Introduction

The purpose of this assignment is to develop a dynamic memory manager. This memory manager will replace the functionality of the **malloc** and free routines built into the operating system.

The heap is considered a contiguous chunk of memory from which storage is allocated using **malloc** and deallocated using free. As storage is **malloc'ed** and freed, memory fragments may appear. A memory manager needs to keep track of the allocated spaces and fragments in order efficiently satisfy memory requests. Furthermore, as memory is freed, fragments must be coalesced to ensure optimal memory use.

Your goal for the assignment is to implement a memory manager, provide a means for a user to allocate and free memory all while keeping statistics of the memory management system.

Coding work on the assignment may be done with a ***partner***. You are also welcome to collaborate with other class members, but the end project work must be your own work. The project reflection and answer to project questions must be done ***individually*.**

Background and References

Your memory manager will have to mimic the behavior of **malloc** and free. It might be helpful to refresh your memory on how **malloc** and free work. Furthermore, you will need to ensure synchronized access to the memory manager in the case where a client application is using pthreads.

The following man pages might be helpful for review:

- man malloc
- man free
- man pthread_mutex_init
- man pthread_mutex_destroy
- man pthread_mutex_lock
- man pthread_mutex_unlock

To view the man pages for the pthread library, they must be installed on your OS image. You can do that by running the command:

```
sudo apt-get install manpages-posix manpages-posix-dev
```

Operating Systems (CSC3210 – Spring 2024)
Week 14 – Memory Management Project DUE: Friday, May 3, 2024 at 11:59 PM

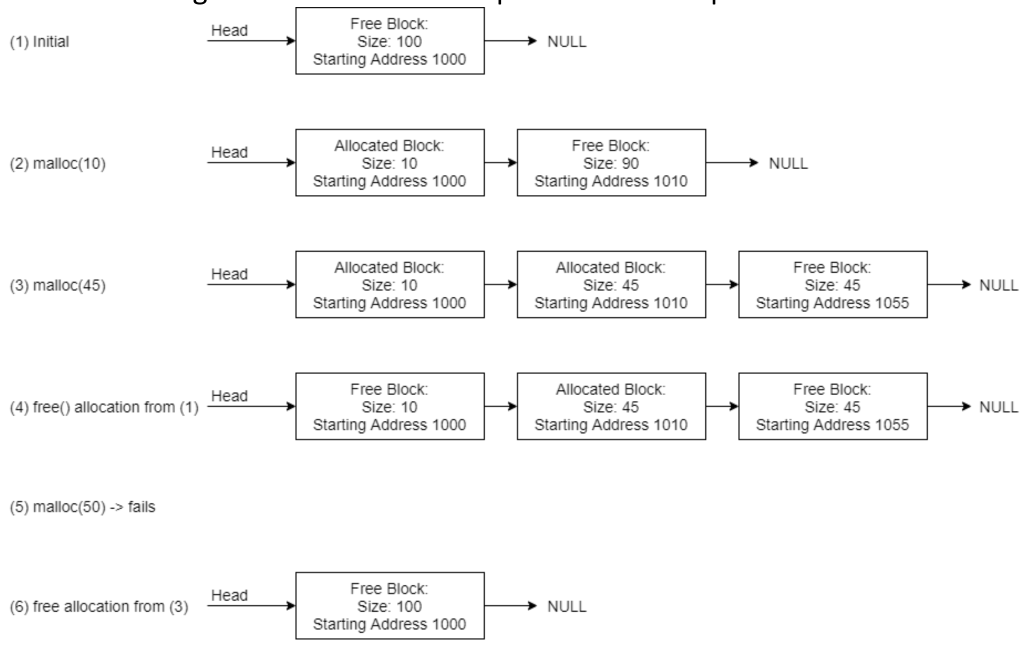
Project Description

You must implement a set of functions that a program can call to access the memory manager used to allocate and free storage. In addition, you will need to keep track of allocated and free blocks of memory.

There are several ways this can be done; one way is to use a linked list. Elements in the list record, the size of the allocated and free blocks and the memory address at the start of the block. Calls to allocate and free storage modify the list based on the request size.

For example:
1. Consider a heap of 100 bytes that starts at address 1000. Initially all heap memory is unallocated (free). So, there is 1 large free block of storage that is 100 bytes long. The list consists of one element.
2. A call to **malloc** for 10 bytes modifies the free block to reduce it by 10 bytes and allocates memory to the process. The free block now has only 90 free bytes. In addition, an element for the allocated block is added to the list.
3. Another call to **malloc**, this time for 45 bytes modifies the free block to reduce it by 45 bytes (leaving 45 left). The list would not have 3 elements: 1 for the first allocated block, 1 for the second allocated block, and one for the free block.
4. A call to free the 10-byte block would convert the allocated block into a free block. A memory allocation request for 10 bytes or less could be satisfied by that free block.
5. However, a call to **malloc** 50 bytes at this point would fail because there are no free blocks big enough to satisfy the request.
6. If a call to free is made to free the 45-byte block allocated in step 3, now there are free blocks next to each other. These should be coalesced (combined) into a single free block

Here's what the list might look like at each step from the example:


(1) Initial
Head
Free Block: Size: 100 Starting Address 1000
NULL
(2) malloc(10)
Head
Allocated Block: Size: 10 Starting Address 1000
Free Block: Size: 90 Starting Address 1010
NULL
(3) malloc(45)
Head
Allocated Block: Size: 10 Starting Address 1000
Allocated Block: Size: 45 Starting Address 1010
Free Block: Size: 45 Starting Address 1055
NULL
(4) free() allocation from (1)
Head
Free Block: Size: 10 Starting Address 1000
Allocated Block: Size: 45 Starting Address 1010
Free Block: Size: 45 Starting Address 1055
NULL
(5) malloc(50) -> fails
(6) free allocation from (3)
Head
Free Block: Size: 100 Starting Address 1000
NULL

Allocating from a free block essentially splits the block into two: One block for the allocated space and one block for the remaining free space.

NOTE: This splitting is correct except for cases where you have a free block that exactly matches the allocation request. You should not have any zero sized blocks in your data structure. If you find a free block that exactly matches the size that you need for an allocation, then there should be no free blocks after it.

As with **malloc** and free, the caller should not be aware of the mechanism that is being used for memory block record keeping. Whatever structure you choose to use should be hidden in your memory manager implementation. This can be done by creating a global structure in your memory manager implementation file (c source file). The internal implementation of the data structure that you use to keep track of blocks is up to you. Internally to the memory manager, you are free to use **malloc** and free (the system versions). This will simulate the overhead that the operating system uses to manage memory. Just make sure you have no memory leaks within your memory manager.

Normally the size of the heap is controlled using system calls to request modifications to the process address space. Your memory manager will work a little bit differently. A user of your memory manager will be required to pass a pointer to a location (pointer) in memory and a size. This storage must be pre-allocated by the user.

Your implementation must support 3 different types of placement algorithms to satisfy allocation requests:

1. First fit – allocates from the first free block that is big enough
2. Worst fit – allocates from the largest free block
3. Best fit – allocates from the smallest free block that is large enough to satisfy the request

A summary of all required functions (including statistics functions) is below:

- void mm_init(void* start, int size) – Initialize the memory manager
    - NOTE: The **mm_init** function is used to manage an **ALREADY ALLOCATED** space.
    - See the sample **testmemmgr.c** for how the **mm_init** function is used.
- void mm_destroy() – cleans up any storage used by the memory manager
- void* mymalloc_ff(int nbytes) – Requests a block of memory be allocated using first fit placement algorithm
- void* mymalloc_wf(int nbytes) – Requests a block of memory be allocated using worst fit placement algorithm
- void* mymalloc_bf(int nbytes) – Requests a block of memory be allocated using best fit placement algorithm
- void myfree(void* ptr) – Free the storage address by ptr
- int get_allocated_space() – Returns the amount of currently allocated space in bytes

- int get_remaining_space() – Returns the amount of current free space in bytes (sum of all free blocks)
- int get_fragment_count() – Returns the number of free blocks (the count of all the blocks, not the size)
- int get_mymalloc_count() – Returns the number of successful malloc's made (since the last mm_init)

Error Checking

Your memory manager can't trust the user, so it must perform error checking. If the user tries to **malloc** storage that can't be satisfied, return a NULL pointer just like the system **malloc**.

Furthermore, if a user tries to free a block that isn't allocated this should cause a segmentation fault. In this case, your memory manager should send SIGSEGV to the process (see the man page for the **kill** system call in section 2):

```
man 2 kill
```

Sample Usage

Here is a sample client program that uses the memory manager. This file is provided along with the starter code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "memory_manager.h"

#define MY_HEAP_SIZE 100

int main()
{
   char my_heap[MY_HEAP_SIZE];
   mm_init(my_heap, MY_HEAP_SIZE);

   printf("1 -- Available Memory: %d, Fragment Count: %d\n",
      get_remaining_space(), get_fragment_count());

   // Allocate 10 bytes
   //   shouldn't fail
   char* ptr1 = mymalloc_ff(10);
   if(ptr1 == NULL) {
      printf("ptr1 - mymalloc_ff(10) failed\n");
      exit(EXIT_FAILURE);
   }

   strncpy(ptr1, "HELLO", 10);
   printf("ptr1 is %s\n", ptr1);

   printf("2 -- Available Memory: %d, Fragment Count: %d\n",
      get_remaining_space(), get_fragment_count());

   // Allocate 45 bytes
   //   shouldn't fail
   char* ptr2 = mymalloc_wf(45);
   if(ptr2 == NULL) {
      printf("ptr2 - mymalloc_ff(45) failed\n");
      exit(EXIT_FAILURE);
   }

   strncpy(ptr2, "GOODBYE", 45);
   printf("ptr2 is %s\n", ptr2);

   printf("3 -- Available Memory: %d, Fragment Count: %d\n",
      get_remaining_space(), get_fragment_count());

   // Attempt to allocate 50 bytes
   //   should fail
   char* ptr3 = mymalloc_bf(50);
   if(ptr3 == NULL) {
      printf("ptr3 - mymalloc_bf(50) failed\n");
   }

   printf("4 -- Available Memory: %d, Fragment Count: %d\n",
      get_remaining_space(), get_fragment_count());

   // Free the first two pointers
   myfree(ptr1);
   myfree(ptr2);

   printf("5 -- Available Memory: %d, Fragment Count: %d\n",
      get_remaining_space(), get_fragment_count());

   printf("Total successful mallocs: %d\n", get_mymalloc_count());

   // Double free, should cause a segmentation fault
```

```
   myfree(ptr2);

   mm_destroy();

   return 0;
}
```

This should produce the output:

```
1 -- Available Memory: 100, Fragment Count: 1
ptr1 is HELLO
2 -- Available Memory: 90, Fragment Count: 1
ptr2 is GOODBYE
3 -- Available Memory: 45, Fragment Count: 1
ptr3 - mymalloc_bf(50) failed
4 -- Available Memory: 45, Fragment Count: 1
5 -- Available Memory: 100, Fragment Count: 1
Total successful mallocs: 2
Segmentation fault (core dumped)
```

Thread Safety

Your memory manager should be thread safe. In other words, all calls to any memory manager function must work correctly with a multithreaded program. Race conditions might exist if two threads attempt to allocate or free memory at the same time. Furthermore, depending on how you implement your statistics, there might be race conditions in those functions as well. Make sure you use concurrency mechanisms (semaphores, mutex locks, condition variables, etc.) to ensure that your block allocation data structure is never corrupted.

NOTE: you can assume mmInit() and mmDestroy() will only ever be called by a single thread. Mutual exclusion is not needed for those functions.

A sample test driver using pthreads is included with the starter code.

Additional Development Requirements

- Do **NOT** include source files inside each other via #include. The #include preprocessor macro should only be used for header files
- You must put all source code in a src directory
- You must submit a make file in your src directory that will build ALL executables by just typing make on the command line.

Code Structure

Code must follow style guidelines as defined in the course material.

Getting Started

The following files have been provided for you. They are saved in Canvas and may be downloaded:

- memory_manager.h – the header file containing the declarations for the memory manager functions
- memory_manager.c – the file containing the implementation for the memory manager functions. This has been started for you but contains, for the most part, empty functions.
- testmemmgr.c – a sample tester file (you may expand this file as needed)
- pthread_testmemmgr.c – a sample tester file using pthreads
- Makefile – a makefile that builds the given tester files and the memory manager

Feel free to create additional functions and files as needed to run your simulation. Consider dividing up code into separate files. Use header files as needed. Do **NOT** forget to use include guards in your header files.

At the top of **EACH SOURCE FILE** include a comment block with your name, assignment name, and section number.

SUBMISSION REQUIREMENT: You must submit with your source code a Makefile which can be used to build your memory manager with test cases (see Testing and Debugging). It should create a separate executable for each test. Your programs must compile without error AND without warnings.

Hints and Tips

You'll need to come up with an efficient data structure to use. If you use the system malloc, make sure you don't have memory leaks. An OS should NEVER have memory leaks. Don't forget to free your mallocs!
You need to ensure thread safety so synchronize access to your memory manager.

Test Cases

Use the included test drivers to get you started with testing your assignment. You are also required to create and submit your own test driver(s). Don't forget to test single threaded and multithreaded programs as well as each of the placement algorithms. Include your test driver(s) in your submission. Don't forget to test large and small amounts of memory (both for the managed space and for requests).

Debugging

See the course debugging tips for using gdb and valgrind to help with debugging.

Your program must be free of run-time errors and memory leaks.

Deliverables

When you are ready to submit your assignment:

- Make sure your name, assignment name, and section number are contained in all files in your submission - in comment block of source file(s)
- Make sure all source and header files are in your src directory **ONLY**. All your files should be zipped and uploaded to Canvas.
- Make sure you cite your sources.
- Make sure your assignment code is commented thoroughly and follows the coding standard.
- Include in your submission, a set of suggestions for improvement, what you enjoyed about this assignment, and a reflection about what the code is doing and screen shots from running your code.
- Make sure all files and supporting documentation are uploaded to Canvas.

Additional Submission Notes

If/when using resources from material outside what was presented in class (e.g., Google search, Stack Overflow, etc.) document the resource used in your submission. Include exact URLs for web pages where appropriate.

NOTE: Sources that are not original research and/or unreliable sources are not to be used. For example:

- Wikipedia is not a reliable source, nor does it present original research: https://en.wikipedia.org/wiki/Wikipedia:Wikipedia_is_not_a_reliable_source
- ChatGPT is not a reliable source: https://thecodebytes.com/is-chatgpt-reliable-heres-why-its-not/

You should ensure that your program compiles without warnings (using -Wall) or errors prior to submitting. Make sure your code is well documented (inline comments for complex code as well as comment blocks on functions). Make sure you check your code against the style guidelines.

To submit, upload a zipped folder of all materials to Canvas.

Extra Credit

While dynamic memory allocation (e.g. the use of malloc) within the OS kernel is possible, it is often advantageous to limit the use of this as much as possible to avoid any potential for memory leaks within the kernel.

For example, internally in the Linux kernel kmalloc() is used
https://www.kernel.org/doc/htmldocs/kernel-api/API-kmalloc.html
as a way to dynamically allocate blocks within the OS kernel's address space.

As a bonus challenge, can you find a way to implement your bookkeeping data structure so that it does not need to use malloc and free internally?

- It must still be able to manage a region of memory that is of varying size. The memory manager doesn't know how big the region is until the call to mmInit is made.
- It must still be able to keep track of a potentially unlimited number of memory fragments and allocated blocks.
- must still be thread safe
- It must keep track of all required statistics

HINT: You can use a portion of the memory region given to you by the user. This will reduce the total amount of storage that can be allocated from the block, but that is fine provided you can avoid using malloc and free internally.

Grading Criteria

- (5 Points) Submitted files follow submission guidelines
  - Only the requested files were submitted
  - Files are contain name, assignment, section
  - Sources outside of course material are cited
- (25 Points) Suggestions and Reflection of assignment
  - List of suggestions for improvement and/or what you enjoyed about this assignment
  - Reflection of what the code is doing, why it works this way, and screen shots of your results
- (10 Points) Code standard
  - Code compiles without errors or warnings
  - Code is formatted and commented per the course coding standard
- (5 Points) Memory management
  - Code contains no memory leaks
- (15 Points) Test Cases - Thoroughness of submitted test cases
- (40 Points) Instructor Tests - Implementation passes all instructor test cases
- (15 Points) Bonus - Implementation of the Extra Credit Challenge