CS360 Malloc Lab

- James S. Plank
- CS360
- URL: http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/labs/Lab-7-Malloc/index.html

You may find the program mymalloc.c from Dr. Marz' second malloc lecture useful.

What you submit

You should submit mymalloc.c, which is described below.

Description

Your job in this lab is to write **mymalloc.c**, which implements the procedures defined in **mymalloc.h**:

```
#include <stdio.h>
#include <stdib.h>

void *my_malloc(size_t size);
void my_free(void *ptr);
void *free_list_begin();
void *free_list_next(void *node);
void coalesce_free_list();
```

my_malloc() and my_free() should work just like malloc() and free() as discussed in lecture. My_malloc() is a buffered interface to sbrk(), doling out heap memory for programs that call it. It does so by managing a free list of memory, using sbrk() to add memory to the free list. When a user calls my_malloc(s), you will return a pointer to at least s bytes of memory, aligned on an 8-byte quantity. As described in class, you will also reserve eight bytes before the pointer. In the first four of these bytes, you should store the size of the memory chunk allocated, which includes everything -- the user's memory, the bookkeeping bytes and any padding.

You may not call **sbrk()** with a value less than 8192. You should only call **sbrk()** with a value greater than 8192 if **my malloc()** was called with a value greater than 8184.

When the user calls **my_free(p)**, the chunk of memory should be returned to the free list. You do not have to coalesce free list entries when you call **my free()**.

Free_list_begin() returns a pointer to the first node on the free list. If the free list is empty, it should return NULL. The first four bytes on a free list node should contain the size of the node (including all metadata, like the size and pointers).

Free_list_next(n) takes a pointer to a free list node and returns a pointer to the next node on the free list. If n is the last node on the free list, then free list next(n) should return NULL.

Finally, **coalesce free list()** should process the free list and combine all adjacent entries.

Let's take some simple examples. First, suppose the user calls **my_malloc(9990)**. Your program will pad 9990 to a multiple of eight (9992) and add 8 bytes for bookkeeping. That's 10000 bytes. Since that is bigger than 8192, you'll call **sbrk(10000)**. Suppose that returns 0x10800. You will put the number 10000 at address 0x10800 and return 0x10808 to the user. There is no free list right now, so **Free_list_begin()** will return NULL.

Now suppose the user calls my_free(0x10808). Those 10000 bytes now turn into a free list node. The first four bytes remain 10000, but we'll modify some stuff to make the free list work. Free list begin() will return

0x10800 and Free list next(0x10800) will return NULL.

Suppose the user calls my_malloc(4992). That is a multiple of eight, so you will add eight bytes and carve 5000 bytes off your free list node. Personally, I'd carve it off the back of the free list node -- it's easier code to write. When this is done, the state of memory is as follows:

- The free list contains one node, which is 5000 bytes and starts at address 0x10800.
- The four bytes starting at address 0x10800 contain the number 5000.
- The my malloc() call returned 0x10800 + 5000 + 8 = 0x11b90.
- The size of the chunk is 5000 bytes, which means that the number at address 0x11b88 is 5000.
- Free list begin() will still return 0x10800.
- Free list next(0x10800) will return NULL.

Now, suppose the user calls **my_free(0x11b90)**. This will put that chunk of 5000 bytes on the free list, and the free list contains two nodes: 0x11b88 and 0x10800. Even though they are contiguous, we don't coalesce them during the **my_free()** call. **Free_list_begin()** will return either 0x11b88 or 0x10800, and **free list begin()** will return the other one.

Now, suppose I call **coalesce_free_list()**. This will combine the two free list nodes into one so that the state of memory is as follows:

- The free list contains one node, which is 10000 bytes and starts at address 0x10800.
- The four bytes starting at address 0x10800 contain the number 10000.
- Free list begin() will return 0x10800.
- Free_list_next(0x10800) will return NULL.

Testing your code and Grading Program Quirks

Were I you, I'd write my own testing program and compile it with **mymalloc.c**, before bothering with the gradescripts.

The way that the gradescript works, is that it copies **mymalloc.h** from the lab directory, and one of the C programs from /home/jplank/cs360/labs/Lab-7-Malloc/Gradescript-Examples into test.c. It then compiles test.c and **mymalloc.c**, and runs the executable. If the executable prints "Correct", then you have passed the gradescript.

The gradescript examines memory in terms of the values returned from **my_malloc()** and the contents of the free list to make sure that everything works. For example, the first few programs will perform a few **my_malloc()** and/or **my_free()** calls that require fewer than 8192 bytes. It will then look at memory and make sure that the 8192 bytes that were allocated are being used in a correct manner. There should be no wasted bytes.

Restrictions

- Your program may have one global variable. Mine is the pointer to the head of the free list.
- You may not call **sbrk()** with values smaller than 8192 bytes (with the exception of calling **sbrk(0)** to check the current end of the heap).
- You may not make any external procedure or system calls with the following exception: You may make one call to malloc(), one call to qsort() and one call to free() during coalesce_free_list().
- To reiterate, you are not allowed to otherwise utilize libraries like the dllist or jrb libraries.
- You should not assume that **sbrk()** is not called by other procedures that the user may use. Since that was

an awkward sentence, I'll try it without the double-negative: You should assume that other procedures may call **sbrk()** between calls to **mymalloc()**.

- You may not allocate more than 8 bytes of bookkeeping for eached allocated chunk.
- Since the **-m32** compiler flag is gone, you will be working with 8-byte pointers. That means that your free list nodes will be 16 or 24 bytes. (When I implemented this, I used a singly linked list, so mine is 16 bytes). The code in the grading script should work on both 32 and 64 machines. My recommendation is that you work on a 64-bit machine, so that your code will work on the lab machines. The compiler yells at me on my mac, saying **sbrk()** is deprecated, but my code still works.