Homework #2: Memory Hole

Issued: Monday, January 27
Due: Monday, February 24

Purpose

This assignment asks you to develop a memory allocator, using a scheme known as the Buddy System.

Your allocator will be similar to the `kmalloc()` allocator of the Linux kernel, or the `malloc()` allocator of the C library. Although your allocator would be easy to port to kernel space, you will test and use it in user space. It will not use `malloc()`, `sbrk()`, or `brk()`. It will obtain its memory from `mmap()`:

`https://en.wikipedia.org/wiki/Mmap`

Interface

Your allocator module's interface is:

```
4  typedef void *Balloc;

5  extern Balloc bcreate(unsigned int size, int l, int u);
6  extern void   bdelete(Balloc pool);

7  extern void *balloc(Balloc pool, unsigned int size);
8  extern void  bfree(Balloc pool, void *mem);

9  extern unsigned int bsize(Balloc pool, void *mem);
10 extern void bprint(Balloc pool);
```

`bcreate` constructs and returns a new allocator. Thus, an application can have multiple allocators. This is realistic, because memory may be non-uniform: regions may have different characteristics (e.g., speeds). The `size` argument specifies the total number of bytes that can be allocated. The `l` (lower) argument specifies that the smallest allocation will be 2^{l} bytes, even if a smaller amount is requested. The `u` (upper) argument specifies that the largest allocation will be 2^{u} bytes; a larger request will fail.

`balloc` requests a block of `size` bytes from an allocator.

`bfree` deallocates a block of memory.

`bsize` return the size of an allocation (not the request size).

`bprint` writes a textual representation of an allocator to `stdout`: a valuable debugging tool.

`bdelete` deallocates an allocator.

Resources

The Buddy System is described, generally, at:

```
http://en.wikipedia.org/wiki/Buddy_memory_allocation
```

There is more than one way to implement the Buddy System. A good description of how the Linux kernel does so, by kernel engineer Mel Gorman, can be found at:

```
https://www.kernel.org/doc/gorman/html/understand/
https://www.kernel.org/doc/gorman/html/understand/understand009.html
pub/doc/understand.pdf (Chapter 6-6.3, page 116)
```

Research the algorithm, but do not view or copy other people's code.

I have also provided two complete modules, and the interface to two modules:

```
pub/hw2/freelist.h
pub/hw2/bbm.[hc]
pub/hw2/bm.[hc]
pub/hw2/utils.h
```

You need not change the code in the two provided modules, but you must document it. This is part of the assignment!

You also need to document and implement the `freelist` and `utils` modules. If you wish, you may change their interfaces (i.e., `.h` files), but these are what I used in my solution.

Other Requirements

1. As above, there is more than one way to implement the Buddy System algorithm. For this assignment:

   (a) Do not store management data in allocated blocks. This would be very wasteful.

   (b) Do store management data in free blocks. In particular, for each block size, maintain a list of free blocks of that size, with a pointer stored at the beginning of each block.

   (c) The Linux-kernel memory-manager document describes how to use a bitmap, for each free list, to quickly determine whether a block's buddy is on that list. A buddy-pair's bit records that either buddy, or both buddies, are allocated.

   The Linux documentation assumes that a block's size, and thus its free list, is known. One way to determine a block's size is to use a bitmap, for each free list, to determine whether either buddy block was allocated from that list.

2. Your allocator must only call `mmap`, via `mmalloc`, during a `bcreate` call. For example:

```
mmap(0,size,
       PROT_READ|PROT_WRITE,
       MAP_PRIVATE|MAP_ANONYMOUS,
       -1,0)
```

3. If `bcreate` is passed a `size` argument that is not a power of two, or larger than 2^u, construct free lists of reasonably sized blocks.

4. You will find this assignment challenging to develop, test, and debug. Good modularity and unit testing help significantly. Write additional code to unit-test your modules. Write a "to-string" function for each data structure. If you follow these suggestions, debugging your code will be easier.

5. After your allocator works, use the "wrapper" provided in `wrapper.c` to compile, link, and test your *unchanged* double-ended queue, from an earlier assignment.

6. Your submission will be evaluated on `onyx`.