CS 354 Lab 2

Implementing Malloc

FAQ

In this lab you will implement a malloc library.

Step 1 The Basic malloc Implementation

Download and untar the sources in lab2.tar.

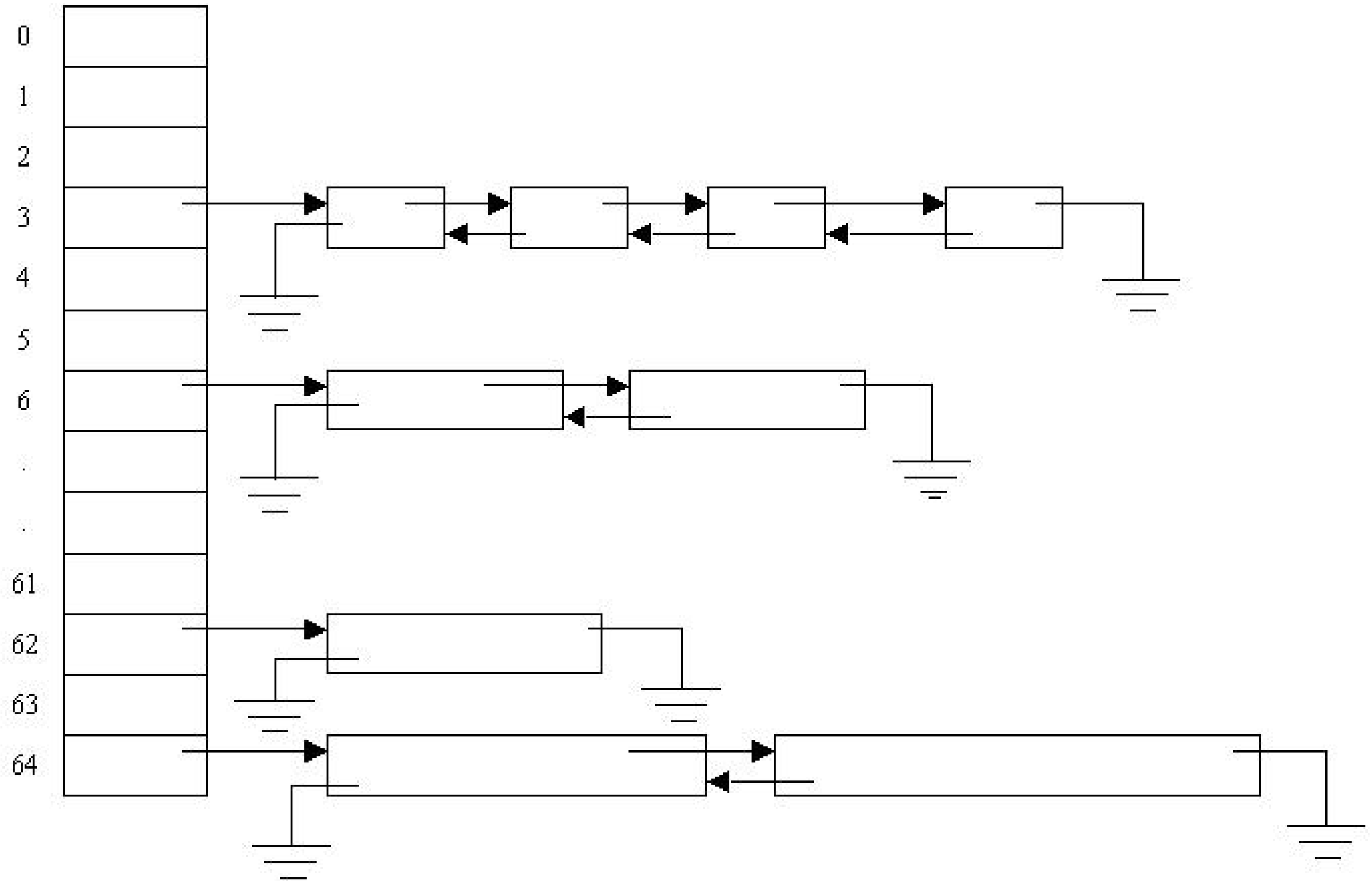
```
tar -xvf lab2.tar
```

This is the basic malloc implementation that you will use to start your project. Once that you have untared the sources, type *make*. You will see that the tests will run and will show some statistics. The tests pass but use a lot of memory. The reason 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 be familiar with the functionality that is provided.

Step 2 Implementing The allocator

You will add code to ***MyMalloc.cc*** to implement a best fit allocator using boundary tags and segregated free lists. The data structure that you will implement is a table of free lists. There will be 65 lists in total numbered 0 to 64. Lists 0 to 63 will have blocks of size 8 * i, where i is the number of the list. 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 double-linked list. Not all the lists may need to be populated at any given time.



b

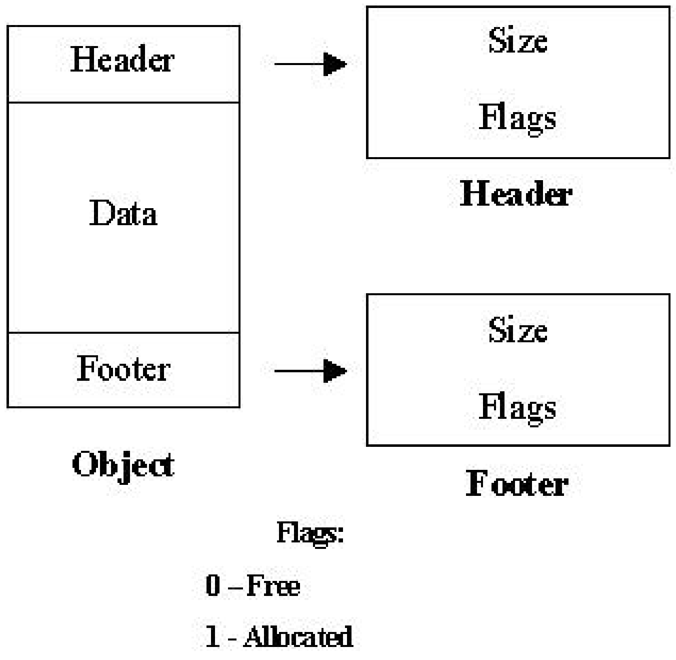
Allocation

During allocation your malloc implementation will do the following:

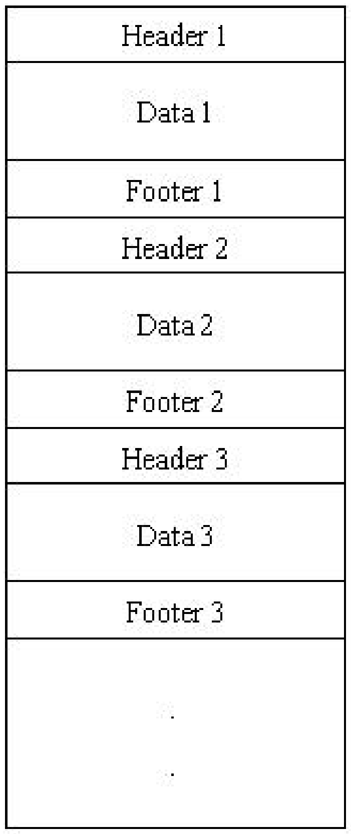
1. Round up requested size to the next 8 byte boundary.
2. Add the size of the header and the footer. The header and footer are explained later:
   - real_size = roundup8(requested size) + sizeof(header) + sizeof(footer)
3. Lookup the corresponding list
   - list = real_size ≥ 512? 64:((real_size)>>3)
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 larger size. If there is one non-empty list with larger objects, 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 corresponding list. If the remainder is smaller than 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.
6. If there is not enough memory in the free lists to satisfy the allocation, then request memory from the OS using sbrk(). If the request is smaller than 16KB then request 16KB.

Headers and Footers

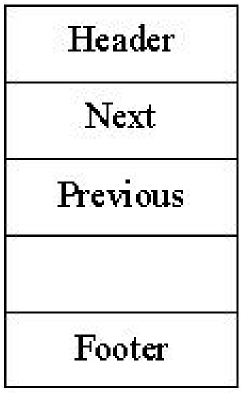
Each allocated or free object will have a header and a footer that will store the size of the object and a flag to tell if the object is allocated or free.



The size in the header will be used by free() to return the object to the corresponding list. Every object in the heap 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

Step 3 Part 1 turnin

You first will implement the allocator without doing coalescing. Follow the steps described in "allocation" part above to implement the allocator and for the deallocation part just return the object back to the corresponding free list without doing any coalescing. The given tests should pass. Also, write more tests on your own. The deadline for this part is September 18th at 11:59pm.

Type the following commands to turnin your first part of the project:

```
turnin -c cs354 -p lab2-1 lab2-src
```

Step 4 Doing Coalescing

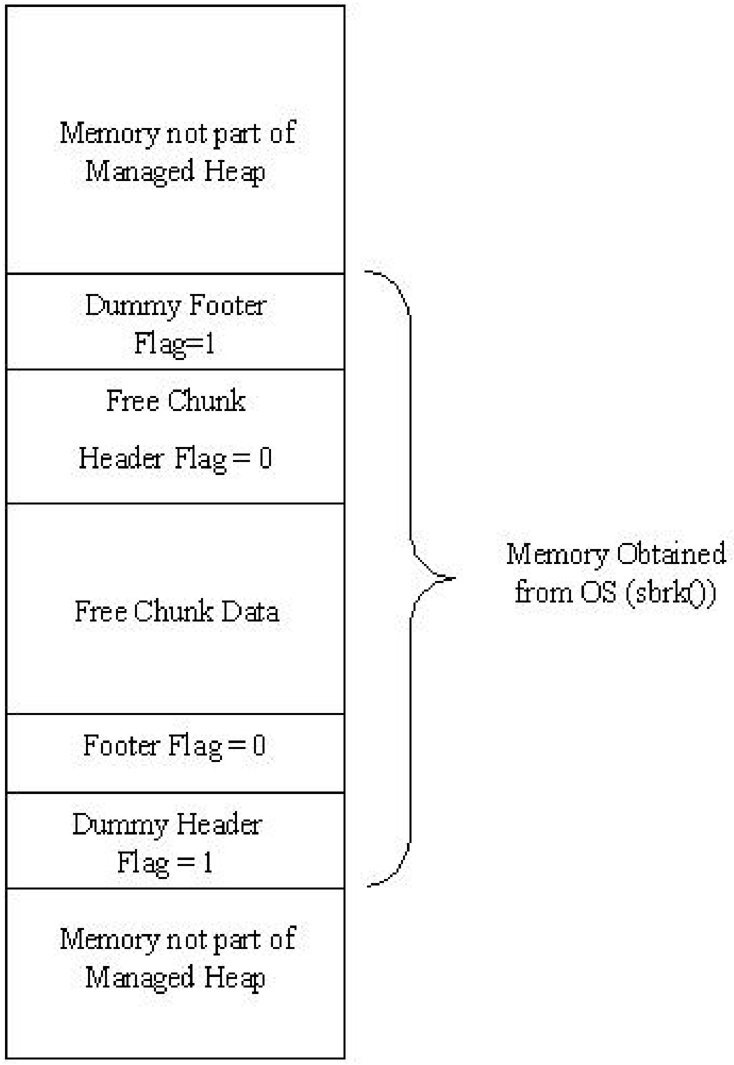
When an object is freed, your allocator will check if the neighbor objects before and after are free and if possible coalesce the freed object with its neighbors by doing the following:

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. 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() bit 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.
b
To prevent this problem, 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.


Memory not part of Managed Heap
Dummy Footer Flag=1
Free Chunk Header Flag = 0
Free Chunk Data
Footer Flag = 0
Dummy Header Flag = 1
Memory not part of Managed Heap
Memory Obtained from OS (sbrk())

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 and you will have to coalesce the memory returned by sbrk with the free memory if possible.

Testing your Project

To test your implementation you will do in step 2 run the script ***run-test*** provided in ***lab2-src.*** However, this set of tests is not complete. You willl have to write your own set of tests to cover the following:

1. Malloc an object from a free list that is a precise 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. See that memory is being reused
11. Free an already freed object should not crash
12. Free an object that was not allocated should not crash.
13. Other conditions not listed in this list.

Turning In

The complete malloc implementation that includes coalescing is due on October 9t at 11:59pm.
^M If you turnin your project on October 2nd before 11:59pm you will get 5/100 pts extra.

Add a README file to the lab2-src directory containing the following:

1. The list of problems that your implementation has including the parts you did not implement.

2. The list of tests programs you wrote and what parts of your implementation they test.

To turnin your project type the following from your mentor account:

```
turnin -c cs354 -p lab2-2 lab2-src
```