Assignment #1: Malloc Library

ECE 650 – Fall 2017

Due by 11:59pm eastern time on Friday, February 3

General Instructions

1. You will work individually on this the project.
2. The code for this assignment should be developed and tested in a UNIX-based environment.
3. You must follow this assignment spec carefully, and turn in everything that is asked (and in the proper formats, as described). Due to the large class size, this is required to make grading more efficient.
4. Part of this assignment deals with multi-threaded execution. Note that **bugs in multi-threaded code are often timing dependent (i.e. race conditions),** This means that incorrect parallel code may (quite often) result in correct execution due to the absence of certain timing conditions in which the bugs can manifest. Therefore, manual code correctness analysis is very important, and **your code will be graded based on code correctness rather than on output correctness from particular test runs**.
5. You should plan to start early on this project and make steady progress over the next two weeks. 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 several 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 a data structure that represents 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.

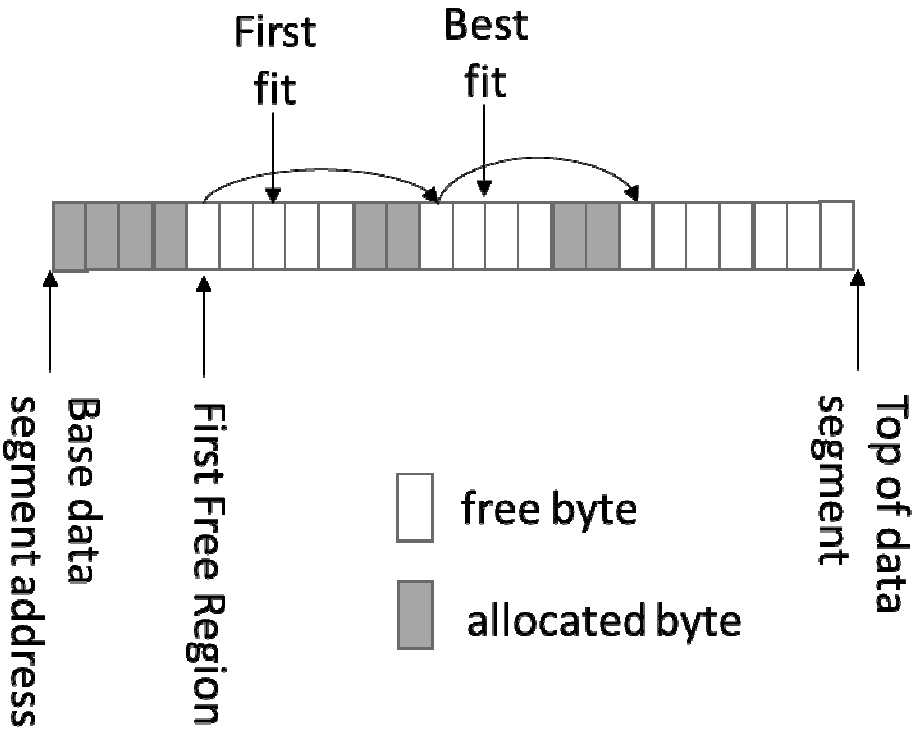
Part 1: Study of Memory Allocation Policies

In the first part of the assignment, 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:



To implement your 3 allocation strategies, you will create 4 functions:

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

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

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 no free space that fits an allocation request, then sbrk() should be used to create that space. However, you do not need 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).

In addition to implementing these malloc functions, you are tasked to conduct a performance study of the malloc() performance 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 program as the implementation of different allocation policies may result in different amounts of memory allocation overhead, and 2) fragmentation (i.e. the amount of allocated data segment space divided by total data segment space). In order to capture #2, you should also implement two additional library functions:

```
unsigned long get_data_segment_size(); //in bytes
unsigned long get_data_segment_free_space_size(); //in bytes
```

Your submission should include two components:

1. Source code (specifics described in the "submission instructions").

2. A written report to describe both 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.).

Part 2: Thread-Safe malloc() implementation

In this part, you will implement a thread-safe version (i.e. safe for concurrent access by different threads of a process) of the malloc() and free() functions. Your thread-safe malloc and free functions should **use the best fit allocation** policy.

```
//Thread Safe malloc/free
void *ts_malloc(size_t size);
void ts_free(void *ptr);
```

In order to exercise and test the thread-safe malloc routines, pthread-based multi-threaded sample programs will be provided. These programs will create threads which will make concurrent calls to ts_malloc() and ts_free(). You are also encouraged to create your own multi-threaded test programs to test your library code. In order to make your functions thread-safe, you will need to provide synchronization to eliminate any data races. Recall that concurrent access means that multiple threads performing ts_malloc() and ts_free() thus may be reading and updating shared data structures used by the malloc routines (e.g. free list information).

To provide necessary synchronization, you may use support from the pthread library and needed synchronization primitives (e.g. pthread_mutex_t). Also, don't forget about Thread-Local Storage! You may find that very useful.

Additionally, your implementation should strive for high performance. In other words, acquiring and releasing a single lock at the start and end of each ts_malloc() and ts_free() call is not a very impressive solution. Your solution should strive to allow concurrency where possible.

Your submitted code for the thread-safe malloc implementation will be tested against the provided sample test programs, as well as additional tests. So you are encouraged to: 1) reason thoroughly through your implementation to ensure there are no race conditions, and 2) create your own additional test cases if you feel they will be helpful to you.

For this part, you should also include the source code in your submission, and additionally you should add to the report. You should describe your thread-safe malloc implementation. For example, what critical sections did you identify, and what was your synchronization strategy to prevent race conditions? This writeup should provide a clear description of your solution.

Detailed Submission Instructions

You will submit this project as a single zip file named **hw1.zip**. Submit hw1.zip by uploading the file into your Sakai drop box. The zip file should contain exactly the following:

1) The report writeup (report.pdf or report.doc), addressing all items described above.
2) All source code 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. If you have not compiled code into a shared library before, you should be able to find plenty of information online, or just talk to the instructor or TA for guidance!
    - 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!