Machine Problem I: A Simple Memory Allocator
CSCE 313-504

Derek Heidtke

September 28, 2015

Objective

The purpose of this assignment is to create an alternate implementation of the well-known malloc() and free() memory allocation functions. This simple memory allocator uses the buddy system design. i.e., The total memory space is given to the user in chunks that are integer powers of two. So, whenever the user requests memory, the largest block is split into two buddies (an 'A' buddy and a 'B' buddy). If that chunk is still too large for the user, the successive blocks are broken down until they are the correct size. When the user is done with the memory, it is joined with its buddy (if possible) in a casdading fashion until all of the memory has been returned. To test the functionality of the allocator, a recursive function, called ackerman(), repeatedly requests and frees memory.

How to Run

Compile by navigating to the working directory and running the command:

```
make all
```

To test the implementation of the memory allocator, run the following command:

```
./memtest [-s <memsize>] [-b <blocksize>] [-o]
```

-s <memsize>	defines the size of the memory to be allocated, in bytes (rounded up to the next highest power of 2). Default is 512kB.
-b <blocksize>	defines the basic (smallest) block size, in bytes (rounded up to the next highest power of 2). Default is 128 bytes.
-o	if this flag is enabled, allocator will print the freelist after each memory request and each memory return. These can be used to verify the correct operation of the allocator. (WARNING: produces a lot of text; use with small ackerman values of 'n' and 'm.')

Results & Discussion

The data in Table 1 shows the "correct" result values of the ackerman function along with the number of allocate/free cycles each calculation took. It is important to emphasize that these values were obtained using the original malloc() and free() functions. As such, it was very unlikely that the ackerman function would run into a situation where the the OS would be unable to give it some memory (because the memory space is so large). This is important because the ackerman function returns a different result if it could not complete without running out of memory. So, when using the customized my_malloc() and my_free() functions, (because the memory space is usually much smaller) the ackerman function is more likely to run out of memory, and thus, result in a different number.

-s 512KB	-b 128B						
n\m	1	2	3	4	5	6	7
1	(3,4)	(4,6)	(5,8)	(6,10)	(7,12)	(8,14)	(9,16)
2	(5,14)	(7,27)	(9,44)	(11,65)	(13,90)	(15,119)	(17,152)
3	(13,106)	(29,541)	(61,2432)	(125,10307)	(253,42438)	(509,172233)	(1021,693964)

Table 1: Original malloc() and free() results and cycle counts for differing 'n' and 'm' values. (result of ackerman(n,m), # of allocate/free cycles)

The data in Tables 2 through Table 7 shows that "correct" results are obtained as long as the number of allocate/free cycles is below a certain threshold. In the case where memsize is 512KB and blocksize is 128B, this number is about thirty. The correctness of the results is determined by the previous Table 1.

-s 512KiB	-b 128B						
n\m	1	2	3	4	5	6	7
1	✓	✓	✓	✓	✓	✓	✓
2	✓	✓	x	x	x	x	x
3	x	x	x	x	x	x	x

Table 2: Whether or not my_malloc() and my_free() results in a "correct" result for memsize = 512KiB and blocksize = 128B.

-s 1024KiB	-b 1024B						
n\m	1	2	3	4	5	6	7
1	✓	✓	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓	✓	✓
3	✓	x	x	x	x	x	x

Table 3: Whether or not my_malloc() and my_free() results in a "correct" result for memsize = 1024KiB and blocksize = 1024B.

-s 4MiB	-b 16KiB						
n\m	1	2	3	4	5	6	7
1	✓	✓	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓	✓	✓
3	✓	✓	✓	x	x	x	x

Table 4: Whether or not my_malloc() and my_free() results in a "correct" result for memsize = 4MiB and blocksize = 16KiB.

-s 16MiB	-b 64KiB						
n\m	1	2	3	4	5	6	7
1	✓	✓	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓	✓	✓
3	✓	✓	✓	✓	x	x	x

Table 5: Whether or not my_malloc() and my_free() results in a "correct" result for memsize = 16MiB and blocksize = 64KiB.

-s 16MiB	-b 4KiB						
n\m	1	2	3	4	5	6	7
1	✓	✓	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓	✓	✓
3	✓	✓	✓	✓	✓	x	x

Table 6: Whether or not my_malloc() and my_free() results in a "correct" result for memsize = 16MiB and blocksize = 4KiB.

-s 128MiB	-b 16KiB						
n\m	1	2	3	4	5	6	7
1	✓	✓	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓	✓	✓
3	✓	✓	✓	✓	✓	✓	✓

Table 7: Whether or not my_malloc() and my_free() results in a "correct" result for memsize = 128MiB and blocksize = 16KiB.

The previous tables show how the ackerman function is less likely to run out of memory (i.e., give "incorrect" results) as memsize and blocksize are varied. As the numbers increase, more of the values are correctly obtained (more of the table is filled in).

Efficiency Analysis

In terms of time efficiency, when memsize and block size are chosen to be sufficiently large (i.e., large enough to ensure that correct results are obtained), the my_malloc() and my_free() functions perform **slightly faster** than the original malloc() and free() functions.

(n,m)	(3,6)	(3,7)
malloc()/free()	10.8s	44.1s
my_malloc()/my_free()	10.7s	43.5s

Table 8: malloc()/free() vs. my_malloc()/my_free() times when $n = 3$ and $m = 6, 7$.

For space efficiency, on the other hand; because the technique that I used relies on storing the block headers within the block of memory itself, there is a strong dependence on header size. To be specific, if memsize is always a power of two, $M = 2^m$; and blocksize is always a power of two, $B = 2^b$. This means that the maximum number of headers possible is $N_B = \frac{N}{B} = 2^{m-b}$. Now, if the size of a header is H, the following is true:

$$SpaceGivenToUser : TotalSpaceNeeded = \frac{2^m}{2^m + H \cdot 2^{m-b}} = \frac{1}{1 + H \cdot 2^{-b}} = MemoryUsageRatio.$$

From Table 9, we can see that H, the header size, directly affects how small of a chunk of memory can be efficient.

	b	1	2	3	4	5	6	7	8	9	10
non-ideal case,$H = 24$	$\frac{1}{1+24 \cdot 2^{-b}}$	0.08	0.14	0.25	0.40	0.57	0.72	0.84	0.91	0.95	0.97
$H = sizeof(Header*)$	$\frac{1}{1+8 \cdot 2^{-b}}$	0.20	0.33	0.5	0.67	0.80	0.89	0.94	0.97	0.98	0.99
ideal case, $H = 1$	$\frac{1}{1+2^{-b}}$	0.67	0.80	0.89	0.94	0.97	0.98	0.99	1.0	1.0	1.0

Table 9: Memory effieciency ratio as a function of header size and basic block size.

When the header is idealized to 1B, nearly any basic block size yields an "efficient" memory usage ratio. But, when the header size is, for example, 24B; the lower basic block size schemes are extremely prohibitive. **The essence of this idea is to keep the basic block size much larger than the size of a header.** If this assumption is made, then the implementation will result in a relatively efficient performance.

If I were to redesign this system, I would do this: instead of storing the entire header in the memory blocks that are being allocated, I would only store a pointer to such a header. This way, the effective size of a header will always be the size of a pointer. With this setup, it would be (relatively) much harder to make the basic block size smaller than the size of a header.