Baraa Orabi & Cagdas Oztekin

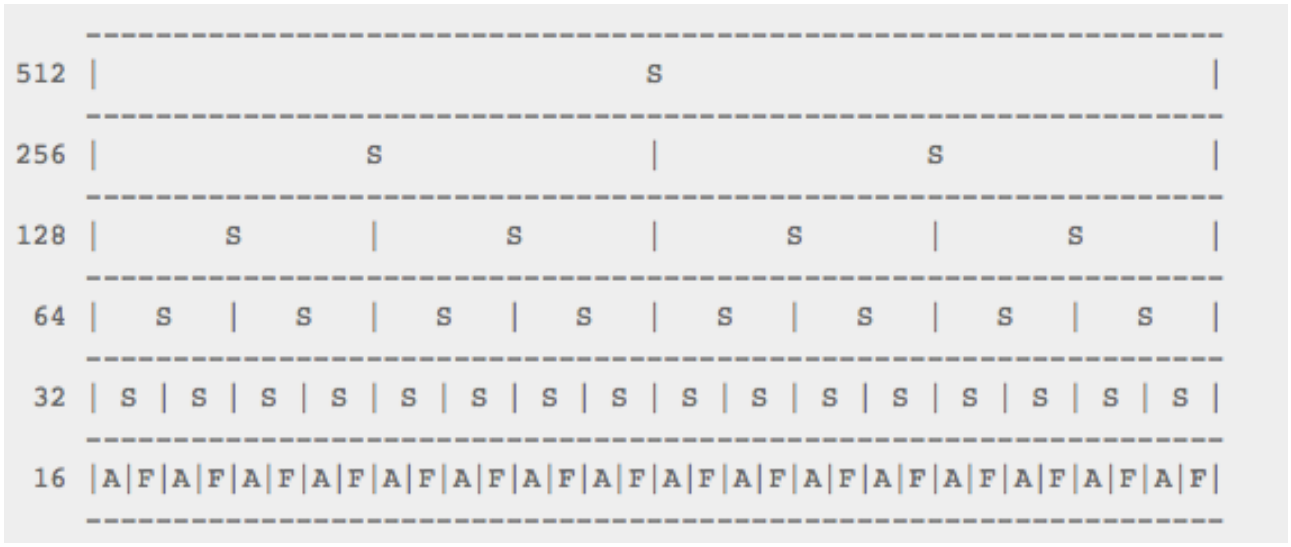
As said multiple times in the assignment sheet, we do not use dynamic variables in our library but only static variables. We have only static variables:

```
#define MAX_LEVELS 17 //assuming
#define MIN_REQ_SIZE 256
//size of the contiguous memory chunk
int size;
//start address of the memory chunk
long int * start_address;
//pointers to the start addresses of the first free blocks of each level
long int free_block_lists [MAX_LEVELS];
int powers_of_two[17];
```

The above code bit is taken from our buddy.c, for the variables that haven't been well-documented: powers_of_two[ ] is somewhat like a lookup table to get the n-th power of two without hassling to calculate it and waste cycles.

Our binit() function, initializes size to input parameter chunksize times 1024 since the parameter is in kilobytes. The start address of the chunk is initialized to the value of input parameter chunkpointer. The powers of two in the array are assigned their respective values. And, importantly, chunk's first 8-byte is assigned 0 and the starting address is added into the list of free blocks at index 0.

Let us explain why these are done:
1
free_block_lists[level] point to a free block at a given level. The reason why we said free_block_lists[0] = start_address is initially, the chunk itself is a free block at the 0-th level, therefore must be in the list at level 0. The other pointers in this list are initially NULL but as allocations/deallocations are done this changes.
2


512 | S |
256 | S | S |
128 | S | S | S | S |
64 | S | S | S | S | S | S | S | S |
32 | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S |
16 |A|F|A|F|A|F|A|F|A|F|A|F|A|F|A|F|A|F|A|F|A|F|A|F|A|F|A|F|A|F|A|F|

(Figure 1. Taken from http://bitsquid.blogspot.com.tr/2015/08/allocation-adventures-3-buddy-allocator.html)

Our implementation complies with the above figure and if we have a chunk with size n, a block with size n/2^p will be at level p. For example, if our chunk size is 512 bytes as in the figure, then a block with size 16 will be at level log(512/16), 5.

So for each block, we're using its first 8-byte to store information about on what level it is and whether it is free or not. And, for each free block on top of the the 8-byte at its start address, we're using two more 8-byte data, one to store the address of the previous free block of the same size as it is and one to store the address of the next free block of the same size.

When we are allocating a block, we are first checking if the requested size lays within the boundaries of our chunk, if it does, we first check if there is an available block at the desired level in our free_blocks_list array, if there is we are allocating the block at the free_blocks_list to the object and we're setting the free_blocks_list's level-th index to be the next block pointed by the allocated block, so our free_blocks_list at that level now points to the next free block at that level.

If there are no available blocks of the requested size in the free_blocks_list we recursively go to the upper level and ask a block from the upper level, if a free block is found it is split and until we reach back to the requested block's appropriate level. This is done recursively and all the free blocks that are generated during this process are added to the free_blocks_list and their next pointers become the original blocks in the list.

If any block on an upper level cannot be found, the request is sent back with an error.

So, when deallocating blocks from the chunk, we are checking if its buddy is also free. How do we find the buddy? We get the chunk size, subtract from it the start address of the current block, divide by 2 to the current level-th power and voila, if what we get is divisible by 2 then it means it was a first block, not a second block when it was split into a block at its level. If our buddy is not free, the rest is free, we just add the current block to the free_blocks_list at its level, and set its next pointer as the previous address in the free_blocks_list at that level.

If the buddy is also free, it is a bit complex what we're doing, (this part is recursive), we have already found its buddy, then as there is no obstacle to their merging we are changing the first block's first 8-byte, decrementing its level value by 1, indicating now it's merged with its buddy and now on the upper level then we are calling the bfree function itself and do the same for the merged block. However, for the free block's already free buddy, we are setting its pointers too. If it was the block in the free_blocks_list, we are setting the block in the free_blocks_list to be buddy's next, else we're setting buddy's prev's next to buddy's next and buddy's next's prev to buddy's next, so that we don't lose the free blocks at that level.

For bprint, we start with the start address of the whole chunk, and depending on what level the current block is at, we print the current block and decide how much we need to move by the level value of the current block. If the first block of the chunk is at level 5, we then move by chunksize/ 2^5 bytes to its buddy. We keep on moving the pointer until we reach an address that is not within the chunk and by then we've have already had printed all the necessary information.

Performance of the program:

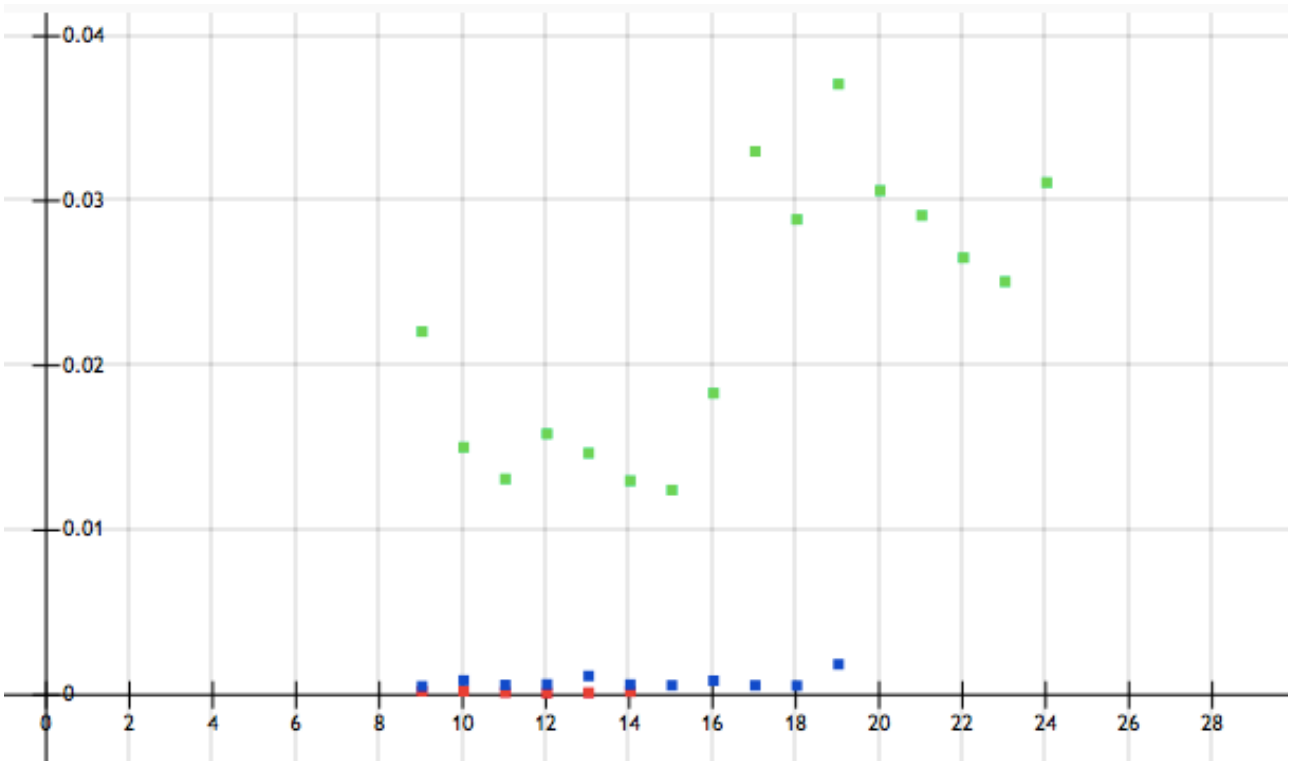


Figure 2.

For Figure 2, we ran the program with consecutive memory allocations of size 2^n. On the x axis you are seeing the changing n, from 9 to 24. On the y axis you see the runtime in milliseconds of the program with such configuration. The runtimes of the program are found over averaging the runtimes of 4 different runs with the same configuration. The red points represent runs with chunk size 32KB, the blue points 1MB and the green points 32MB. Since a block size cannot exceed the chunk size the red points go from n equals 9 to 14, the blue ones from 9 to 18 and the green ones from 9 to 24.

The most obvious interpretation from the plot is, increasing chunk size decreases the performance, which is very much expected since we are allocating and deallocating 2^(chunksize)/2^(blocksize) and as chunk size increases the result of the expression increases, and although there are fluctuations in the graph it can be said that increasing block sizes, resulting in a fewer number of allocations and deallocations, shortens the runtime of the program.