PA1 – BUDDY ALLOCATOR

Muhammad Taha Haqqani
CSCE 313-507

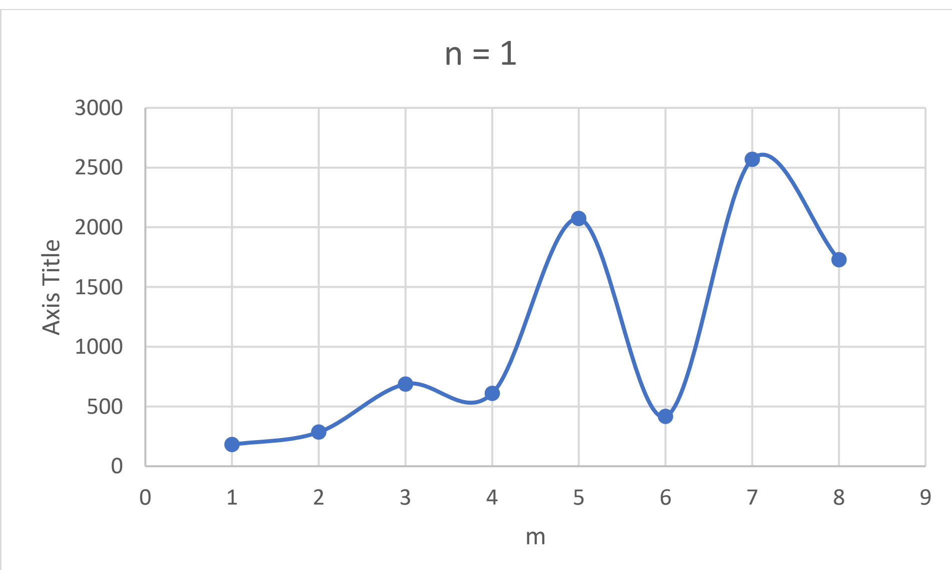
Introduction

In this Programming Assignment, we implemented a buddy-system memory manager that allocates memory in blocks with sizes that are a power of two and multiples of the basic block size. The user can request memory using the alloc() function and the memory manager will find the appropriate block size. The alloc() function will sometimes use the split() function if the available block is too large and can be split into smaller block sizes. This memory is deallocated using the free() function which puts the memory block back in the list. This memory block can be merged sometimes if its buddy is available. This is why its called a buddy system allocator. Key points in this assignment:

- Every memory block has a blockheader which contains the information about each memory block like the size of the block and the next block in the list
- A linked list is used to hold all blocks of the same size
- A vector is used to hold the lists and every index of the vector contains a list for a specific block size
- Some memory (24 bytes in my program) is used up to store header information
- The memory returned to the user will have the header removed so that the user cannot manipulate the header information
- The buddy system is not an ideal memory manager because a lot of memory gets wasted in storing header information and not all the memory given to the user is being used

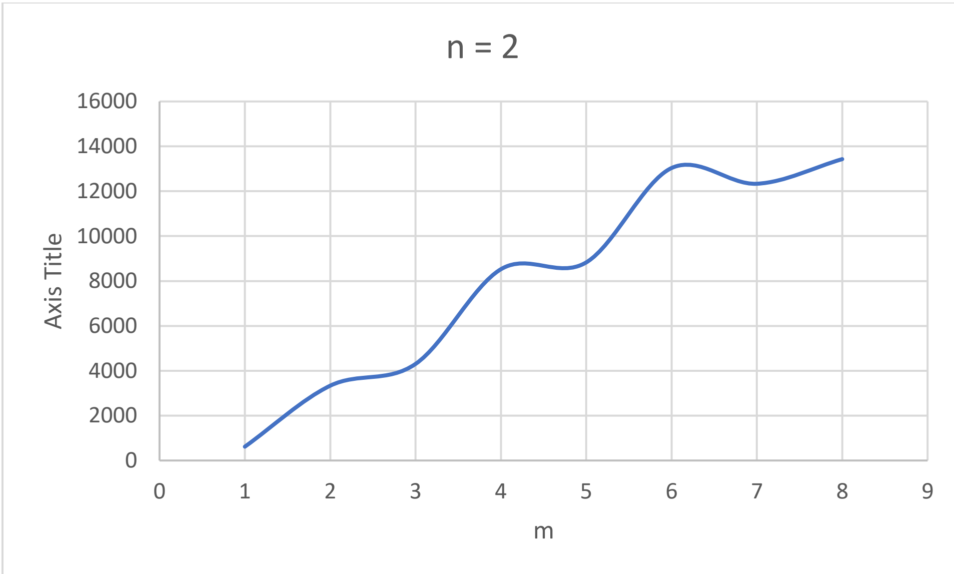
Ackerman Graphs:

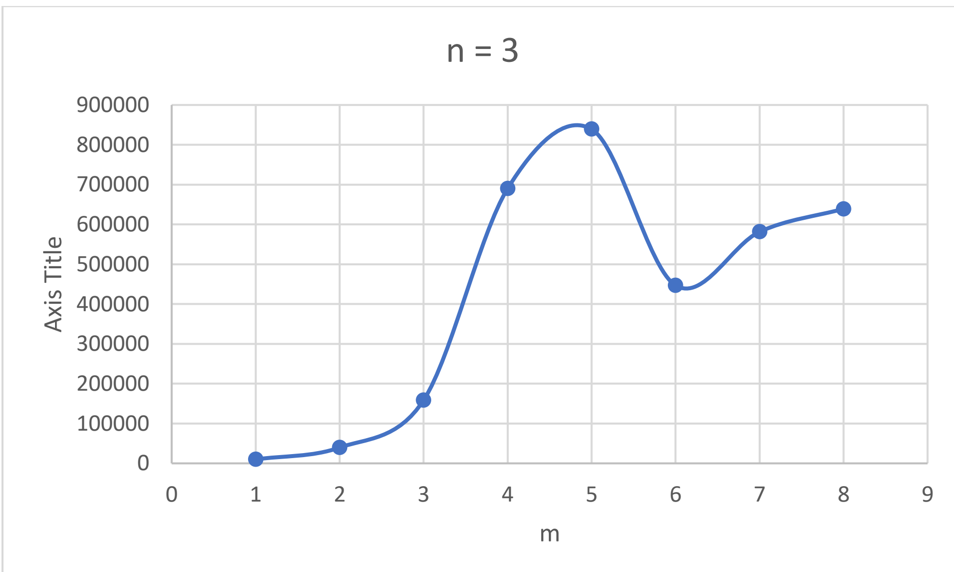
The following graphs and data were obtained when the program was run with basic block size = 128 and total memory = 512 MB. The conditions were kept constant to avoid discrepancy in the results.



PA1 – BUDDY ALLOCATOR

Muhammad Taha Haqqani
CSCE 313-507





PA1 – BUDDY ALLOCATOR

Muhammad Taha Haqqani
CSCE 313-507

```
Please enter parameters n (<=3) and m (<=8) to ackerman function
Enter 0 for either n or m in order to exit.

  n = 1
  m = 1
Ackerman(1, 1): 3
Time taken:  [sec = 0, musec = 281]
Number of allocate/free cycles: 4

==================================================================
Please enter parameters n (<=3) and m (<=8) to ackerman function
Enter 0 for either n or m in order to exit.

  n = 1
  m = 2
Ackerman(1, 2): 4
Time taken:  [sec = 0, musec = 261]
Number of allocate/free cycles: 6

==================================================================
Please enter parameters n (<=3) and m (<=8) to ackerman function
Enter 0 for either n or m in order to exit.

  n = 1
  m = 3
Ackerman(1, 3): 5
Time taken:  [sec = 0, musec = 681]
Number of allocate/free cycles: 8

==================================================================
Please enter parameters n (<=3) and m (<=8) to ackerman function
Enter 0 for either n or m in order to exit.

  n = 2
  m = 1
Ackerman(2, 1): 5
Time taken:  [sec = 0, musec = 626]
Number of allocate/free cycles: 14

==================================================================
Please enter parameters n (<=3) and m (<=8) to ackerman function
Enter 0 for either n or m in order to exit.

  n = 2
  m = 2
Ackerman(2, 2): 7
Time taken:  [sec = 0, musec = 3390]
Number of allocate/free cycles: 27

==================================================================
Please enter parameters n (<=3) and m (<=8) to ackerman function
Enter 0 for either n or m in order to exit.

  n = 2
  m = 3
Ackerman(2, 3): 9
Time taken:  [sec = 0, musec = 4420]
Number of allocate/free cycles: 44

==================================================================
```

The following deductions can be made from the graphs above:

- The number of cycles increase as the value of n and m increase
- The time taken increases as the value of m and n increase
- The running time increases as number of cycles increase

The following conclusions and observations can be made from the results:

- The running time increases because the alloc() and free() can only run one at a time
- This the bottleneck of this memory manager
- There is some fluctuation in the graph because spitting and merging is not happening in every alloc() and free() call
- Most of the time is consumed in the recursive splitting and merging calls