Project 3: A Custom malloc()
Due: Sunday, November 15, 2015, at 11:59pm

Description

In our discussions of dynamic memory management we discussed the operation of the standard C library call, malloc(). Some implementations of malloc designate a region of a process's address space from the symbol _end (where the code and global data ends) to brk as the heap. We will be taking a slightly different approach and asking the OS for a large region of memory to act as our heap.

As part of dynamic memory management, we also discussed various algorithms for the management of the empty spaces that may be created after a malloc()-managed heap has had some of it's allocations freed. In this assignment, you are asked to create your own version of malloc based upon the buddy allocator scheme.

Details

The buddy allocation algorithm requires a large free space that we can repeatedly divided to find a reasonably-sized chunk to return. We will request this initial free space directly from the OS by requesting the OS allocate us many contiguous pages using the mmap() system call.

We will create an initial space of 1GiB by doing the following:

```
void * base = mmap(NULL, MAX_MEM, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0, 0);
```

We can easily set MAX_MEM to 2^30 by using a left shift trick:

```
1 ≪ 30
```

The pointer base will now point to a region of 1GiB contiguously allocated in size that we can use for our buddy allocator.

As we discussed in class, we will use a doubly-linked list representation for the free lists. Each chunk in the free list contains *chunk header* information, the *prev pointer*, and the *next pointer*. The chunk header is one byte long where the first bit is the occupancy bit and the remaining 7 bits represent the size of the chunk. The size is represented by $\log_2 N$ where N is the size in bytes. Hence, the 7 bits can represent a size of up to 2^{127} bytes. The occupancy bit and the size are required for the coalescing algorithm. The minimal size of a chunk is 32 bytes (the block header and the pointers take up 17 bytes). The block header is always present in a chunk regardless of whether that chunk is allocated or free. However, the prev and next pointers are only needed when a chunk is free because they are only used to link chunks in a free list. Hence, the size of *usable* space in an allocated chunk is N-1 bytes (one byte

is taken up by the block header). Obviously, your malloc should return a pointer to this usable space, not a pointer to the chunk header.

Now we should build a table of 26 pointers to be the heads of the doubly-linked lists of our chunks of each power-of-two size we support (2^5 to 2^{30}). The first pointer would be the head for the 32-byte chunk list. The last pointer would be the head for the 1-GiB chunk list.

We have discussed in class how to discover a buddy for a given chunk using the XOR operation (^ in C). It is important to note that this only works when the base of the chunk is at address 0, which is never going to be the case for us as mmap will return an address from the memory-mapped area at high addresses in our address space. That means to use the XOR trick, we will always need to subtract base from the address to get the "offset" into our 1GiB region.

Requirements

You are to create two functions for this project.

1. A malloc() replacement called `void *my_buddy_malloc(int size)` that allocates memory using the buddy allocation scheme.
2. A free() called `void my_free(void *ptr)` that deallocates a pointer that was originally allocated by the malloc you wrote above.

Your free function should coalesce buddy free blocks as we described in class.

As you are developing, you will want to create a driver program that tests your calls to your mallocs and frees. For grading, we will use the driver `/u/SysLab/shared/mallocdrv.c` and a second driver program in addition to this one in order to test that your code works. Mallocdrv.c is designed to use the standard C library malloc as-is. Make sure you modify the MALLOC and FREE macros defined at the top to your own functions to have it work with your code. Also, you will need to include your mymalloc.h header file appropriately. The output from using your own buddy malloc and from using the C library malloc should be identical.

Environment

For this project we will again be working on `thoth.cs.pitt.edu`

Hints/Notes

- In C, the sentinel value for the end of a linked list is having the next pointer set to NULL.
- gdb is your friend, no matter what you think after project 2

What to turn in

- A header file named mymalloc.h with the prototypes of your two functions
- A C file named mymalloc.c with the implementations of your two functions
- The test program you used during your initial testing
- A Makefile for your project. It should 1) on typing 'make', produce object files for mymalloc.c and the shared test driver then link them into an executable file, and 2) on typing 'make clean' remove all object files and the executable file that was produced by typing 'make'.
- Any documentation you provide to help us grade your project

To create a tar.gz file, if your code is in a folder named project3, execute the following commands:

```
tar cvf USERNAME-project3.tar project3
gzip USERNAME-project3.tar
```

Where USERNAME is your username.

Then copy your file to:

```
~wahn/submit/449/
```