Implementing Memory Allocators and Testing Their Efficiency

Ranjani Ramanathan

16 December 2019

Abstract: This project is a simulation comparison of three different methods for coalescing adjacent blocks of memory: FirstFit, BestFit, and NextFit. Three separate programs were created for each allocator. They were tested to see which ones were able to request the most memory and which ones took the least amount of time. The findings were that BestFit made the most efficient use of memory in a particular scenario, NextFit spread out fragmentation, and the timings were too varied to conclude which method had the best performance time.

Introduction: In Assignment 3, students were asked to create a create a simulation in C for an operating system's memory allocator. It did successfully allocate memory whenever a free chunk was available, but there was a lot of fragmentation. Fragmentation is the inability to use memory that is free. External fragmentation is when there are free memory chunks that add up to a total of a lot of free memory, but they are too small/spread apart to allocate larger memory chunks. Internal fragmentation is when memory is allocated, but the object inside the memory is too small, leaving the rest of the memory wasted. In assignment 3, my group made a free header of that memory so it did not have internal fragmentation. External fragmentation was a problem for my group, which is the main type that my final project is going to solve. There are multiple reasons fragmentations are caused by programs. The first isolated deaths, where one chunk of memory frees while it is adjacent to allocated memory. Also, programs can change the types of allocations midway through, for example, a different method could run and it could transition from requesting small memories to requesting large memories [1]. The main goal of the project is to allow as many requests as possible to be satisfied by the memory allocator. To do that, I implemented programs of three fitting methods; coalescing with first fit, best fit, and next fit. The memory allocator were tested with random sequences of allocate and free calls with different sizes, to see how many allocation requests were able to be accepted.

There is one fallacy: the sample memory allocations that I create do not resemble a computer, this is a random allocator. In computer workloads, allocator policy depends on program behavior. For example, real workloads resemble a Markov Model, which means that memory usage and free times all have probabilities based on previous memory sequences and lifetimes [1]. Probabilistic programs are not sequences of few events like the simulator I will create, but they are "large scale patterns involving many events" [1].

Also, it would be helpful if there was data to see which methods take the most or least time. It is unlikely that the results will be significant because the overall program takes miniscule time, but the software does have a time tracker.

Method (Implementation Ideas):

Coalescing: I would like to combine adjacent free blocks. Combining will increase the size value in the header, so it will be able to allocate memories of larger sizes. There are some changes for the free list in order to accommodate coalescing. It needs to keep the free list in order instead of adding to the end of the list like Assignment 3 because coalescing needs to find adjacent headers by using pointers.

Best Fit: I would like to allocate memory at the block that fits the memory size the most closely so that it minimizes the sizes of leftover memory. There is a new variable called "closest". The program needs to loop through the whole list to store in that variable the pointer of the free header with the closest size to the size needed to be allocated. This one also involves coalescing. However, I predict this method will take the longest because it has to search the entire list every time before allocating memory. However, an exhaustive search is also the worst case for First and Next Fit in cases where the appropriate memory is at the end of the list.

Next Fit: This is like FirstFit with coalescing, but it will start the next allocation where it left off so that there is not a large external fragmentation concentration at the beginning, a common problem seen in FirstFit. This one has a new pointer called "begin". After allocating memory, it will place it at the free header afterword to start from there.

Method (Procedure of software):

FirstFit/Coalescing: The method for allocation is similar to Assignment 3. It loops through the memory, and allocates memory at the first free header whose size is appropriate. In coalescing only, whenever a new header is made, starting from the beginning of the free list, the head and the header after it will both will loop till head is above and header after it is below the new free header, and mark the two “front” and “tail” respectively. There is one special case where the new memory is at or above the head, where the new memory becomes the head and the free space after it becomes the tail. Once the head and tail are at their places, the program will use address addition. It will coalesce the new if the front’s address plus the size equals the address of the new, or coalesce the tail if the new header’s address plus the size equals the tail. Lastly, it will give new pointers because the coalesced memory would be null. If coalescing fails, it will simply connect the previous to new to tail.

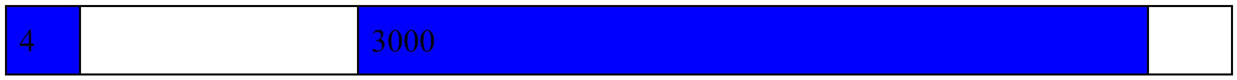
BestFit: From the head, it will loop through the same way as the FirstFit, but it will not stop when the size is okay. Instead, it will store that header in the variable closest if the size difference between the free header and the allocation parameter size is smaller than the header already in closest. Whenever remembering closest, it will also make a marker to the header before the closest so that it can connect that to the next in the free list. This one attempts to coalesce as well, but with the header marked closest, the one before it, and the one after it.

NextFit: This one has a global variable called “begin”. It needs to be global because it is a placemark in the memory for remembering where the last allocation was, and it needs to be used in multiple methods. Specifically, it needs to be marked after coalesce but found in the alloc method, and it cannot be reset at the beginning of methods. Like the FirstFit method, it loops through the memory to find the first free header above the size of allocation request, and then allocates right away. It also coalesces the same way as FirstFit. After, however, the begin placemark will be placed on the header right after the allocated memory. Then, the next looping for the alloc method won’t start at the beginning of the list, but at the header right after the last memory allocation. If it reaches the end of the list, it will go back to the beginning and loop until it finds an appropriate memory size or it reaches the begin marker.

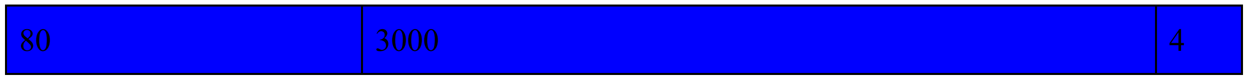
Timer**:** There will be two clock objects, one to record the amount of CPU ticks when the program starts and one to record the amount of CPU ticks when the program ends. The time returned will be the difference between the two clock objects in the unit of CPU ticks, because seconds are too large a unit.

Results: The table is time in CPU ticks for each of the fit methods, each ran three times and the average was taken. Each table represents a different series of memory allocations. The blocks are memories, and the blue sections are allocated and white are free. (Drawings are not to scale).

	NextFit	BestFit	Coalescing
Time 1	101	112	90
Time 2	102	111	99
Time 3	110	138	93
Average	104.333333	120.333333	94



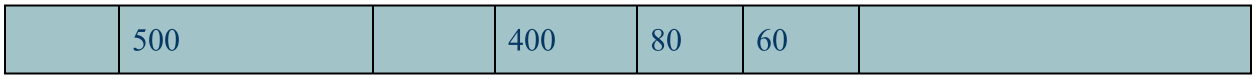
(Coalescing)



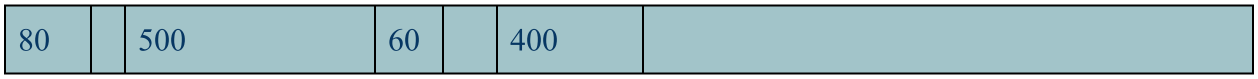
(BestFit and NextFit)

This request series was testing BestFit in particular. There were two chunks available after the middle big chunk was allocated, the big one, 80 bytes, at the beginning, and small one, 20, at the end. The requests came as 4 bytes then 60. Coalescing failed because it allocated 4 bytes at the 80 chunk and the 60 was unable to be allocated. BestFit, however, chose the smaller chunk to put 4 bytes in, and because the large chunk was still available, it put 60 bytes.

	NextFit	BestFit	Coalescing
Time 1	118	134	93
Time 2	95	138	106
Time 3	88	198	129
Average	100.333333	156.666667	109.333333



(NextFit)



(BestFit and FirstFit)

This was testing NextFit in particular. After the 2500 then 400 bytes were allocated, NextFit put 80 and 60 where allocation left off, which is after 400. However, BestFit and FirstFit put the memories in the order of free spaces available. The memory under BestFit and FirstFit shows the problem: there is too much small memory fragmentation at the beginning. NextFit spreads the fragmentation out.

Conclusion:

I learned that there was a lot of inconsistencies when the clock object timed the allocation request series. The method that performs the quickest depends on the memory request series. There are a few reasons for this. In addition to the time looping through the list to find requests, which is what the timer is supposed to test, there could be other chores that make the time variable, such as checking, coalescing and pointing memory. Also, the series requested were so short that differences did not seem to extend time. However, NextFit does seem to take the longest, unlike the predicted BestFit.

Secondly, NextFit and BestFit were successful in solving the Coalescing method's shortcomings. BestFit was able to allocate memory that FirstFit was not because it made sure that there were as many large size fragmentation blocks available. When NextFit allocated right after the allocation left off, the fragmentation was more evenly spread out. In the future, faster and more realistic allocators could be made. For instance, there is the Binary Allocator, where block sizes are multiples of 2 machine words, and the addresses are in binary notation. Blocks are made smaller by splitting in half, and are made larger by doubling small ones. They are in binary size, coalescing is done quickly. There is no address finding needed. Since the freed block's mate is the same size, it checks the address of the freed's size words above and below it to see if there is another free block the same size. For example, if the newly freed with size 2 is at address 011001, the mate after it is at 011011 which is two bits greater [3]. It is also recursive, so that when large blocks are made after coalescing, they combine with other same size blocks near it. The second type is the Cache Allocator, which focuses more on preserving the data in the memory. "The cost of initializing and destroying the object exceeds the cost of allocating and freeing memory for it"[2], because there is a high cost in creating and removing data for the object when it is allocated or freed. When an object is in the cache, all the data is stored in there, so it can be freed and re-allocated without creating and destroying the object and the data in it. The cache also contains memory management policies for that particular object so that it can allocate and free quickly and in a consistent procedure [2].

References

[1]Wilson, Paul, et al. 1995. Dynamic Storage Allocation: A Survey and Critical Review. *International Workshop on Memory Management* (September 1995): 2-72. This is the main article that I will use. It has the most pages about memory management. The main sections are analysis of fragmentation (types of fragmentation, causes and their probabilities) and the types of allocators (describing headers and fit methods).

[2] Bonwick, Jeff. The Slab Allocator: An Object-Caching Kernel Memory Allocator. This describes how to make the memory management more efficient by caching the most frequent sizes. It then goes on to describe the interface of the cache, and tells an example using the mutex variable, and then compares performance of that with the regular.

[3] Knowlton, Kenneth. 1965. A Fast Storage Allocator. *Communications of the ACM,* 8:10 (October 1965). This provides a special method on how to chunk the memory in binary, or in halves. It gives an overview of a regular free storage list, and tells how the binary strategy delves out of that.