Navajit Baruah
S. Harish Koushik
Submission Date: 6-Dec-15

Project 3 – Buddy Memory Allocator

Introduction

In a buddy system, the allocator will only allocate blocks of certain sizes, and has many free lists, one for each permitted size. The permitted sizes are usually either powers of two, or form a Fibonacci sequence (see below for example), such that any block except the smallest can be divided into two smaller blocks of permitted sizes.

When the allocator receives a request for memory, it rounds the requested size up to a permitted size, and returns the first block from that size's free list. If the free list for that size is empty, the allocator splits a block from a larger size and returns one of the pieces, adding the other to the appropriate free list.

When blocks are recycled, there may be some attempt to merge adjacent blocks into ones of a larger permitted size (coalescence). To make this easier, the free lists may be stored in order of address. The main advantage of the buddy system is that coalescence is cheap because the "buddy" of any free block can be calculated from its address.

A binary buddy heap after allocating a 10 kB block; note the 6 kB wasted because of rounding up.

For example, an allocator in a binary buddy system might have sizes of 16, 32, 64, …, 64 kB. It might start off with a single block of 64 kB. If the application requests a block of 8 kB, the allocator would check its 8 kB free list and find no free blocks of that size. It would then split the 64 kB block into two block of 32 kB, split one of them into two blocks of 16 kB, and split one of them into two blocks of 8 kB. The allocator would then return one of the 8 kB blocks to the application and keep the remaining three blocks of 8 kB, 16 kB, and 32 kB on the appropriate free lists. If the application then requested a block of 10 kB, the allocator would round this request up to 16 kB, and return the 16 kB block from its free list, wasting 6 kB in the process.

A Fibonacci buddy system might use block sizes 16, 32, 48, 80, 128, 208, … bytes, such that each size is the sum of the two preceding sizes. When splitting a block from one free list, the two parts get added to the two preceding free lists.

A buddy system can work very well or very badly, depending on how the chosen sizes interact with typical requests for memory and what the pattern of returned blocks is. The rounding

typically leads to a significant amount of wasted memory, which is called internal fragmentation. This can be reduced by making the permitted block sizes closer together.

Implementation details:

We have extended the skeletal code framework given in the project consisting of files buddy.c & list.h. Our implementation consists of modifications solely in buddy.c We have extended the `page_t` structure to include few more parameters for book keeping of pages as shown below which are initialized in function: `void buddy_init()`

```
typedef struct {
    struct list_head list;

    /* TODO: DECLARE NECESSARY MEMBER VARIABLES */
    int page_idx;    /* index in g_pages[] */
    char *page_addr; /* page start address in g_memory */
    int block_order; /* block size in power of 2 */
    int page_free;   /* Page is free or not */
} page_t;
```

We have assumed that the array `g_pages[1<<MAX_ORDER]` maps one-to-one to total available memory bank `g_memory[]`. Each `g_pages[i]` holds the book-keeping information for one physical pages in `g_memory[]`.

Functions Implemented in the code:

Initialization: `void buddy_init()`

We have used the initialization function to initialize the `page_t` structure and the list head. The list head is used to initialize to the free list of the order from MIN ORDER to MAX ORDER. The list_add api provided in the the list.h library is used to add the pages to the free list.

Allocation: `void *buddy_alloc(int size)`

When it receives a allocation request for `size` we find the equal or next higher power of 2 which we have to allocate (`block_order`) such that **`2^(block_order) >= size`**. Then loop through all free lists in `free_area[MAX_ORDER +1]` to find available list having a free area equal to `block_order`. If space is available in that list then allocate it to the resource, else, we move on to the next order till we find space available. Split the larger block in **two** as many times as necessary iteratively to get a block of a size `block_order`. We add all the created free blocks during splitting to the appropriate order list in `free_area[order]`. If

a free block is found we delete that entry from `free_area[order]` list and assign it to the request resource.

Free: `void buddy_free(void *addr)`

Input to the free function is the page address (`addr`) of the memory to free, in `g_memory[]`. From page_address we get the index of the page `page_idx` and from it the `page_t` struct. From there the size of the block to free in power of two. Using these two values(index and size) we calculate the buddy using the formula B2 = B1 XOR (1<<order). Check if buddy is free. If free, we coalesce the buddy. Time taken for the searching of buddy is O(1). We do this for next higher buddy of the coalesced block. Run the loop from block order to MAX_ORDER and break if we cannot coalesce any next higher order.

Discussion:

We were able to test the implementation using provided test code. We run a couple of more test cases and it seems to work fine.