CSC 452/552 Operations Systems

Project 4 Buddy System

Name: Enoch Levandovsky

Bronco ID: 114071701

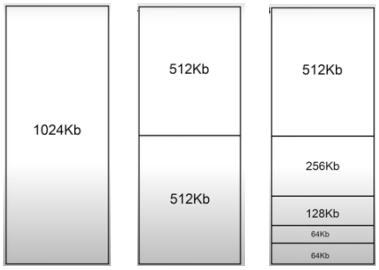
Date: December 3, 2024

1. Project Overview
2. Project Management Plan
    a) Task 1: Implement Buddy
    b) Task 2: Creating Extra Test Cases
3. Project Deliveries
    a) How to compile and use my code?
    b) Summary of Results.

1. Project Overview

The goal of this project was to learn about how to implement a dynamic memory allocation system using a well-known algorithm, the buddy algorithm. The buddy algorithm is a very fast memory management system, using binary trees, to allocated and merge unused blocks for future use cases.

The algorithm start as so, say you have a 1024kb memory block. If a program wanted to allocate 64kb of memory, we would first check if 64kb of memory fits inside the memory. If yes, then we would check if 64kb fits in half of 64kb. If yes, then we break the memory block into 2 512 kb. We would do this recurisvly till, the split up memory block is just enough for 64kb, as shown below.



One of the 64kb block are reserved and the rest of the blocks are still available for the system to use. The other 64kb block that is unused, is known as the **buddy**, every memory block is going to have a buddy, except for the main block, in which the algorithm will try to merge buddies each deallocation, attempting to always have the most largest block available for future uses.

One draw back is only buddies can be merged, which may cause a lot of fragmentation in the memory system make large memory allocation harder as many small allocation are called throughout the system.

This project plans to implement this algorithm along with a few unit tests to make sure all function are up and working.

2. Project Management Plan

1. Implementing buddy

To implement the buddy system, a tree like structure is need to keep track of all the memory blocks, as well as a method to keep track of buddies for de-allocation. Thus the **buddy_init** method is crucial part of the full algorithm to make sure the correct data structures is implemented. Before I explain how I created buddy_init, I will explain all the other functions first.

Btok

The goal of this function is to convert bytes to the power of k negating the header size, which by my observation is 24bytes (rounded up is 32). So the minimum K that we will need with the header is 6, which is consistent of the hardcoded constant provided by the teacher.

buddy_calc

The goal of the buddy calc is to find the buddy in order to deallocate it. Instead of storing a reference on every memory piece, one could algorithmically calculate the buddy location as discussed by the original paper.

$$\text{buddy}_k(x) = \begin{cases} x + 2^k, & \text{if } x \bmod 2^{k+1} = 0; \\ x - 2^k, & \text{if } x \bmod 2^{k+1} = 2^k. \end{cases}$$

However, we can also achieve this via an XOR operation as so

```
   101110010110000
 ^ 000000000010000
------------------
   101110010100000
```

This would allow us to get the left buddy if we are on the current right buddy, or the right buddy if we are currently on the left buddy.

Buddy_malloc

This is the core of the code and is a substitute for the system malloc. We start off with basic check to make sure we received not null values *line:46*. Then we check to see what is the power/K value needed to store the piece of memory *line:53*. Once we get the K value, then we check if our system can support that size *line:56*. After we confirm are memory system can support it, then we check the pool of available memory blocks if there is enough free memory *line:68* because there is a chance the memory is too fragmented or just not enough space. Otherwise we remove that block from available blocks and allocate the memory in that block *line:81*. Although we can store the memory in that block, we have to check if we can be more efficient and split block up, but when we split the blocks, we send the unused buddy blocks to available blocks. line:85,101. Once we found/created the right block we allocated the memory into it line:111-113

Buddy_free

Buddy free would do the opposite of buddy_malloc except that it would first check if its buddy is free, if its buddy is free it would merge the blocks together *line134-160*

after the final block is determined to set free, it would add it to the pool of available blocks *line:165*

buddy_init

The purpose of buddy_init is to initialize the data structures needed to maintain buddy mallac. We first set default minimum block size to DEAFULT_K, that is also conveniently minimum memory needed to store the headers for the smallest memory block. From there we find the large possible K value we can get for the size passed in as the value *line222-245*. Then we allocate the memory needed *line252.* And pass that block as the first available block for memory allocation.

Print_buddy_block

The last additionally added function worth noting is print_buddy_block. This is a debugging function that prints a tree of memory allocation and available blocks as shown below. This takes advantage of the pool function which has a list of all the available allocation.

```
Buddy memory system:
                            64 byte bock AVAIL
                            64 byte block RESERVED
                          128 byte bock AVAIL
                        256 byte bock AVAIL
                      512 byte block RESERVED
                    1024 byte bock AVAIL
                  2048 byte bock AVAIL
                4096 byte bock AVAIL
              8192 byte bock AVAIL
            16384 byte bock AVAIL
          32768 byte bock AVAIL
        65536 byte bock AVAIL
      131072 byte bock AVAIL
    262144 byte bock AVAIL
  524288 byte bock AVAIL
Buddy memory system:
```

2. Creating Extra Test Cases

The project came with some a few basic test cases such as testing initialization, check allocation and deallocation.

I create 3 custom new test cases

test_oom

The purpose of this function is to try to test for an OOM by allocating more memory then possibly avalible. It check for that by asserting the bad mem_allac returns a NULL value

Test_print

This is testing function to test if print_buddy_block prints correctly. The testing function should PASS 100% of the time. A user could do additional check to see if the tree is correctly allocated as it should

Test_buddy_realloc

Since no default test function tested this function, I created an additional function test the buddy_realloc. The purpose of this is to see if I relocate a large size of memory from an already allocated memory block. I use print and assertions to check if memory was reallocated correctly.

Project Deliveries

Run my code

To run the code do the following

To compile the code
`make`

To run a test code
`./myprogram`

To run the tests + custom tests
`./test-lab`

and outputs would like like as so

```
Buddy memory system:
                            64 byte bock AVAIL
                            64 byte block RESERVED
                          128 byte bock AVAIL
                        256 byte bock AVAIL
                      512 byte block RESERVED
                    1024 byte bock AVAIL
                  2048 byte bock AVAIL
                4096 byte bock AVAIL
              8192 byte bock AVAIL
            16384 byte bock AVAIL
          32768 byte bock AVAIL
        65536 byte bock AVAIL
      131072 byte bock AVAIL
    262144 byte bock AVAIL
  524288 byte bock AVAIL
Buddy memory system:
```

3. Summary of results

Buddy_malloc behaves just as it should and should, at high level, behave just like the regular c malloc function. There is not much results from this as it's a system designed to run in the background. I have passed all the test cases provided by the project descriptions and added a few more.

```
  524288 byte bock AVAIL
Buddy memory system:
  524288 byte bock AVAIL
  524288 byte block RESERVED
Buddy memory system:
tests/test-lab.c:215:test_buddy_realloc:PASS

-----------------------
6 Tests 0 Failures 0 Ignored
OK
```

In this project I have demonstrated creation of my own malloc system. I learned many challenge that can come when trying to create an efficient memory allocation system that includes managing the memory to make sure its not too fragmented. The goal was to make **buddy_malloc** perform exactly the same as **malloc**. I also learned my about memory pointer arithmetic that allowed me to allocate and deallocate memory very quickly.