p3: myHeap Dynamic Memory Allocator

Due Oct 28 by 11:59pm **Points** 120 **Submitting** a file upload **Available** Oct 11 at 12am - Oct 29 at 11:59pm

Announcements:

- 10/12 FIXED file name in assignment from p3heap.c to **p3Heap.c** to be consistent with actual file name in provide code.
- 10/11 Released - Get Started Today by reading the assignment and drawing heap blocks with their **size_status** value.

Learning GOALS

The purpose of this program is to help you understand the nuances of building a heap "dynamic" memory allocator, to further increase your C programming skills by working more with pointers, bit-masking, shift operators, and to start using **Makefile**s 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 **balloc()** and **bfree()**.

Please note that this project is worth 120 points which makes it count more (2x earlier projects) towards your projects portion of your final grade. Please do not wait to start as you will want time to consider each function and how best to implement.   All material needed for this project has been presented as of Lecture 11 (Free Block Splitting and Coalescing).

Students caught using coded solutions from previous students, or the internet, will not earn any credit for this assignment and will be reported to the college deans. You can do this work, we can help, but waiting or using someone else's solution will keep you from solving all the problems presented.

BACKGROUND

Heap memory allocators have two distinct tasks.

1. 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.
2. Second, the memory allocator manages this memory by maintaining the internal structure of the heap including tracking the size and status of each *heap 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 sets the Free block footer.

If *immediate coalescing* is used, it would also check to see if the newly freed block can be coalesced with any adjacent free neighbors to make larger blocks. For this assignment, you will NOT implement immediate coalescing rather you will implement a type of *delayed coalescing*. For the coalescing of this program, you will coalesce adjacent free blocks only if a request to coalesce is made by the process. In that case, the entire heap will be searched an any adjacent free block will be coalesced into a single free block.

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. This allows us to focus on the more consistent virtual memory view of memory as a contiguous block.

The C programming language defines its allocator with the functions `malloc()` and `free()` which are 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:

```
cp -r /p/course/cs354-deppeler/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", "p3Heap.c" and "p3Heap.h" as well as a subdirectory named "tests". Make sure the copy you make also has a subdirectory named "tests".

In "p3Heap.c" is the completed code for the functions: `init_heap(int sizeOfRegion)` and `disp_heap()`. Look at what these functions do and how they do it. Also note the global variable `heap_start`, 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:

init_heap(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 in the **tests** directory (discussed below) already call `init_heap`.

For improved performance, `init_heap(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. `heap_start` 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 `balloc()` calls.

disp_heap()

This function prints the list of all the heap 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 "p3Heap.h". Do not change any part of the function `init_heap()` implementation.

You'll be coding the project in 2 parts, but you'll use the same source file "**p3Heap.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 `balloc()` and `bfree()` functions as explained further in the section below.

PART A: Memory Allocation

Write the code to implement `balloc()`. Use the **best-fit placement policy** when allocating blocks and **split the best-fit block** only if the best-fit free block is larger than the request and if it can be split such that its remaining free block still meets minimum block size. Recall that `heap_start` is the beginning of the implicit free list structure as defined and described in the file "**p3Heap.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 `balloc()` is described below:

> **void *balloc(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, for this project additional heap memory will <u>not</u> be requested from the OS than what was originally allocated by `init_heap()`. `balloc()` 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 `balloc()`.

Once the chosen **best-fit** free block is located, we'll split the free block in two if the remaining free block is also a double-word aligned block (multiple of 8). This split will help minimize internal fragmentation. The first part becomes the allocated block, and the remainder becomes a new free block that must 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 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:

- `test_alloc1`: a simple 8-byte allocation
- `test_alloc1_nospace`: allocation is too big to fit in available space
- `test_writeable`: write to a chunk from Mem_Alloc and check the value
- `test_align1`: the first pointer returned is 8-byte aligned
- `test_alloc2`: several allocations in multiples of 4 bytes
- `test_alloc2_nospace`: the last allocation is too big to fit
- `test_align2`: a few allocations in multiples of 4 bytes checked for alignment
- `test_alloc3`: many odd sized allocations
- `test_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 `bfree()`. When freeing memory for this program, do not use **immediate coalescing**. Instead, this project requires that you implement delayed coalescing (PART C).

    `int bfree(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 `init_heap()`, or points to a free block then this function just returns -1. When freeing a block you set the header and footer of the free block, and update the p-bit of the next heap block's header. CAUTION: Do not set the p-bit of the next heap block if the next heap block is the END MARK.

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:

- `test_free1`: a few allocations in multiples of 4 bytes followed by frees
- `test_free2`: many odd sized allocations and interspersed frees
- `test_fit1`: a few allocations in multiples of 4 bytes followed by frees followed by allocations to see if the best fit is found (may wish to add and review disp_heap results to confirm best fit is found)
- `test_fit1_nospace`: a few allocations followed by frees followed by allocations that will not fit until freeing is implemented.
- `test_fit2`: many odd sized allocations and interspersed frees followed by allocations to see if the best fit is found (may wish to add and review disp_heap results to confirm best fit is found)
- `test_fit2_nospace`: a few allocations followed by frees followed by allocations that will not fit until freeing is implemented.

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 C: Delayed Coalescing, only coalesce when coalesce() is called

Implement *delayed coalescing* where adjacent free blocks are coalesced into larger free blocks in a separate function only when the function is called. The user will call this function explicitly to coalesce adjacent free blocks into larger free blocks. The function should traverse the entire heap and coalesce all sets of adjacent free blocks into larger free blocks.

This will allow the user to call coalesce() if balloc() fails. They can test the result of calling coalesce, but they must explicitly call balloc() again if desired. This breakdown of the allocation procedure into separate functional steps in this way will make it easier to develop and test your work incrementally. The coalesce() function returns 0 if there were no adjacent free blocks coalesced and it returns any positive integer if there were any adjacent free blocks that were able to be coalesced into a larger free block.

    `int coalesce():`

REMINDER: Allocated blocks have headers with size and status information while free blocks have headers with size and status information and they have footers (blockHeader with free block size only). CAUTION: The following two functions have not yet been tested. If you suspect there are problems, be sure to write your own test functions for the coalesce operation and best-fit allocation policy, and report any suspected problems.

- `test_coalesce1`: a few allocations followed by frees followed by allocations that will not fit followed by coalescing followed by allocations that now fit
- `test_coalesce2`: a few allocations followed by frees followed by allocations that will not fit followed by coalescing that requires more than two adjacent blocks to be coalesced followed by allocations that now fit
- there are additional test_coalesce programs. Each one should be examined for correctness before being relied upon to detect errors in your heap implementation.

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 "p3Heap.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 "p3Heap.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 by typing the name of the test at the command, ie. ./test_alloc1
4. Change directory back to the p3 project directory to continue to incrementally develop your code in "p3Heap.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. This will protect your code should you decide to change the contents and size of a block header structure.
- **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 set a mask variable so you can use something like `((size_status & abit_mask) == 0)` to check if the a-bit 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 "p3Heap.c" as needed for your allocator functions, but not to replace the need for those functions.
- For your own testing "main" programs, make sure you call `init_heap()` first to allocate the initial space in the memory-mapped segment being used to simulate the heap segment.

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 -g -Wall -m32 -std=gnu99** on the Linux lab machines. So your program must compile there, and without warnings or errors.
- We will check your programs for leaks and other errors using **valgrind --leak-check=full --error-exitcode=1.** So your program must run without errors or leaks.

Project p3 sp21 (1)

Criteria	Ratings		Pts
1. Compiles without warnings or errors	**9 pts** No warnings or errors	**0 pts** Any errors or at least 5 warnings	9 pts
2. Follows commenting and style guidelines	**6 to >0.0 pts** Full Marks	**0 pts** No Marks	6 pts
Execution test (provided): test_alloc1	**5 pts** Full Marks	**0 pts** No Marks	5 pts
Execution test (provided): test_alloc1_nospace	**5 pts** Full Marks	**0 pts** No Marks	5 pts
Execution test (provided): test_writeable	**5 pts** Full Marks	**0 pts** No Marks	5 pts
Execution test (provided): test_align1	**5 pts** Full Marks	**0 pts** No Marks	5 pts
Execution test (provided): test_alloc2	**5 pts** Full Marks	**0 pts** No Marks	5 pts
Execution test (provided): test_alloc2_nospace	**5 pts** Full Marks	**0 pts** No Marks	5 pts
Execution test (provided): test_align2	**5 pts** Full Marks	**0 pts** No Marks	5 pts
Execution test (provided): test_alloc3	**5 pts** Full Marks	**0 pts** No Marks	5 pts
Execution test (provided): test_align3	**5 pts** Full Marks	**0 pts** No Marks	5 pts

Execution test (provided): test_free1	5 pts Full Marks	0 pts No Marks	5 pts
Execution test (provided): test_free2	5 pts Full Marks	0 pts No Marks	5 pts
Execution test (provided): test_fit1	5 pts Full Marks	0 pts No Marks	5 pts
Execution test (provided): test_fit1_nospace	5 pts Full Marks	0 pts No Marks	5 pts
Execution test (provided): test_fit2	5 pts Full Marks	0 pts No Marks	5 pts
Execution test (provided): test_fit2_nospace	5 pts Full Marks	0 pts No Marks	5 pts
Execution test (provided): test_coalesce1	5 pts Full Marks	0 pts No Marks	5 pts
Execution test (provided): test_coalesce2	5 pts Full Marks	0 pts No Marks	5 pts
Execution test: bestfit Checks using best-fit instead of next-fit or first-fit	5 pts Full Marks	0 pts No Marks	5 pts
Execution test: sanity Checks invalid input for myAlloc() and myFree()	5 pts Full Marks	0 pts No Marks	5 pts
Execution test: bigtest	5 pts Full Marks	0 pts No Marks	5 pts
Execution test: full	5 pts Full Marks	0 pts No Marks	5 pts

Total Points: 120