# Project #1: Malloc Library Part 1

# ECE 650 – Spring 2022

**See course site for due date**

## General Instructions

1. You will work individually on this project.
2. The code for this assignment should be developed and tested in a UNIX-based environment, specifically the Duke Linux environment available at login.oit.duke.edu or https://vcm.duke.edu/.
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. You should plan to start early on this project and make steady progress over time. It will take time and careful thought to work through the assignment.
5. You have a chance to preview your program results tested by our auto-grader if you submit your homework on Gradescope 2 days before the due date. Then you are allowed to improve your program and regrade before the due date.

## 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. Similarly, it is also required to split free regions if the ideal free region is larger than requested size.**

**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 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.

**In this assignment, you will develop a malloc implementation and study different allocation policies. In Homework 2, you will make this implementation thread-safe.**

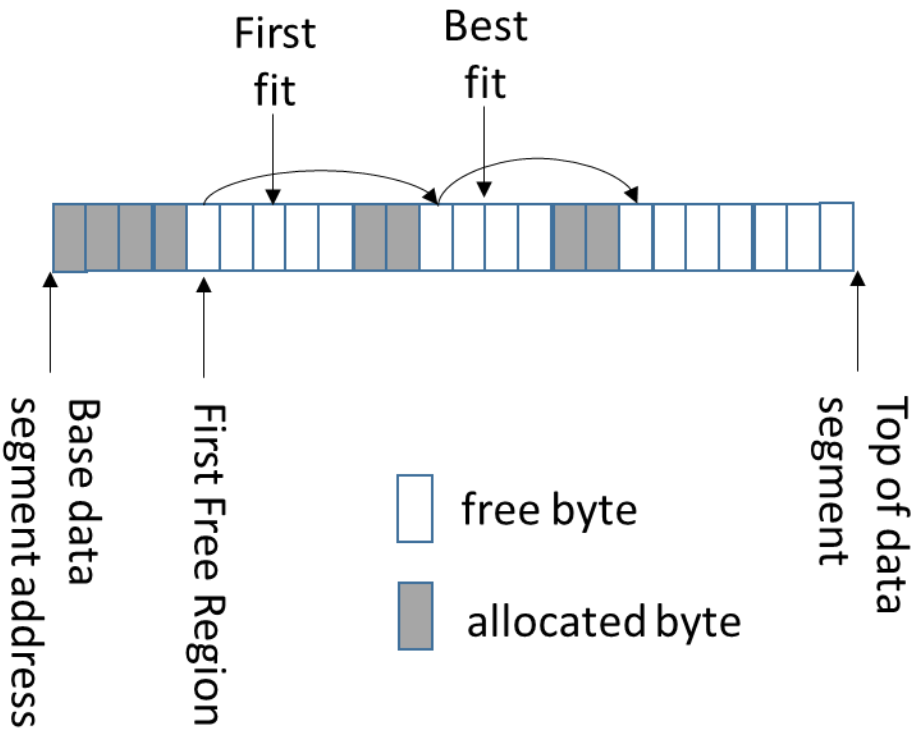
## Study of Memory Allocation Policies

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/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. Moreover, when a request for allocation is received, **only the requested size of memory must be allocated** (with the exception that the remaining free space is too small to keep track of). 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).

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. Since we are managing the memory space by ourselves, there is no need to call the original free() function.

## Requirement: Performance study report

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 unallocated 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
```

The functions must include the spaced used for metadata also.

`get_data_segment_size() = entire heap memory (this includes memory used to save metadata)`

`get_free_space_segment_size() = size of the "free list" = (actual usable free space + space occupied by metadata) of the blocks in your free list`

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 in a folder under Resources: **homework1-kit.tgz**

This archive can be retrieved on the command-line using `wget` if hosted on a website or downloaded to a local computer and transferred to the VM using `scp` for *nix/Mac systems or by using a file transfer app on Windows (Mobaxterm for Windows has built-in file transfer with drag and drop). It can then 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

**tl;dr: Write your own tests!** At least a basic print data structure and minimal testing can help you catch several bugs.

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. If you'd like a general introduction to developing test cases, see Software Testing by Sarah Heckman.

For debugging make sure to use tools such as gdb and valgrind. Please contact TAs and get help if you are not comfortable with using these tools. Also, please use the relevant online

resources that may help you. Make sure to check for any errors in valgrind as these may be uncaught bugs in your program.

### A refresher on gdb in emacs

- In your Makefile(s), change -O3 to -ggdb3. -O3 is an optimization flag and -ggdb3 is a debugging flag. -ggdb3 is required to let gdb add information to the compiled binary that will help it pinpoint exact line numbers, function names, etc.
- Here is a link to how you may use gdb in emacs: AoD_1: Debugging with GDB in Emacs You may have a better reference video, please feel free to share that with others.
- In emacs open any file in the directory where you have the binary you need to run. e.g. if it is mymalloc_test from general_tests, navigate to that directory and open a file.
- Then run M-x gdb
- Then type (r) for run.
- Your code ends in a seg-fault--If you have followed all the above steps, gdb in emacs already shows you the line on which the seg-fault has occurred.
- You can then run (bt) for backtrace which will show you which series of function calls led you here, e.g. line number in main test file which resulted in this function call.
- Please consult this GDB cheat sheet GDB QUICK REFERENCE GDB Version 4 for more.

### Valgrind

Some flags that may be useful with valgrind:

```
valgrind -v --leak-check=full --track-origins=yes ./program
```

## Getting Help

Please read Eric Steven Raymond's How To Ask Questions The Smart Way. If you don't know how to do anything mentioned in the document, make sure to look it up online first and then seek further clarification with more details in your question. The TAs are always happy to help. However, remember that you need to help them help you.

## Grading Rubrics

code correctness(54'), 12 test cases total

code check(16'), splitting and merging blocks required

report(30'), including implementation description, performance result presentation and result analysis

## Detailed Submission Instructions

1. A written report called **report.pdf** that includes 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. 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. 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, 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!