CS 354 Lab 6: Memory Allocation

In this lab, you will implement a malloc library.

IMPORTANT: You may work in teams of 2 students (and *no more than 2* students). You may decide to work individually but the grading criteria will be the same for all the teams regardless of whether the project is an individual project or a team project. If you work in a team, turnin your project from only one account and include a lab6-src/README file with the logins of the team members on two separate lines.

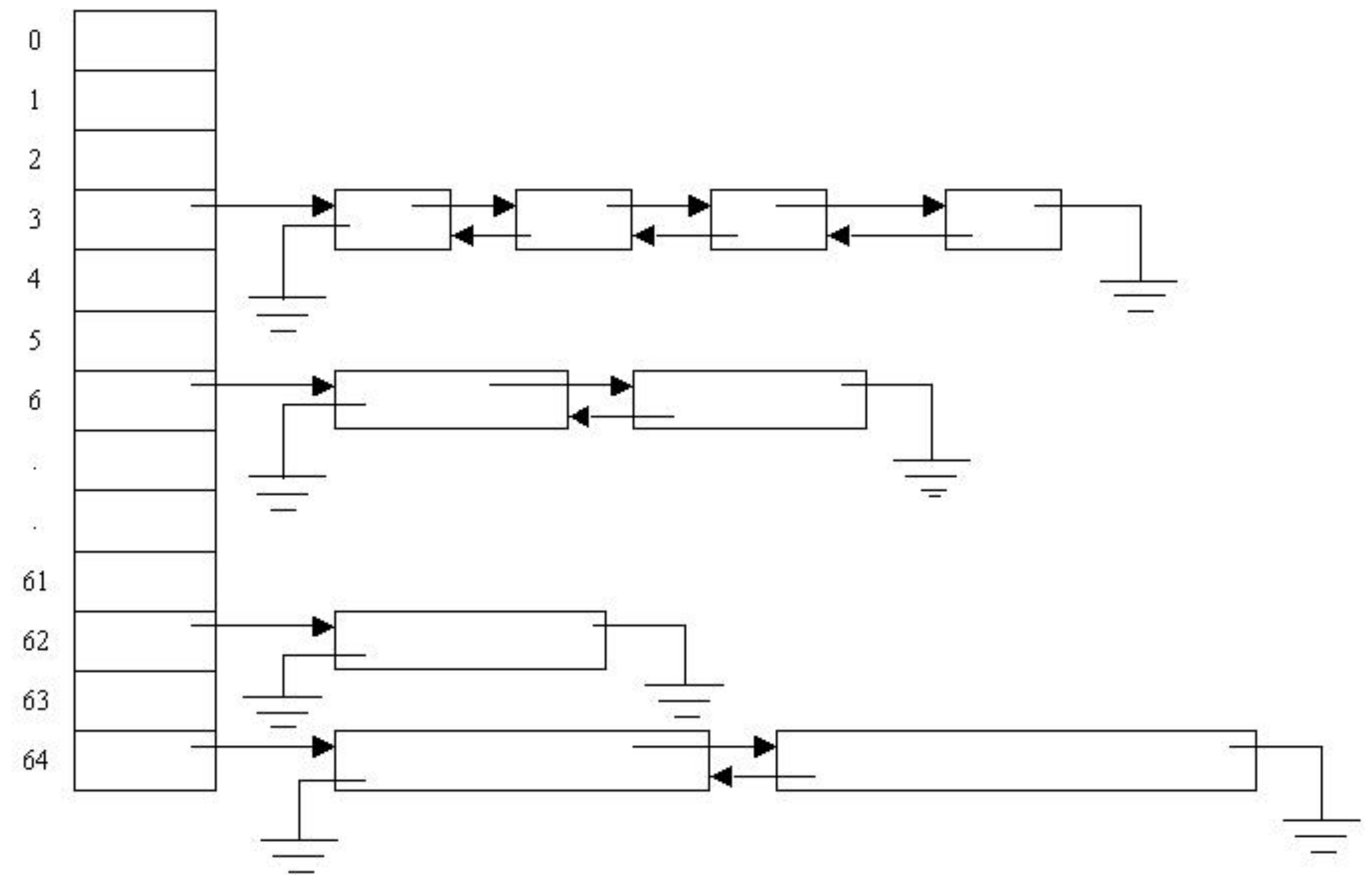
The Basic malloc Implementation

Download, uncompress, and untar the sources in lab6-src.tar.Z. This is the basic malloc implementation that you will use to start your project. Once you have untarred the sources, type *run-test*. You will see that 4 tests will run and will show some statistics. The tests are very inefficient because they use a system call every time memory is required. In addition, some tests use a lot of memory. The reason for these problems is because this basic allocator never frees memory. Every call to ***malloc()*** requests memory from the OS and the memory freed is never recycled.

Read the sources in ***MyMalloc.cc*** to become familiar with the functionality that is provided.

Implementing The allocator

You will add code to ***MyMalloc.cc*** to implement a **best fit** allocator using boundary tags and several free lists. The data structure that you will implement is a table of free lists. There will be 65 lists numbered 0 to 64. Lists 0 to 63 will have blocks of size 8 * i, where i is the number of the list (yes, the first 3 lists will always be empty). List 64 will contain blocks larger or equal than 64 * 8 = 512 and will be ordered by size in ascending order. Each list will be a doubly-linked list. Not all the lists may be populated at any given time.



Allocation

During allocation, your malloc implementation will do the following:

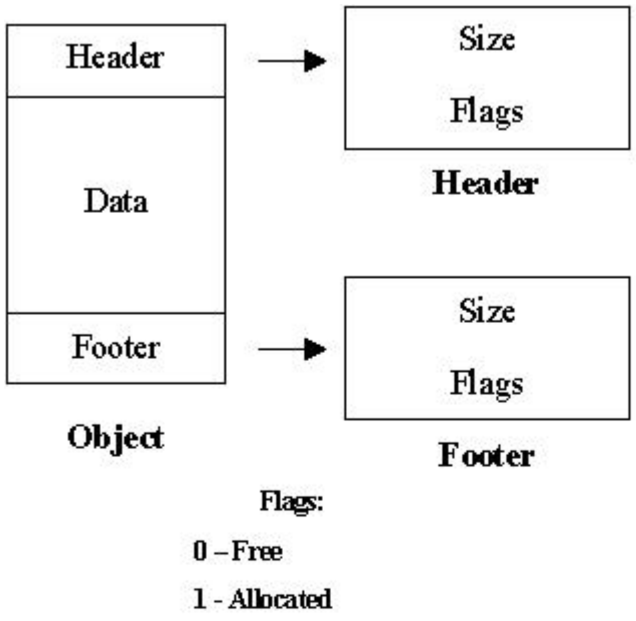
1. Round up requested size to the next 8 byte boundary.
2. Add the size of the header and the footer. The header and footer are explained later:
    - real_size = roundup8(requested size) + sizeof(header) + sizeof(footer)
3. Look up the corresponding list
    - list = real_size ≥ 512? 64:((real_size)>>3)
4. If the list is non-empty, remove a block from that list and return it.
5. If the list is empty, search for a list with objects of a larger size, going through lists from small to large. If a non-empty list with larger objects is found, remove one of the objects and split it. Use one portion of the object to satisfy the allocation request, and return the remainder to the appropriate list. If the remainder is smaller than or equal to the size of the header plus the footer, making the object unusable, then do not split the object and use the entire object to satisfy the allocation (creating internal fragmentation).
6. If there is not enough memory in the free lists to satisfy the allocation, then request memory from the OS by using the system call sbrk(). *If the request is smaller than 16 KB, then request 16 KB, and add the unused portion to the appropriate free list.*

It is important to make the critical sections in your code multi-thread safe.

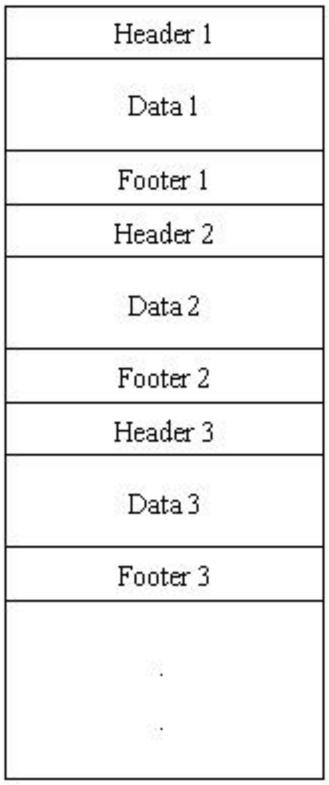
Use an appropriate synchronization mechanism, e.g., mutex_init, mutex_destroy, mutex_lock and mutex_unlock are provided by the thread library, and various mechanisms provided by the real-time library were used in Lab 4. You can check their man pages using the "man" command. ***You are NOT allowed to call malloc or new to allocate memory in your implementation.***

Headers and Footers

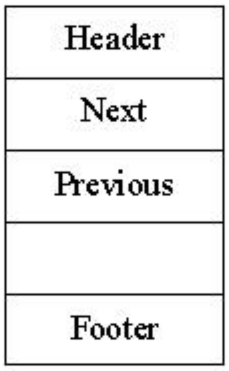
To make it easier to check an adjacent object, each allocated or free object will have a header and a footer that will store the size of the object and a flag to indicate if the object is allocated or free.



The size in the header of an allocated object will be used by free() to return the object to the corresponding list. Every object in the heap (whether allocated or free) will have a header and a footer and will look like this:



Besides the header and footer, a free object will also contain a next and previous pointer to the next and previous object in the free list. These previous and next pointers will be used to place the free object in the corresponding free list.



Free Object

Without Coalescing

You will find it easier to first implement and test the allocator without doing coalescing. Follow the steps described in "allocation" part above to implement the allocator. For the deallocation part, just return the object back to the corresponding free list without doing any coalescing. Test your implementation using the given tests, and write more tests on your own. **Make sure to finish this part before Thanksgiving so that you can finish the lab by the due date.**

Performing Coalescing

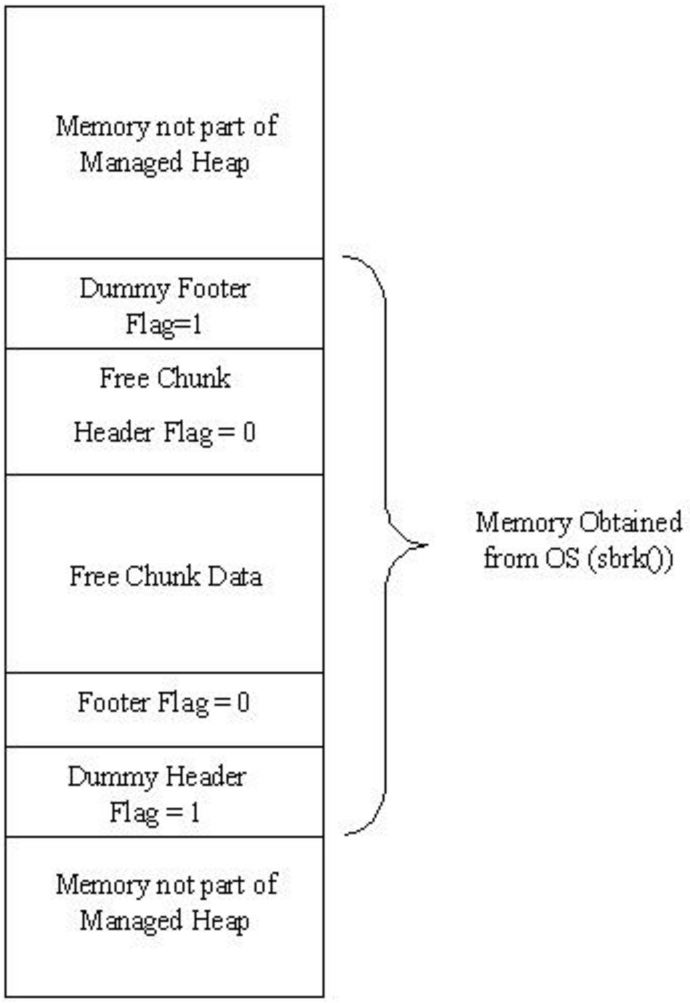
When an object is freed, your allocator will check if the neighboring objects before and after the object to be freed are free, and if either or both are free, coalesce the freed object with its neighbor(s) by doing the following:

1. Check the footer of the *left* neighbor object (the object just before the object being freed) to see if it is also free. If that is the case, remove the left neighbor from its free list using the previous and next pointers and coalesce this object with the object being freed.
2. Regardless of whether the object was coalesced with its left neighbor or not, check the header of the *right* neighbor object (the object just after the object being freed) to see if it is also free. If that is the case, remove the right neighbor from its free list using the previous and next pointers and coalesce it with the object being freed.
3. Place the freed object in the corresponding free list and update the header and footer.

Fence Posts

If the object being freed happens to be at the beginning of the heap, it is likely that the coalescing algorithm will try to coalesce the memory beyond the the beginning of the heap crashing your allocator. The same will happen if the object being freed is at the end of the heap and the coalescing algorithm tries to coalesce the memory beyond the end of the heap. Additionally, since your malloc library is not the only one that calls sbrk() but other libraries do, it will be possible that there are going to be areas in the heap that cannot be coalesced because they were not allocated by your malloc library.

To prevent these problems, every time a new free chunk of memory is requested from the OS, the allocator will add a "dummy footer" or "fence post" at the beginning of the chunk with the flag set to 1, meaning that the memory before this section of memory is not free and cannot be coalesced. Also, at the end of the chunk, add a "dummy header" or fence post with the flag set to 1 meaning that the memory beyond the chunk cannot be coalesced. That is in addition to the header and footer added to the chunk when it is placed in the free list.



IMPORTANT: If the memory returned by the previous sbrk() done by your allocator and the memory allocated by the new sbrk() call happen to be consecutive, then no fence posts will be necessary between the two, and you will have to coalesce the memory returned by sbrk with the free memory if possible.

Testing your Project

To test your implementation, first run the script ***run-test*** provided in the ***lab6-src*** directory. Note that this set of 4 tests is very simple and incomplete. Two additional simple tests are given in your lab6-src directory: test5.cc and test6.cc. Try these as well, and modify them to check other cases. In addition, you will have to write your own tests to check the following cases:

1. Malloc an object from a free list that is a precise size match
2. Malloc an object from a free list that needs to be split
3. Malloc an object from a free list that if split will give a remainder too small to be used in another allocation
4. Free an object that needs to be coalesced with the left neighbor only
5. Free an object that needs to be coalesced with the right neighbor only
6. Free an object that needs to be coalesced with both right and left neighbors
7. Free an object that does not need to be coalesced

8. Free an object at the beginning of the heap to test fence posts
9. Free an object at the end of the heap to test fence posts
10. Coalesce memory with memory obtained from a previous sbrk() call to see that the unnecessary fence post is being deleted
11. Check that freeing an already freed object would typically not crash
12. Check that freeing an object that was not allocated would typically not crash
13. Other interesting cases not listed in this list....

Turning In

The complete malloc implementation that includes coalescing is now due on **Tuesday, December 7th, 2004, at 11:59 pm.**

If you choose to work in a team, you must turnin your project from only one account and include a lab6-src/README text file with the mentor account logins of the two team members on 2 separate lines. The file should not contain anything else.

To submit:

1. Make sure to remove any debugging print statements that YOU have added.
2. Change directories to your `lab6-src` directory.
3. For team projects, check that there is a text file called README in the directory that only contains the 2 logins of the two team members on 2 separate lines.
4. Type '`make`' and ensure that everything built correctly.
5. Type '`make clean`' to clean up the object files, etc.
6. Use '`cd ..`' to change to the parent directory of your `lab6-src` subdirectory.
7. Type '`turnin -c cs354 -p lab6 lab6-src`' to submit your work for lab 6.
8. Type '`turnin -c cs354 -p lab6 -v`' and verify that the files you submitted are correct. Do not forget the -v. If you forget -v, your original submission will be corrupted.