p3: Dynamic Memory Allocator

Due Mar 13 by 10pm **Points** 120 **Submitting** a file upload **Available** until Mar 17 at 10pm

This assignment was locked Mar 17 at 10pm.

Learning GOALS

The purpose of this program is to help you understand the nuances of building a heap memory allocator, to further increase your C programming skills by working more with pointers, and to start using Makefiles and the gdb debugger.

OVERVIEW

For this assignment, you will be given the structure for a simple shared library that is used in place of the heap memory allocation functions **`malloc()`** and **`free()`**. You'll code two functions in this library, named **`allocHeap()`** and **`freeHeap()`**.

BACKGROUND

Heap memory allocators have two distinct tasks. First, the memory allocator uses the `sbrk()` system call to ask the operating system to allocate the heap segment of its virtual address space. However, in the code we've provided, we use `mmap()` to simulate a process's heap in the memory-mapped segment. Second, the memory allocator manages this memory by maintaining the internal structure of the heap including tracking the size and status of each block. When the process makes a request for heap memory, the allocator searches its list of heap blocks for a free block that is large enough to satisfy the request. The chosen block might be split into two smaller ones with the first part having its status set to allocated and the second part being free. Later when process frees the heap memory, the allocator changes that allocated block's status to freed and checks to see if it can coalesce it with free neighbors to make larger blocks.

This heap memory allocator is usually provided as part of a standard library rather than being part of the operating system. Thus, the memory allocator operates entirely within the virtual address space of a single process and knows nothing about which physical memory pages have been allocated to this process or the mapping from virtual addresses to physical addresses.

The C programming language defines its allocator with the functions `malloc()` and `free()` found in "stdlib.h" as follows:

- `void *malloc(size_t s)`: allocates `s` bytes and returns a pointer to the allocated memory. The memory is not cleared.
- `void free(void *ptr)`: frees the memory pointed to by `ptr` that was previously allocated by `malloc()` (or `calloc()` or `realloc()`). The behavior is undefined if `ptr` is a stray pointer or if an attempt is made to free an allocation more than once. If `ptr` is `NULL`, the function does nothing and simply returns.

UNDERSTAND

First, you'll need to understand the project code that we've provided. Copy the entire contents from the following directory into your working directory for this assignment:

```
/p/course/cs354-skrentny/public/code/p3
```

Make sure your directory layout is the same as we've provided. In the p3 directory, you'll find the files named "Makefile", "heapAlloc.c" and "heapAlloc.h" as well as a subdirectory named "tests". Make sure the copy you make also has a subdirectory named "tests".

In "heapAlloc.c" is the completed code for the functions: `initHeap(int sizeOfRegion)` and `dumpMem()`. Look at what these functions do and how they do it. Also note the global variable `heapStart`, which is a pointer to the header of the first memory block. You do not want to change this variable or you will lose the address to access to your heap. The functions we've completed are:

`initHeap(int sizeOfRegion)`

This function initializes the "heap" space that the allocator will manage. This function should be called **once at the start of any main program** before calling any of the other allocator functions. The test main programs we've provided (discussed below) already call `initHeap`.

For improved performance, `initHeap(int sizeOfRegion)` rounds up the amount memory requested to the nearest page so it is possible that more memory might be allocated than originally specified by `sizeOfRegion`. This memory is initialized as a single free block followed by an end mark for the heap space. `heapStart` will be pointing to that free block's header, which is the beginning of the implicit free list that your allocator uses to allocate blocks via `allocHeap()` calls.

`dumpMem()`

This function prints the list of all the memory blocks (both free and allocated). **Use this to debug and determine if your code works properly by adding calls to this function to our test programs as needed.** Implementing functions like this are very helpful and well worth your time when working on complex programs. It produces useful information about the heap structure but it only displays information stored in the block headers. It is recommended that you extend the implementation to display the information stored in the footers of the free blocks too. No points will be deducted if you chose not to.

SPECIFICATIONS

Note: Do not change the interface. Do not change anything within the file "heapAlloc.h". Do not change any part of the function `initHeap()`.

You'll be coding the project in 2 parts, but you'll use the same source file "**heapAlloc.c**" to code both the parts. You should finish Part A before implementing Part B. Your implementation can be tested using the test main programs we've provided in the "tests" subdirectory. These tests call your implementation of `allocHeap()` and `freeHeap()` functions as explained further in the **Test the Code** section below.

PART A: Memory Allocation

Write the code to implement `allocHeap()`. Use the **next-fit placement policy** when allocating blocks and **split the block** if possible. Recall that `heapStart` is the beginning of the implicit free list structure as defined and described in the file "**heapAlloc.c**". This function uses headers with size and status information (p and a bits) and free block footers for the heap structure as was discussed in lecture. The function `allocHeap()` is described below:

`void *allocHeap(int size);`

This function is similar to the C library function `malloc()`. It returns a pointer to the start of the allocated payload of `size` bytes. The function returns `NULL` if there isn't a free block large enough to satisfy the request. Note, additional heap memory will not be requested from the OS than what was originally allocated by `initHeap()`. `allocHeap()` returns **double-word (8 bytes) aligned** chunks of memory to improve performance. The total size of the allocated block including header must be a multiple of 8. For example, a request for 8 bytes of memory uses 16 bytes, which is divided into 8 bytes for the payload plus 4 bytes for the header, and an additional 4 bytes for padding to achieve double word alignment. To verify that you're returning 8-byte aligned pointers, you can print the pointer in hexadecimal this way:

```
printf("%08x", (unsigned int)(ptr));
```

The last hexadecimal digit displayed should be a multiple of 8 (that is 0, or 8). For example, 0xb7b2c048 is okay, and 0xb7b2c043 indicates a problem with your alignment. Here `ptr` is the pointer returned by `allocHeap()`.

Once the chosen free block is located we'll split the block in two (if possible) to minimize internal fragmentation. The first part becomes the allocated block, and the remainder becomes a new free block that should be at least 8 bytes in size. Note, only free blocks contain footers, and footers only store the block size. Splitting should always result in one allocated block followed by one free block that sum to the size of the original free block that was split.

You can verify if your implementation is working correctly using the tests we've provided in the "**tests**" subdirectory. Successfully passing each test earns 5 points additional points towards your program's score. These allocation tests are not exhaustive, but they do help incrementally develop your code. They are arranged in the increasing order of difficulty:

- `alloc1`: a simple 8-byte allocation
- `alloc1_nospace`: allocation is too big to fit in available space3
- `writeable`: write to a chunk from Mem_Alloc and check the value
- `align1`: the first pointer returned is 8-byte aligned
- `alloc2`: a few allocations in multiples of 4 bytes
- `alloc2_nospace`: the second allocation is too big to fit
- `align2`: a few allocations in multiples of 4 bytes checked for alignment
- `alloc3`: many odd sized allocations
- `align3`: many odd sized allocations checked for alignment

A test fails with an error message "Unexpected error." followed by an aborted message. Note you will only receive credit for these tests if your code runs and passes them. Comment out buggy code for any tests that you haven't passed so that your program compiles, runs, and doesn't crash on your successful tests.

PART B: Memory Freeing

Write the code to implement `freeHeap()`. When freeing memory, use **immediate coalescing** with the adjacent free memory blocks. This function uses headers with size and status information and free block footers for the heap structur. The function `freeHeap()` is described below:

`int freeHeap(void *ptr):`

This function is similar to the C library function `free()`. It frees the block of heap memory containing `ptr`'s payload and returns 0 to indicate success. If `ptr` is `NULL`, `ptr` is not 8 byte aligned, not within the range of memory allocated by `initHeap()`, or points to a free block then this function just returns -1. When freeing a block you must coalesce it with its adjacent free blocks. Remember, this requires only one header and one footer in the coalesced free block, and you should not waste time clearing old headers, footers, or data.

You can verify if your implementation is working correctly using the tests we've provided in the "**tests**" subdirectory. Successfully passing each test earns 5 points additional points towards your program's score. These freeing tests are not exhaustive, but they do help incrementally develop your code. They are arranged in the increasing order of difficulty:

- `free1`: a few allocations in multiples of 4 bytes followed by frees
- `free2`: many odd sized allocations and interspersed frees
- `coalesce1`: check for coalescing free space
- `coalesce2`: check for coalescing free space
- `coalesce3`: check for coalescing free space
- `coalesce4`: check for coalescing free space
- `coalesce5`: check for coalescing free space (first chunk)
- `coalesce6`: check for coalescing free space (last chunk)

A test fails with an error message "Unexpected error." followed by an aborted message. Note you will only receive credit for these tests if your code runs and passes them. Comment out buggy code for any tests that you haven't passed so that your program compiles, runs, and doesn't crash on your successful tests.

TEST the Code

In the project directory that you copied, we've provided a file, named "**Makefile**", that is used to compile your code in "heapAlloc.c" into a shared library named "heaplib.so". While you are working in this project directory, enter the command `make` on the Linux command line to make this shared library.

To test if your allocator works properly, our test programs in the "tests" subdirectory will also need to be compiled. Change your working directory to the "**tests**" subdirectory. There is a second "**Makefile**" in this subdirectory that is used to compile our tests by entering `make` on the Linux command line. This will make executables for all the test programs in this directory and link them to the shared library that you've compiled beforehand.

Please note that the tests we've provided are not exhaustive. They cover a good range of test cases, but there will be additional tests that we'll use to grade your code. You can create your own test program linked to the shared library by using a similar compiler command as in the second "Makefile".

Summary: After you've made code changes to "heapAlloc.c"

1. Make sure your working directory is the p3 project directory and enter the command `make`.
2. Change directory to the "tests" subdirectory and enter the command `make` a second time.
3. Run the tests in that subdirectory.
4. Change directory back to the p3 project directory to continue to incrementally develop your code in "heapAlloc.c".

HINTS

- **Remember to add the size of the block header** when determining the block size. Use of `sizeof(blockHeader)` to get that size rather than using 4 bytes.
- **Double check your pointer arithmetic's automatic scaling**. `int*` would increment the address by 4 bytes. Cast your heap block pointers to `void*` or `char*` to set the scale factor to 1. What scale factor is used for `blockHeader*`?
- You might want to use '`(size_status & 1) == 0`' to check if a block is allocated or not. Note the parentheses are required.
- Check return values for your function calls to make sure you're not getting unexpected behavior.
- You may introduce global variables in "heapAlloc.c" as needed for your allocator functions.
- For your own testing main programs, make sure you call `initHeap()` first to allocate the initial space in the memory-mapped segment being used to simulate the heap segment.

GDB for Debugging C

The GNU debugger, gdb, is a powerful tool for resolving errors in C code. To run **gdb** on a particular program named **a.out** enter at the command prompt (we'll assume is $) in the directory where a.out resides:

```
$gdb a.out
```

which launches the debugger and loads the a.out program. The command **run** causes the debugger to run the program. For help finding segmentation faults in p3, start gdb on the test that causes the segmentation fault and when in gdb enter the command run. This will show you where the segmentation fault occurs in your code (or possibly in library functions that are incorrectly used).

You can use gdb to set breakpoints in your code to stop execution so that you can take control of the debugging session. For example, a common thing to do before running in gdb is:

```
(gdb) break main
```

to set a breakpoint at the **main()** function of the program, and then type:

```
(gdb) run
```

to run the program. When the debugger enters the **main()**, it will stop running the program and pass control back to you.

You will need to learn some basic commands in gdb to set breakpoints in your code, step through instructions, and examine the contents of variables. Below are some commands to get you started, but you'll need to read up more about gdb on your own to explore these and other commands fully.

- **break <location>**: sets up a breakpoint at the location which can be a function name or a line.
- **continue**: resumes the execution

- **stepi**: steps through the code one instruction at a time

Hint: Using the up and down arrows on the keyboard (or ctrl-p and ctrl-n for previous and next, respectively) allows you to go through your gdb history and easily re-execute old commands.

Here are a few good tutorials and resources online to get started with gdb:

- Basic gdb **example.** (http://www.cprogramming.com/gdb.html)
- A nice **introduction.** (https://www.youtube.com/watch?v=sCtY--xRUyI)

(https://www.youtube.com/watch?v=sCtY--xRUyI)
for those who like videos
- Handy gdb **cheatsheet.**

REQUIREMENTS

- **Do *not* use any of C's allocate or free functions in this program!** You will not receive credit for this assignment if you do since that simply avoids the purpose of this program.
- Your program must follow style guidelines as given in the **Style Guide**.
- Your program must follow commenting guidelines as given in the **Commenting Guide**. Keep the function header comments we've put in the skeleton code.
- Your programs must operate exactly as specified.
- We will compile your programs with **gcc -Wall -m32 -std=gnu99** on the Linux lab machines. So your program must compile there, and without warnings or errors.

SUBMITTING Your Work

Leave plenty of time before the deadline to complete the two steps for submission found below. Given the unprecedented suspension of in-person instruction to help mitigate the spread of COVID-19, the grace period for this assignment has been extended to 10:00 pm on 3/17. Submitting during this grace period results in your submission being marked late but it will be accepted for grading without penalty. No submissions or updates to submissions are accepted after this grace period.

1.) Submit only the file listed below under Project p3 in Assignments on Canvas. Do not zip, compress, or submit your file in a folder.

- **heapAlloc.c**

Repeated Submission: You may resubmit your work repeatedly so we strongly encourage you to use Canvas to store a backup of your current work. If you resubmit, Canvas will modify your file names by appending a hyphen and a number (e.g., heapAlloc-1.c).

2.) Verify your submission to ensure it is complete and correct. If not, resubmit all of your work rather than updating just some of the files.

- **Make sure you have submitted all the files listed above.** Forgetting to submit or not submitting one or more of the listed files will result in you losing credit for the assignment.
- **Make sure the files that you have submitted have the correct contents.** Submitting the wrong version of your files, empty files, skeleton files, executable files, corrupted files, or other wrong files will result in you losing credit for the assignment.
- **Make sure your file names exactly match those listed above.** If you resubmit your work, Canvas will modify your file names as mentioned in **Repeated Submission** above. These Canvas modified names are accepted for grading.

Project p3 (1)

Criteria	Ratings		Pts
1. Compiles without warnings or errors	**9.0 pts** **No warnings or errors**	**0.0 pts** **Any errors or at least 5 warnings**	9.0 pts
2. Follows commenting and style guidelines	**6.0 to >0.0 pts** **Full Marks**	**0.0 pts** **No Marks**	6.0 pts
Execution test (provided): alloc1	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): alloc1_nospace	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): writeable	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): align1	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): alloc2	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): alloc2_nospace	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): align2	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): alloc3	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): align3	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): free1	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): free2	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): coalesce1	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts

Criteria	Ratings		Pts
Execution test (provided): coalesce2	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): coalesce3	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): coalesce4	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): coalesce5	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test (provided): coalesce6	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test: nextfit Checks using next-fit instead of best-fit or first-fit	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test: sanity Checks invalid input for allocHeap() and freeHeap()	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test: bigtest	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test: full	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
			Total Points: 120.0