Memory in C

Unknown macro: {style}

cite

Unknown macro: {font-family}

Introduction

In C, programmers are required to take care of memory management themselves. It is their job to free memory before losing the last pointer to an address (ie, avoid memory leaks), while being careful not to free it ahead of time, which would create dangling references.

Time Complexity of *—malloc*

Since there are no unexpected garbage collections, if the user never frees memory, then *—malloc* is a simple operation. C keeps track of the heap (the memory area where data structures created with *—malloc* are stored) with a free pointer. As long as nothing has been freed, all of the memory before the pointer is in use, and everything after is free. A call to *—malloc* simply moves the free pointer forward by the specified amount, and returns the original location. This sounds like an O(1) operation if moving the free pointer were simply adding a value to some pointer. By running some experiments, it is clear that allocating memory is actually O(n), at least as long as you keep your memory needs low.

If you want to allocate 100,000 bytes of memory, it is no faster or slower to allocate it into 1,000 10-byte chunks than 10 1,000-byte chunks, as shown in the following table:

size per malloc	total time	time per malloc
10	4 ms	.0004 ms
100	4 ms	.0040 ms
1000	4 ms	.0400 ms
10,000	4 ms	.4000 ms
100,000	4 ms	4.000 ms

When a C program begins execution, the operating system obligingly gives it a certain amount of heap space. If you need more, then the program must ask for more, and then wait for the request to be fulfilled before continuing execution. In our case, when we begin to request 10^10 bytes, the computer becomes sluggish as the program runs, and it takes about 3 min longer than expected.

total size allocated	total time	time per malloc(1000)
10^6	1 ms	.00100 ms
10^7	12 ms	.00120 ms
10^8	108 ms	.00108 ms
10^9	1050 ms	.00106 ms
10^10	192405 ms	.01924 ms

Affects of *—free* on *—malloc*

The prior discussion was simplified by the fact that memory, once used, was never freed. When a block is *—free* d, its coordinates are stored in a data structure so that it may be used for a future allocation. This structure has several bins corresponding to different sized blocks. When *—malloc* is called, it first looks in the appropriate bin to see if there is an available block. If it fails, then it will instead just put it at the free pointer's location, as described above.

The first experiment simply adds a *—free* command to the allocation loop. This makes the smaller sizes generally have constant time complexity. On the first computer we used, 1,000 bytes takes longer. We suspect that this is near a bin boundary, so perhaps it takes longer to find the previously used address.

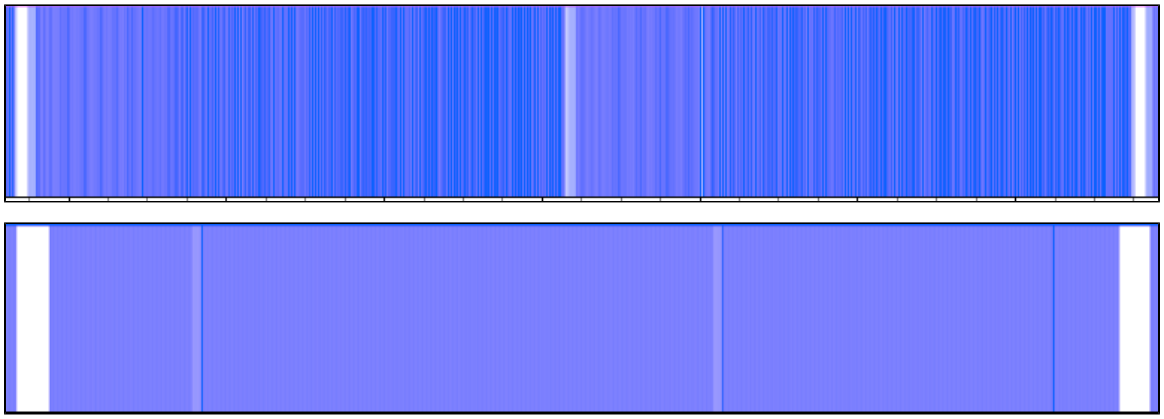
The following table was created using timeb for the first two data columns, and Saturn for the second two. All times are in milliseconds.

	50,000 trials		500,000 trials	
size	**time per without freeing**	**time per with freeing**	**total time without freeing**	**total time with freeing**
10^1	0.000080	0.000080	36.15	35.78
10^2	0.000200	0.000080	27.52	35.95
10^3	0.001100	0.000160	28.45	35.31
10^4	0.004140	0.000080	17106.55	35.87
10^5	0.004960	0.000080	49246.22	42.27
10^6	0.414780	0.000700	70571.28	216.10
10^7	0.413360	0.079460	50573.43	651.18
10^8	0.414920	0.947320	52254.86	1295.20
10^9	5.649460	0.052200	50361.75	1305.01

Our second experiment allocates different sizes. First we had it rotate among four different sizes. When the sizes are similar (30,40,50,60 or 300,400,500,600) it is very easy to see that the average time to allocate and each chunk (.00018ms and .00022ms, respectively) is longer than the time allocate and free a constant size (.00008ms for 60 and .00016ms for 600).

Lastly, we increased the range and number of sizes. The size to allocate is the square of the loop variable, mod 1000. Thus the sizes vary from 1 byte to 100,000 bytes fifty times over the course of the program. We tried both starting at 1 and building to 100,000 and starting at 100,000 and decreasing to 1. We used Saturn so that we could see where the time is spent.

The graph shows the amount of time spent allocating and freeing for each time through the loop (the x-axis is time, and the color indicates whether or not the program is in the allocation function or not). The wider the blue bar, the more time is taken for a single allocation. The white space is the time spent incrementing the loop variable and computing the modulus and square. You will notice that there are some wide white bars. These occur when we switch from 100,000 bytes to 1 byte (or vice versa). Our best guess is that this switch requires a shift in the cache content. It is surprising that this would occur during the loop rather than in the allocation step, so perhaps there is something else happening.



The top graph shows the time distribution when the size increases (and then jumps back down to 1 byte), and the bottom is for decreasing sizes. Both show the time taken for 2000 iterations, though the bottom takes half the time.

The top graph illustrates that larger blocks of memory in general take more time to allocate since the bars generally get wider as time progresses. This distinction is lost in the second graph because sometimes the chunks all use the same address. That is, the first time through, 100,000 bytes are allocated and freed. The next allocation needs fewer, so it can use the same address. This continues until we pass below 500,000 bytes. Presumably this is a bin edge. A new address is used for all chunks down to 250,000. A third address takes care of sizes down to 130,000. The remaining allocations alternate between a fourth address and moving to a new location. Then the next cycle repeats the process, using the same first-fourth addresses, and one of a couple sets of alternate addresses. In contrast, the top graph's program had to keep getting new addresses the first time through, so that there are *many* possibilities in the freed memory data structure, which are difficult to sort through.