Project #3: Memory Allocator*

1 Objectives

This project will give you experience with implementing and optimizing data structures and algorithms for memory allocation, using `mmap` to request virtual memory from the OS, and replacing library calls at run-time like we did in project #1 using the LD_PRELOAD environment variable.

In this project, you will write your own memory allocator, as a shared library that can be used with any existing program. Your allocator will implement the standard C allocation API (`malloc`, `calloc`, `realloc`, and `free`).

2 Allocator details

Your allocator will be implemented in C and run as a layer between a program and the operating system. The program will make requests for memory (i.e., `malloc(118);`) and your layer will, in turn, request pages of virtual memory from the operating system, using the `mmap` system call.

Input: The "input" to your program will be a series of calls to `malloc`, `calloc`, `realloc`, and `free`. Your library will handle these calls and provide blocks of memory appropriately.

Output: Your allocator should **not** produce any output. Any output will probably cause the grading program to mark your solution as incorrect.

Implementing the Allocator: Your allocator will be a BiBOP (Big Bag Of Pages)-style allocator with segregated free lists. Your allocator will store small objects in lists segregated by different sizes, in powers of two: 2, 4, 8, …, 1024. For small objects, your allocator will request memory from the OS in 4K chunks (pages), using the **mmap** system call. For objects larger than 1024 bytes, you should allocate a separate chunk of memory for the

*This project is based on one developed by Emery Berger and Mark Corner at UMass Amherst

Listing 1: Requesting a page from the OS.

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <string.h>

#define PAGESIZE 4096

//used for initializing the memory to zero.
int fd = open("/dev/zero", O_RDWR);

//ask the OS to map a page of virtual memory
//initialized to zero (optional)
//initializing your memory may make debugging easier.
void * page = mmap (NULL, PAGESIZE,
                    PROT_READ | PROT_WRITE,
                    MAP_PRIVATE, fd, 0);

//unmap the page, when you're done.
munmap(page, PAGESIZE);
```

object using **mmap**. Each page should contain only objects of a single size. When a large block is freed, the **munmap** system call can be used to free or unmap those pages. Listing 1 shows how you will do this. You do not need to free pages for your small block lists. Just keep them around in case they are needed in the future.

Your allocator may not use the system allocator—you may not call the **brk** system call or the system's **malloc**, **realloc**, **calloc**, or **free** directly, or indirectly. Calling a function that allocates memory for you (other than mmap and munmap) will be considered cheating. Also, note that **printf** and it's related functions often allocate memory on the heap (specifically when doing variable substitution). Calling these functions (without taking appropriate precautions) within your library will most likely crash the running process (due to infinite recursion).

Your allocator may use some global variables of reasonable size (<2K) to implement the allocator. These variables should be allocated statically (not on the heap) when the process starts.

You may organize your pages and objects however you choose, as long as each page only contains objects of a single size; however, your allocator will be graded on functional correctness, as well as its time and space efficiency. So, I recommend the following:

- Only open `/dev/zero` once, if at all.
- Each page should have a header of some kind that contains metadata that is needed to locate the objects stored on the page.
- Word-aligning objects within a page may improve the efficiency of some instructions (but probably only a little).
- Think about what information you need to store, and what information you can do without.
- Consider maintaining both free and allocated lists. Consider using linked lists or other linked data structures to improve performance. Note that these lists will be a little different than those you have used before, since you can't use **malloc**. You will have to place the nodes yourselves.
- Start with something simple that works. Functional correctness is weighted more heavily than efficiency. So, put something together that works well, first. Then worry about trying to improve space and time efficiency.

3 Compiling your allocator

During development and debugging, you should compile your allocator as a shared library, like so.

```
gcc -g -Wall -fPIC -shared allocator.c -o libmyalloc.so
```

When trying to squeeze out every last ounce of performance, you should compile your allocator as a shared library, like so.

```
gcc -O2 -DNDEBUG -Wall -fPIC -shared allocator.c -o libmyalloc.so
```

The `-O2` and `-DNDEBUG` options are optimizations that should be used for performance reasons. They may make your library run faster, but harder to debug. The `-DNDEBUG` option, for example, will remove your assertions from your program. You may use `clang` instead of `gcc`, if you like. Your source file can also be named something other than **allocator.c** (and you can have multiple source files), but your library **must** be named **libmyalloc.so**. Otherwise, the grading script will not be able to test your solution.

To run a program (**test**) with your library, run the following command.

```
% LD_PRELOAD=./libmyalloc.so ./test
```

If you want to debug your allocator, it is often handy to compile it as part of a test program, rather than as a shared library. This is done, as follows:

```
gcc -Wall -g -o test allocator.c test.c
```

This will make it easier to use `gdb`, `valgrind` or other similar tools to step through your allocator code, which will likely be extremely important.

Debugging your allocator being used by another program is also possible (assuming that you compile your library with debug information included, `-g`). If you wanted to figure out why your allocator SEGFAULTs when run with `ls`, you would do something like this:

```
gdb ls
(gdb) set environment LD_PRELOAD=./libmyalloc.so
(gdb) start
```

3.1 Testing your allocator

You should test your library against both real programs and test programs. Note that if you `export` the LD_PRELOAD variable then command line tools, like `time`, `grep`, and `gdb` will also be using your allocator. If your allocator doesn't work yet, this will give you results that may be very confusing. So, maybe don't do that.

Your program should compile and run on the McAdams lab machines. Lack of thorough testing on the lab machines is asking for an extremely undesirable outcome.

I will give you a few tests via the git repository. These tests are very basic tests, and passing these tests does not guarantee a perfect (or even good) score on the project. There will be additional **private** test cases that I will use in grading. You should write additional tests that test your allocator more thoroughly, especially looking at the boundary conditions.

I will expect thorough testing, but I do have my limits. For example, I'm not going to test your code on complex multi-threaded apps, like Firefox. I have decided to limit testing of your allocator to single process, single threaded, GUI-less programs (e.g., `ls`, `df`, `wget`, `cat`, and of course hand-crafted programs that explore boundary conditions).

Note, that this project will have more boundary conditions than the previous two. Consider using random testing—create a program that randomly allocates, reallocates, and frees blocks of memory of random size. This will help you explore possibilities that you haven't yet thought of.

4 Submission Instructions

This project is due by 4:00 PM on April 4^{th}. Absolutely no late assignments will be accepted.

When your project is complete, archive your source materials, using the following command:

```
> tar cvzf project3.tgz README Makefile <list of source files>
```

The **Makefile** should compile your shared library (by running **make**).

The **README** file should include your name, a short description of your project, and any other comments you think are relevant to the grading. It should have two clearly labeled sections titled with KNOWN PROBLEMS, and DESIGN. The KNOWN PROBLEMS section should include a specific description of all known problems or program limitations. This will not hurt your grade, and may (in rare cases) improve it. I am more sympathetic when students have thoroughly tested their code and clearly understand their shortcomings, and less so when I find bugs that students are ignorant of or have tried to hide. The DESIGN section should include a short description of how you designed your solution (especially anything you thought was interesting or clever about your solution). You should also include references to any materials that you referred to while working on the project. Please do not include special instructions for compilation. The compilation and running of the tests will be automated (see details below) and your instructions will not be followed.

Please make sure you include only those source files, not the object files. Before you submit this single file, please use the following commands to check and make sure you have everything in the `project3.tgz` file:

```
> mkdir temp; cd temp
> tar xvzf /path/to/project3.tgz
> make
```

These commands should put all of your source files in a temporary directory. It should not create any subdirectories within that directory. Running `make` after extracting the files should build your **`libmyalloc.so`** library, which should end up in the temporary directory.

Submit your `project3.tgz` file via **handin.cs.clemson.edu**. You must name your archive **project3.tgz**, and it must compile and work. We will compile your library using your **Makefile**, and use it (via LD_PRELOAD) with a variety of test programs.

5 Grading

Your project will be graded based on the results of functional tests, performance tests (both time and space) and the design of your code. We will run several tests to make sure it works properly. We will time your library and track its space-efficiency (by measuring the number of `mmap` calls that occur) when running several benchmark programs. It should not crash. Specifically, you will receive 10% credit if your code successfully compiles (without any errors), and 10% for code style and readability. The rest of your score will be determined by your library's functionality (70%) and efficiency (5% for time and 5% for space).

Solutions that are within 2x of my solution's time performance and within 1.5x of my solution's space usage will be considered good enough for full credit. I will give up to 5% extra credit to solutions that receive full credit and are faster or more space efficient than my solution.

Efficiency is a relatively small part of your grade. You should focus on creating a working library first, before you even think about optimizing anything.

Your source materials should be readable, properly documented and use good coding practices (avoid magic numbers, use appropriately-named variables, etc). You code should be `-Wall` clean (you should not have any warnings during compilation). **Our testing of your code will be thorough. Be sure you test your application thoroughly.**

6 Collaboration

You will work independently on this project. You must *not* discuss the problem or the solution with classmates, and all of your code must be your own code.

You may, of course, discuss the project with me, and you may discuss conceptual issues related to the project that we have already discussed in lecture, via Piazza (do not post any code or implementation details of the algorithms). **Collaborating with peers on this project, in any other manner will be treated as academic misconduct.**