Assignment #1: Malloc Library Part 1

ECE 650 – Spring 2022

See course site for due date

General Instructions

1. You will work individually on this the project. **Remember you are not allowed to see anyone else's code, to show your code to any other student, to copy from any solution you may find on the internet. Any of this would be a violation of the Duke code of conduct and will result in disciplinary action.**
2. The code for this assignment should be developed and tested in a UNIX-based environment. You can use the same VM that has been made available to you for ECE551. **We will be testing your delivery in that environment, please be sure there are no porting issues.**
3. You must follow this assignment specs carefully, and turn in everything that is asked (and in the proper formats, as described).
4. You should plan to start early on this project and make steady progress over time. It will take time and careful though to work through the assignment.

Implementation of malloc library

For this assignment, you will implement your own version of the dynamic memory allocation functions from the C standard library (actually you will have the chance to implement and study several different versions as described below). Your implementation **is to be done** in C code.

The C standard library includes 4 malloc-related library functions: `malloc()`, `free()`, `calloc()`, and `realloc()`. In this assignment, you only need to implement versions of `malloc()` and `free()`:

```
void * malloc(size_t size);
void free(void * ptr);
```

Please refer to the `man` pages for full descriptions of the expected operation for these functions. Essentially, `malloc()` takes in a size (number of bytes) for a memory allocation, locates an address in the program's data region where there is enough space to fit the specified number of bytes, and returns this address for use by the calling program. The `free()` function takes an address (that was returned by a previous `malloc` operation) and marks that data region as available again for use.

The submission instructions at the end of this assignment description provide specific details about what code files to create, what to name your new versions of the `malloc` functions, etc.

As you work through implementing `malloc()` and `free()`, you will discover that as memory allocations and deallocations happen, you will sometimes free a region of memory that is adjacent to other also free memory region(s). **Your implementation is *required* to coalesce in this situation by merging the adjacent free regions into a single free region of memory.**

Hint: For implementing `malloc()`, you should become familiar with the `sbrk()` system call. This system call is useful for: 1) returning the address that represents the current end of the processes data segment (called program break), and 2) growing the size of the processes data segment by the amount specified by "increment".

```
void *sbrk(intptr_t increment);
```

Hint: A common way to implement `malloc()` / `free()` and manage the memory space is to keep an adequate data structure to represent a list of free memory regions. This collection of free memory ranges would change as `malloc()` and `free()` are called to allocate and release regions of memory in the process data segment. You may design and implement your `malloc` and `free` using structures and state tracking as you see best fit. Please devote some time to decide on which data structure is most adequate for the needs of memory allocation and deallocation.

In this assignment, you will develop a malloc implementation and study different allocation policies.

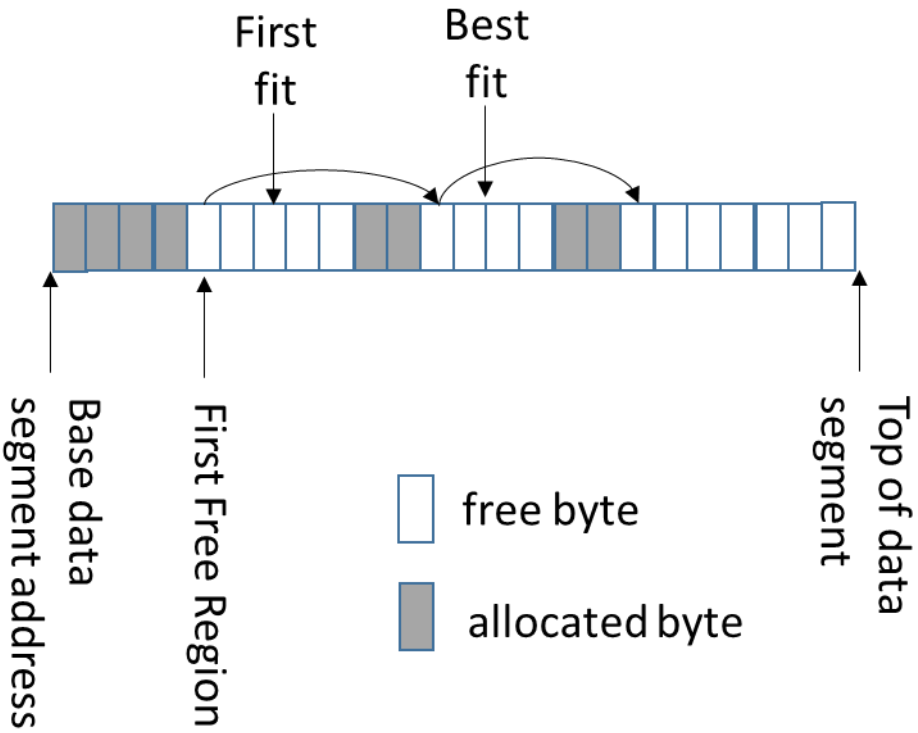
Study of Memory Allocation Policies

Your task is to implement 2 versions of `malloc` and `free`, each based on a different strategy for determining the memory region to allocate. The two strategies are:

1. **First Fit**: Examine the free space tracker (e.g. free list), and allocate an address from the first free region with enough space to fit the requested allocation size.

2. **Best Fit**: Examine all of the free space information, and allocate an address from the free region which has the smallest number of bytes greater than or equal to the requested allocation size.

The following picture illustrates how each strategy would operate (assuming free regions are traversed in a left to right order) for a `malloc()` request of 2 bytes:



Requirement: `malloc()` implementations

To implement your allocation strategies, you will create 4 functions:

```
//First Fit malloc
void * ff_malloc(size_t size);

//Best Fit malloc
void * bf_malloc(size_t size);
```

Note, that in all cases, a policy to minimize the size of the process's data segment should be used. In other words, if there is free space that fits an allocation request, then `sbrk()` **should not be used** to increase the memory segment.

Requirement: `free()` implementations

To implement your memory deallocation strategies, you will create 2 functions:

```
//First Fit free
void ff_free(void * ptr);

//Best Fit free
void bf_free(void * ptr);
```

On `free()`, your implementation is required to merge the newly freed region with any currently free adjacent regions. In other words, your bookkeeping data structure should not contain multiple adjacent free regions, as this would lead to poor region selection during malloc.

You **are not** required to perform any type of garbage collection (e.g. reducing the size of the process's data segment, even if allocations at the top of the data segment have been freed).

Requirement: Performance study report

In addition to implementing the four functions above, you are asked to conduct a performance study of the `malloc()` with different allocation policies. Several programs for experimentation will be provided that perform `malloc()` and `free()` requests with different patterns (e.g. frequencies, sizes). The metrics of interest will be:

1) the run-time of the programs, as the implementation of different allocation policies may result in different speeds of memory allocation;
2) the memory fragmentation is a number in the interval [0,1] (interpreted as a percentage), which is measured as follows:

$$1 - (\text{size of largest allocatable memory block} / \text{total size of free memory})$$

In order to capture #2, you should also implement two additional library functions:

```
unsigned long get_largest_free_data_segment_size();//in bytes

unsigned long get_total_free_size();//in bytes
```

Then, using these functions, you can use the included test programs as well as test programs of your own creation to evaluate and compare the algorithms.

The Starter Kit

A starter kit is included in a file on Sakai: **homework1-kit.tgz**

This archive can be extracted using "`tar xvzf homework1-kit.tgz`".

The kit includes:

- **`Makefile`**: A sample Makefile for libmymalloc.so.
- **`general_tests/`**: General correctness test, see README.txt for details.
- **`alloc_policy_tests/`**: Allocation policy test cases, see README.txt for details.
- **`alloc_policy_tests_osx/`**: Same thing adapted for MacOS.
  NOTE: Mac OS is not the grading platform; these files are provided as a convenience. Additionally, you may need to change #include "time.h" to #include "sys/time.h"

Testing your system

Code is provided for minimal testing, but the provided materials will not exhaustively evaluate the correctness of your implementation. It is recommended to create your own test software that uses your library, both to aid during development and to ensure correctness in a variety of situations.

We will make reference to ECE551 standards for code quality when grading your delivery. Therefore, you must return software that is adequately commented, portable, indented, readable, and does not show undefined behavior.

Detailed Submission Instructions

1. Please submit a written report called **report.pdf** to **Sakai** (a submission link will be posted soon). The report should include an overview of how you implemented the allocation policies, results from your performance experiments, and an analysis of the results (e.g. why do you believe you observed the results that you did for different policies with different malloc/free patterns, do you have recommendations for which policy seems most effective, etc.).

2. Please submit a zipped file **hw1_netID.zip** to **Sakai** (e.g., if the netID is abc123, the name of the file should be hw1_abc123.zip). All source code should be included in a directory named "**my_malloc**".

    - There should be a header file name "**my_malloc.h**" with the function definitions for all *_malloc() and *_free() functions.

- You may implement these functions in “**my_malloc.c**”. If you would like to use different C source files, please describe what those are in the report.
- There should be a “**Makefile**” which contains at least two targets: 1) “all” should build your code into a shared library named “libmymalloc.so”, and 2) “clean” should remove all files except for the source code files. The provided Makefile may be used as-is, expanded upon, or replaced entirely. If you have not compiled code into a shared library before, you should be able to find plenty of information online. With this “Makefile” infrastructure, the test programs will be able to: 1) #include “my_malloc.h” and 2) link against libmymalloc.so (-lmymalloc), and then have access to the new malloc functions. Just like that, you will have created your own version of the malloc routines in the C standard library!