COSC 301: Operating Systems
Lab 3: Buddy system malloc library
Due: 4 November 2013, 23:59:59
Fall 2013

1 Overview

In this project, you will write a memory management library for user applications. Your library will implement the `malloc` and `free` functions that you have used in programs so far.

The goals for this project are to learn how malloc and free actually work and how heap memory is managed, and to understand better how pointers and pointer arithmetic work in C.

As with previous projects, you are encouraged to work in groups of 2–3 students. Only one submission needs to be made for the group. Please make a note in the file(s) which students worked on it.

2 Detailed Description

2.1 Getting Started

In this project, you will create a library that implements the necessary calls to support heap memory allocation and deallocation for applications. We will implement our own versions of the standard `malloc` and `free` calls (as well as one other function), and use the "buddy algorithm" for managing the heap. Because we will support the standard `malloc` and `free` calls, our test program will be written as a "normal" program that uses `malloc` and `free`[1].

Note that because we are (re-)implementing `malloc` and `free` in this project, you will not be able to use the built-in C-library versions of these functions! That is, if you wish to allocate heap memory for data structures used by your library, you must use your own home-grown mechanisms — that's the key point of this project. We will instead use the `mmap` system call (described below) to request a large memory segment from the OS, and manage that memory segment within our library.

To get started with this project, fork and clone the repo at `https://github.com/jsommers/cosc301_proj03`. There are three source files in the repo: a Makefile, `buddy.c` and `main.c`. `main.c` currently just has a few calls to malloc and free to help with testing. Your code to implement `malloc` and `free` should all go in `buddy.c`.

You should also read the *Free Space* chapter in OSTEP: `http://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf`. It contains some useful information on the buddy algorithm, as well as other useful discussion on how malloc libraries are implemented. Note, however, that there are some differences with how you should implement the free list from how it is described in the text. Still, there are substantial similarities and you should carefully read that chapter.

2.2 Functions to implement

The three functions you will need to implement in `buddy.c` are as follows.

- **`void *malloc(size_t)`.**

  Basic malloc functionality.

[1] Note that we *could* compile our code in such a way to "inject" our library into *any* program that uses malloc. It's not hard. If you're curious, just ask.

The type `size_t` behaves like the type unsigned int. Your `malloc` should return a pointer to a block of memory of the requested size as void *, or NULL if there's no memory left in the system.

To maintain some bookkeeping information, you will need to add 8 bytes to each allocation request. You will also need to raise the request size to the nearest power of two greater than or equal to the request size. For example, if a request for 50 bytes is made, you should add 8, then raise that value (58) to the next larger power of 2 — 64. As another example, a request for 4 bytes should be raised to 16 (4 + 8 = 12, and the next larger power of 2 is 16).

- **`void free(void *)`.**

  Basic free functionality; it should free the block of memory pointed to by the given argument by returning the block of memory back to your free list.

  `free(NULL)` should have no effect.

- **`void dump_memory_map(void)`.**

  This function should print a "map" of allocated and free memory in the heap. The map should identify each free block, but you do not need to identify each allocated block; it is sufficient to identify a "region" of allocated memory.

  An example of what this function might print out for a heap size of 4KB (4096 bytes) is shown below (you can choose your own format):

```
Block size: 64, offset 0, allocated
Block size: 64, offset 64, free
Block size 128, offset 128, allocated
Block size: 256, offset 256, free
Block size: 512, offset 512, free
Block size: 1024, offset 1024, free
Block size: 2048, offset 2048, free
```

  You may wish to invoke this function at different points while testing in order to verify that your `malloc` and `free` functions are working correctly. Your library should register the `dump_memory_map` function to be called automatically when a test program exits. There is a system call that enables this: `atexit()`. Note that `atexit()` should only be called once from within your library. (Note that there is already some code in the template `buddy.c` file that does this.)

2.3 Maintaining the heap

As described above, you must use the buddy algorithm for allocating memory and managing the heap. The basic idea with the buddy algorithm is as follows:

1. Create a fixed-size heap that is size 2^n for some value of n. In the git repo, the starter code in `buddy.c` creates a fixed-size heap of 1MB (2^{20} bytes) using the `mmap` system call. Please carefully read that code and the related comments.

2. For each allocation request (via `malloc`) add 8 bytes, and raise that value to a power of 2 that is equal to or greater than the request size. Let's call this modified request size the "upped request".

3. Search the heap for a block that is equal to or larger than the upped request.

   (a) If the free block found is larger than the upped request, recursively divide the free block in half and update the free list until there is a free block exactly the size of the upped request.

   Splitting a free block in two creates a pair of identically-sized free blocks, called "buddies". This is where the name "buddy system" comes from.

   (b) Update the free list for the block to be given to the caller of `malloc`.

   (c) Return the block.

   (d) If there is no free block available to satisfy the `malloc` request, return `NULL`.

4. For each call to `free`, update the free list. You can assume that the pointer given as the argument to `free` is the same address that you returned in an earlier call to `malloc`. It is the caller's responsibility to make sure this is the case, so if there is heap corruption, it isn't your fault!

   If the buddy of the newly freed block is also free, you must join the buddies to make a new, larger free block. Note that joining a pair of buddies may result in an additional pair of buddies that can be joined — you should keep joining buddies as long as it is possible.

2.3.1 Keeping track of the free list

To maintain the free list, you should use an "embedded" linked list in the heap. This linked list is not a linked list in a traditional sense because you will not be using `malloc` to allocate nodes and link them together. Instead, you will embed fields (two integers) in the first 8 bytes ($2 \times sizeof(int)$) of every free block. The following sketches how you could do this.

For each free or allocated block, you could maintain a *header* that contains two integers: a size and an offset, or "next" value. Each of these items should be 4 bytes in length, giving a total header size of 8 bytes. The block size should be stored in number of bytes as an integer. The offset should also be stored as an int and should contain the distance (offset) between the first byte of the current free block and the first byte of the next free block. For the last free block in the chain, the value of the offset should be 0, to indicate NULL.

Note that with this design you should never allocate a block for less than 8 bytes due to the overhead of the block header. For example, when you receive a malloc request for 10 bytes, you will actually need to allocate 32 bytes to satisfy the request: 10 + 8 = 18, and the power of 2 greater than or equal to 18 is 32. The first 8 bytes will be used for library bookkeeping and the last 10 bytes will be for the user. *Your malloc function should return a pointer to the first byte of the user bytes* — not to the header. To accomplish this (and other necessary bookkeeping tasks within your library), you will have to become familiar with "pointer arithmetic".

In C, if p is a pointer to a block of heap memory, the expression `p + i` actually means `p + i * sizeof(p)`. Thus, if p is a pointer to an int, p+1 will yield the address of p + 4 bytes (assuming an int consumes 4 bytes). If p is a pointer to char, p + 1 will yield the address of p + 1 byte (the char type consumes one byte). Note that you cannot use pointer arithmetic with pointers to void (i.e., void *) since sizeof(void) is nonsensical.

So, to allocate a block of size 10 (upped to 32) and return that to the user, all while maintaining our library bookkeeping, we might do something like the following. Here, let's assume that the heap size is 64 bytes to begin with, so our free list should point to the entire heap, with the header fields containing 64 (the size of the free block) and 0 (offset to next free block is 0, indicating that this is the last free block in the chain). Our heap should like like this:

Assuming that a variable `freelist` points to the first byte of the heap, we might update our free list using the following code to allocate 32 bytes. **Note**: this code is crafted very specifically to divide the 64 byte heap in half and to update the freelist. It is *not* general enough, but should hopefully serve as an example to better understand what you'll need to do.

```
void *example_partial_malloc() {
    int request_size = 32;

    // get a pointer to the block we're allocating (head of the freelist)
    uint32_t *block_to_allocate = freelist;
    *(uint32_t + 0) = 32; // write header size: 32 bytes
    *(uint32_t + 1) = 0; // write header offset: 0 bytes

    // free list should point to a block 32 bytes from the beginning
    // of the original free list block.
```

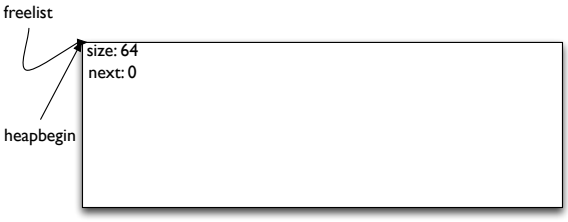
```
    freelist = (uint8_t*)freelist + request_size;
    *((uint32_t*)freelist + 0) = 32; // write header size: 32 bytes
    *((uint32_t*)freelist + 1) = 0; // write header offset: 0 bytes

    // return a pointer to the 8th byte *after*
    // the beginning of the block that is being allocated
    return (void*)(block_to_allocate + 2);
}
```

Your free list should always be ordered by address. The offset part of the header should point to the next block in the free list, or zero if the current block is the end of the free list. For the offset, the value represents the number of bytes between the first bytes of consecutive free blocks.

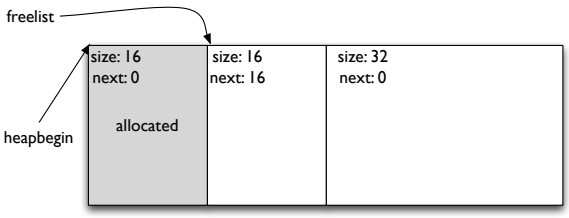
2.4 Example

Here is an example of maintaining the heap and free list for a series of `malloc` and `free` calls, assuming that the heap contains 64 bytes.

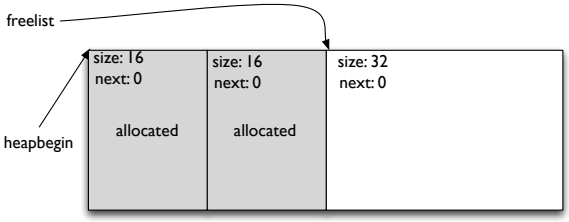
- The heap after initialization. Nothing has been allocated; all 64 bytes of the heap are free.



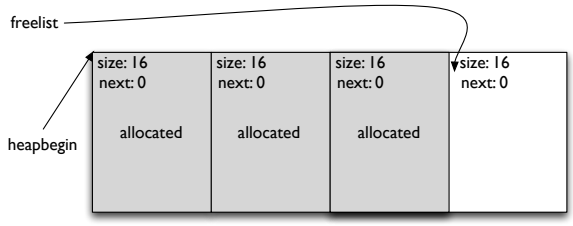
- The heap after satisfying an (upped) allocation request for 16 bytes. The heap was first divided in 2, giving two free blocks of 32 bytes. The first free block was then divided in 2, giving two more blocks of 16 bytes each. The first of those blocks was allocated. The free list includes two blocks. The "next" value in the first free block indicates that the next free block is offset by 16 bytes from this block (that is, it is located 16 bytes away from this block).



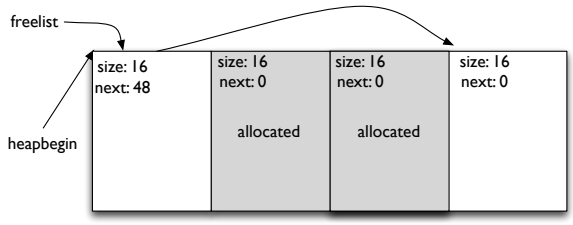
- The heap after satisfying another upped allocation request for 16 bytes. The second chunk of 16 bytes was given out and the free list was updated.



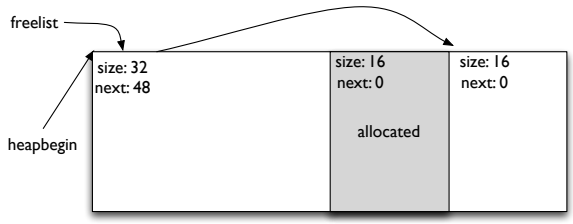
- The heap satisfying yet another upped allocation request for 16 bytes. The 32 byte block was split and the free list now refers to the last 16 byte block on the heap.



- The first 16 byte block is freed. The free list is now updated to point to that block first, then the last 16 byte block. Note that when we receive a pointer to the block to be freed, the pointer we receive will be to an address that is exactly 8 bytes *after* the beginning of the block. Thus, we can determine the exact allocation size by looking at the integer (4 bytes) located 8 bytes before the beginning of the address (pointer) we receive in the call to `free`. (Otherwise, there is no way that we could tell whether the block to be freed is 16 bytes or 32 bytes — we have to use this embedded header information.)



- The heap after the first 16 byte block is freed. That block is coalesced with its 16 byte "buddy", giving a free block of 32 bytes.



- Note that if the last allocated block is freed, we will have a picture similar to the first diagram above: all blocks will be coalesced with their "buddies", leaving us with a single 64 byte free block.

3 Tips and Hints

Some general tips for this project:

- Implement the simplest free list allocation strategy: first fit. You aren't being graded on efficiency in this project, so take the path of least resistance. You can easily modify this later to do next fit (rotating first fit) if you wish, but initially, keep it simple.
- Initially, don't worry about coalescing adjacent free blocks. Deal with that later, once you get some of the basics working.
- Be sure that you understand pointer arithmetic. Understanding how pointer addition works in C is essential to getting this project done correctly. The Kernighan and Ritchie C book has excellent information. You may also wish to write some simple test programs to verify that you understand how pointer math works.
- Draw out specific corner cases and various scenarios for allocation and freeing in order to understand how your free list information will change. The above example scenario should help, but drawing additional scenarios is also a good idea.

- Use `assert` function calls copiously in your library code to check for unexpected conditions and to ensure that certain invariants hold (e.g., a pointer is non-NULL or similar). These error checks are essential for debugging libraries and help to flag error conditions early on in the development process. Once you `#include <assert.h>`, you can call the assert function and pass a boolean expression, e.g., `assert (x == 0);`. If `x` is not zero, your program will crash — this is the "feature" of the assert function! So, by sprinkling `assert` statements here and there, you can verify whether things that should be true, indeed are true, and cause your program to crash if they don't.
- Declare all internal functions (those that are not called by clients of the library) "static" to prevent naming conflicts with programs that link with your malloc library. For example:

```
static void malloc_helper(void) {

    // a helper function that user programs shouldn't (and can't) touch

}
```

  In C, this use of the keyword static means that only functions within the current file (e.g., buddy.c) can access the static functions. Similar things can be done with private library variables.

4 Submission

Commit and push your code to github, and submit your repo name to Moodle.