Project 3 Write Up
Tom Leech and Jackson Kearl

Implementation:

We decided to implement a binned free list using memory blocks in powers of two. Each memory block in the free list has a header composed of the log size of the block, a free value to mark the block as free or not, and pointers to the next and previous blocks in the bin. Each memory block also has a footer containing the log size and free value. When a block of memory is allocated from the free list, we allow the caller to overwrite the next and previous pointers but the size and free values in the header and footer are to be kept intact. We accomplish this by placing them outside of the range returned to the caller.

My_malloc:

First, we check if there is an available chunk of memory of the appropriate size. If there is, we remove that node from the free list and return it. Otherwise, we check up the bins to find the next biggest chunk of memory in the free list. If a larger chunk of memory does not exist, we call mem_sbrk to add more memory to the heap. If a larger chunk does exist, then we split it into smaller pieces until we have a correctly sized chunk of memory to return.

My_free:

To free a block of memory, we create a new node in our free list and add it to the appropriate bin. We then search for an opportunity to coalesce our newly freed block.

Coalese_search:

Using our header and footer, we can find the blocks of memory adjacent to the newly freed block. We then check to see if either adjacent block is equal in size to the current block and if so, remove both blocks and replace them with one larger block double the size of the original. We then recursively call coalese_search on the larger block. One issue with this strategy is that it may sometimes suboptimally coalesce a chunk with a neighbor on one side whereas coalescing with the other neighbor may have allowed for an even longer chain of coalescing.

My_realloc:

We first check the simple cases when the new size is equal to zero, the given pointer is null, or the new size would fall in the same block as the old size. Then, if the new size would fall in a smaller block, we adjust the header, free the unused old memory, and return the original pointer. If the new size would fall in a larger block than the original, we malloc a new piece of memory, copy the old data over, free the old block, and return our new block of memory.

Discarded Work:

We looked into implementing arbitrarily sized bins for small sized allocations (up to 256 bytes), in order to only allocate exactly as much memory as required when there are many small chunks which would otherwise ruin utilization if they don't line up with a power of 2. However, we observed no improvement in utilization in this strategy and a reduction in throughput, so ended up discarding it.

Future Improvements:

We will look into using fibonacci block sizes in order to overcome the adversarial "just past a power of 2" cases that result in utilization of around 50%. Fibonacci should help even in adversarial-for-fibonacci cases because the maximum overhead ratio is phi rather than 2. We will also look into ways to more optimally coalesce chunks of memory.

Packing of chunk type:

```
                    Start of pointer passed to caller ---------------(requested memory here)--------------- End of region passed to caller
                              V                                                                                          V
```

Size (uint32_t)	IsFree (uint32_t)	Next (freeblock*) (overwritten when occupied)	Prev (freeblock*) (overwritten when occupied)	Data	Data	…..	Data	Size (uint32_t)	IsFree (uint32_t)