./test-malloc
```

If you find yourself using these commands over and over again, you might want to figure out ways to make them more convenient to run—more convenient even than pressing the up-arrow through your shell history. (Hint: Makefile.)

Behavioral requirements

Your library must implement the functions `malloc`, `free`, `calloc`, `realloc`, and `malloc_usable_size` with the behavior described in the relevant manpages. When this library is pre-loaded, modest programs like `ls` and `cat` must work correctly. I won't test your library against ambitious programs like Firefox. (For the record, my implementation works with vim, though, and it's pretty usable.)

Your library may *not* just ask `sbrk(2)` for a bazillion bytes once at the beginning and then allocate out of that. It must be moderately intelligent about growing the size of the heap as necessary.

Your library should, where possible, reuse chunks of memory that have been relinquished using `free`.

Addresses returned by your implementation of `malloc`, `calloc`, and `realloc` must be 16-byte aligned.

Deliverable requirements

Three files:

- `my-malloc.c`, which implements the functions described above.
- `Makefile`, whose default target should build the aforementioned source file into a shared library named `my-malloc.so`. It may have as many other targets as you like (and probably should, to make your life easier), but the default one *must* build the library `my-malloc.so`.
- `README`, which will include a list of authors, a list of known bugs, and a list of resources you consulted in your work (a list of URLs is totally acceptable—if I can see the kinds of references you're using, I can get a better idea of how to tailor the instruction to your needs—just keep the README file open as you work and paste them there when you find something useful).

Style requirements

Your code must exhibit:

- appropriate use of comments;
- appropriate use of stack, heap, global, and static variables;
- appropriate use of constants;
- appropriate decomposition;
- clear organization (eg, grouped header files, struct definitions, function prototypes);
- well-named variables, functions, and other identifiers;
- verification of command-line arguments, if applicable;
- checking and appropriate handling of all return values;
- consistent style (capitalization, indenting, etc).

Submission Process

Your submission files should be in a git repository on basin in this directory:

```
/home/<username>/<secret>/assignment3
```

Email me when you want feedback, with a description of the type of feedback you would like.

Last modified: 10/27/2022 14:37:26