City University of Hong Kong

CS3103

Operating Systems

Dynamic Memory Allocation Report

Group 34

Nder Sesugh (54619300)

Srinivas Sivakumar (54486977)

December 6, 2018

1 Design and Summary

This allocator implementation makes use of segregated lists for organizing free blocks. The segregated list (*free_list*) consists of 20 classes defined by the macro *NUM_CLASSES*. The size classes, as shown in Table **??**, are in powers of two starting from 32 (2^5) until 2^{25}. Each free list is organized as an explicit list which is maintained in an ascending order. Although maintaining a sorted free list makes insertions slow (when compared to unsorted lists), it makes finding the best fit faster; hence, better utilization.

Size class	Max size
0	$2^5 - 1$
1	$2^6 - 1$
2	$2^7 - 1$
...	...
18	$2^{23} - 1$
19	...

Table 1: Default size classes using a power-of-two alignment

Each free block consists of a header, a footer, a pointer to the previous block in the list and a pointer to the next block in the list. The size of the header and footer are both 4 bytes. Information about the block sizes are found in Table **??**.

Vital Statistics	
Minimum block size	24 bytes
Header size	4 bytes
Footer size	4 bytes
Next pointer size	8 bytes
Previous pointer size	8 bytes

Table 2: Block information

To allocate a block, we determine the size class of the request (using the *get_size_class* method) and perform a first fit search of the appropriate free list for a block that fits. This is implemented in the *find_fit* function. If a free block is found, then we (optionally) split it and insert the block in the appropriate free list. Splitting is done if the block is the unused portion is large enough to be a free block, that is, it is, at least 24 bytes. If no fit is found, then we search the free list for the next larger size class.

We repeat until we find a block that fits. If not fitting block can be found in all the free lists, the malloc function increases the heap size, allocate the block out of this new heap memory and place the leftover in the appropriate size class.

When a block has been allocated, the block is removed from the list by calling the *delete_block* function. The pointers of the corresponding nodes are updated before removing the node from the list.

To free a block, it is coalesced and the resulting block is placed in the appropriate list.

2 Results

Figures **??** shows the trace results of the basic allocator discussed in section **??**. Two optimizations to the basic segregated list algorithm, discussed in section **??**, were carried out to obtain a performance index of 98%. The first was choosing not to extend the heap, when initializing the allocator (in *mm_init*). This led to better utilization - a 2% increase. The trace results when only this optimization was employed is shown in figure **??**.

The other optimization made was in placing blocks. If the size of the block is greater than a certain threshold (100 bytes in our case), the allocated block will be placed in a higher address space. This leaves the remaining free block in a lower address space.

Results for mm malloc:

trace	valid	util	ops	secs	Kops
0	yes	89%	12	0.000032	379
1	yes	99%	5694	0.001101	5173
2	yes	99%	5848	0.000750	7794
3	yes	99%	6648	0.001060	6271
4	yes	99%	5380	0.000964	5578
5	yes	66%	14400	0.001038	13876
6	yes	96%	4800	0.001378	3482
7	yes	95%	4800	0.001368	3509
8	yes	55%	12000	0.001577	7609
9	yes	51%	24000	0.002178	11017
Total		85%	83582	0.011447	7302

Perf index = 51 (util) + 40 (thru) = 91/100

Figure 1: Trace results of base allocator without optimizations

Results for mm malloc:

trace	valid	util	ops	secs	Kops
0	yes	89%	12	0.000007	1846
1	yes	99%	5694	0.001018	5594
2	yes	99%	5848	0.000754	7759
3	yes	99%	6648	0.001105	6018
4	yes	99%	5380	0.000968	5560
5	yes	99%	14400	0.000864	16659
6	yes	96%	4800	0.001674	2867
7	yes	95%	4800	0.001597	3006
8	yes	55%	12000	0.001822	6588
9	yes	51%	24000	0.002291	10476
Total		88%	83582	0.012098	6908

Perf index = 53 (util) + 40 (thru) = 93/100

Figure 2: Trace results of allocator using first both optimization

Results for mm malloc:

trace	valid	util	ops	secs	Kops
0	yes	89%	12	0.000028	423
1	yes	99%	5694	0.000802	7101
2	yes	99%	5848	0.000775	7551
3	yes	99%	6648	0.001088	6109
4	yes	99%	5380	0.000968	5560
5	yes	99%	14400	0.000745	19326
6	yes	95%	4800	0.001426	3366
7	yes	95%	4800	0.001344	3571
8	yes	95%	12000	0.001245	9642
9	yes	88%	24000	0.001735	13836
Total		96%	83582	0.010155	8230

Perf index = 58 (util) + 40 (thru) = 98/100

Figure 3: Trace results of allocator using both optimizations