Casey Brooks
CSCE 313 - 504
MP1 Analysis

Programs need memory so that they can store data and access it outside the local scope, and it turns out that making a program to allocate memory is never quite perfect. In order to reduce fragmentation, which is the waste of small blocks of memory due to the way blocks are allocated and deallocated, our job was to create a fibonacci buddy system. I was not able to get my allocator working, but I have done as much as I could manage and in the code I tried to explain what should be going on.

Nevertheless, it isn’t difficult to understand why the fibonacci buddy system is a much better solution to the reduction of fragmentation than a binary buddy system or a simple first-block-available. The latter seeks to find the first block that is available and returns the exact amount needed by the caller, but over time as these blocks are allocated and deleted, there forms a lot of tiny little blocks that cannot be used, which makes the performance of this allocator slow down greatly as it searches through a huge list for an available block, and it could run out of memory much sooner because of the bad external fragmentation. A binary buddy system works much better by reducing external fragmentation, but at the cost of high internal fragmentation, where more memory is allocated than is actually needed by the user.

The fibonacci system works better than both those above because it allows a greater number of block sizes than the binary system, which allows the system to more closely match an available block to what the user calls, and the buddy system reduces external fragmentation greatly. Overall, the cost of allocating or deallocating a single block is greater than either above system because it must do some additional calculation to find the correct fibonacci numbers for the block sizes. However, in the long term, as many blocks are allocated and deallocated, this individual cost is amortized among the total time and fragmented memory saved by this more efficient implementation.

My current implementation, though not working, calculates the correct fibonacci numbers to be used for each step, and this is a potential bottleneck in the system. Each call to allocate or deallocate a block necessarily calculates numerous fibonacci numbers. A more efficient implementation would be to start the allocator by creating a lookup table of all the possible fibonacci numbers from 1 to the total amount of memory available to the program and just search for the correct number each time rather than calculate. This would reduce the execution time of each individual call, which makes an overall faster program, but at the cost of more memory being used by the allocator.