COSC1112/1114

Operating System Principles

Semester 2, 2019

Assignment 1 – Segmented Memory Allocator

Name: Duncan Do

Student Number: s3718718

Contents Page

Solution Design

In order allow the allocation of data into memory through 3 different strategies; all three were separated through an if, else if, else if architecture. The dependant variable being the 3rd argument passed into command line. Each strategy itself uses a different if statement distinguish them (according to the kind of strategy they are for). The body of each block itself; are very similar. However, that common code was not extracted into an external method due to the certain variables that each need to access.

Despite the strategy being decided on the command line; because the initial allocations to memory don't rely on the strategy (allocating while the freed list is empty), that logic is separated and placed before the switch case for the strategies.

Experiment Choices

First course of action was to allocate many elements to test with. However, to test features such a worst fit, I required different sized elements to allocate. Thus, I allocated 3 different data types. Bools, which are 1 byte long, integers which are 4 bytes long, and long long ints where are 8 bytes long.

Initially, I was planning to use a random number generator to randomly select which of the 3 data types to allocate. However, due to the desire for consistency between tests; this was changed to allocate an amount of each data type, one at a time (all the integers, then all the bools, then all the long long ints). This structure was also used when re allocated elements to memory after some de allocation. **Data set used in appendix 1**

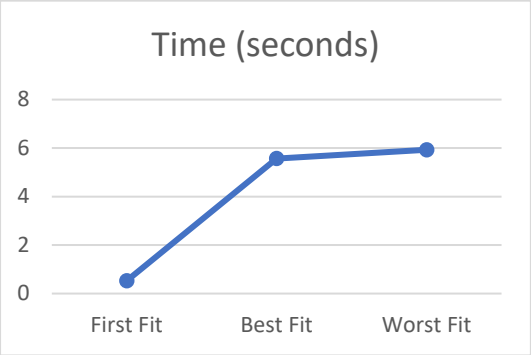
For deallocation, only half of the elements were de allocated. In addition to this, for the sake of randomness; an if statement was used to determine whether to de allocate or not. The condition being whether the iterator running through the allocation list was even (mod % 2 = 0). This does not create real randomness but does allow elements of each data type to be deallocated.

The reallocation is where the strategy is used. Thus, a timer encapsulated the loop where the re allocation occurred. Further results include the amount of space left in the freed list after the re allocation strategy.

Results

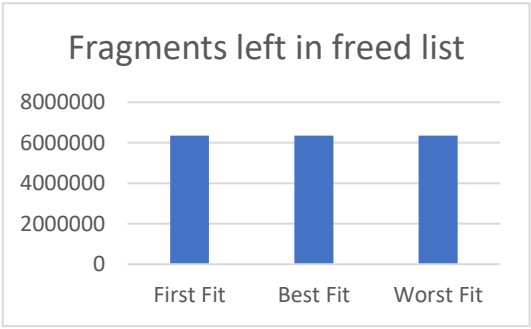
For the sake of consistency, all tests were run with the same data. Meaning the same elements allocated and deallocated, in the same order. Furthermore, the test was run multiple times with varying sizes for the data set; despite large variance in data set size (100, 100000, 1000000 etc); the relative results of each strategy in comparison to each other was negligible. The graphs below are from the data set with an initial 90,000 allocations.

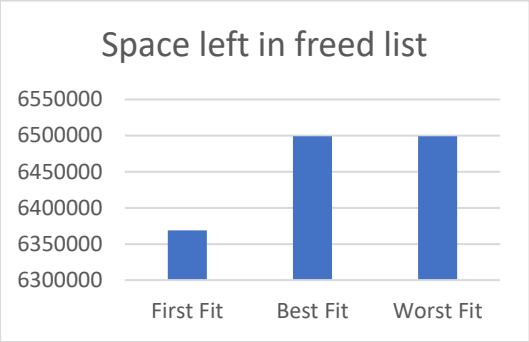
Initial Allocations	90,000
De-Allocations	45,000
Re-Allocations	30,000
Data types used	Integer (4 Bytes) Bool (1 Byte) Long Long Int (8 Bytes)



In terms of how long each strategy took to re-allocate 30,000 elements (10,000 of each listed data type), best and worst fit were relatively the same; with best fit taking approximately (over multiple runs) 5.57 seconds and worst fit taking approximately 5.93 seconds. The reasoning for such a similar result is the fact that both strategies must iterate through the entire list of allocated memory "chunks" to find the "best" and "worst" fit respectively. First fit's time will vary depending on the sizes if the elements in the list of freed memory "chunks". However, unless the only large enough memory "chunk" is at or near the end of the list; first fit will always be drastically faster in relation to best and worst fit. This is because the strategies is able to stop/break once it finds a single suitable memory "chunk"; the other two strategies cant guarantee that the first, second, or third (etc.) suitable memory "chunk" is the most/least suitable until it reaches the end of the list.

Due to the stagnant data, all 3 algorithms produced the same number of fragments in the freed list. This suggests despite the differing algorithms, the amount of times a freed memory chunk was split (part used to allocate the data and part put back into the list of freed memory) was consistent across the board. With the use of a random number generator to further vary the allocations, some variance might occur. However, the results from the multiple data set sizes suggests it will be minute.





The resulting data from analysing the amount of space left in the freed list after each strategy suggests that the increased time for best and worst fit is not worth the processing time as it results in more wasted memory space. Despite seeming like matching the space to near exact size in best fit would result in less memory space left in the freed list (opposite to worst fit), the results say otherwise. Perhaps with extremely high variance and randomness in the data set, the data can be skewed more.

Data tables for graphs in appendix 2

Discussion

List Size Growth

The solution was built on the design that the program break would be pushed back by the size of **each** allocated element **every** time one was allocated. This resulted in the list size (cumulative size of all elements) to grow in exact respect to the size of each element added.

List Size Growth cont. + Fragmentation

An alternate approach can be suggested, one where a page of memory (1024KB) is added at a time and only when the list cannot fit the latest "to be added" element. The use of paging would theoretically reduce the amount of external fragmentation of memory chunks but give risk to internal fragmentation.

The cause of fragmentation in this solution is the desire to not waste space in freed memory "chunks" if the "to be allocated" element can fit into part of the freed memory; saving the remainder for another allocation.

Despite 3 different strategies, the amount of fragmentation by the end was consistently high. A suggested solution to reduce the severe amount of fragmentation is to combine freed memory chunks to fit incoming allocations. Meaning, if there is freed memory in the list, and no single memory "chunk" can hold the size of the incoming "chunk", then as a last resort before simply appending the new chunk to the end of the allocation list; the locations of the freed memory "chunks" would be inspected to determine if any were adjacent to each other. If so, then they could be combined into a single memory chunk.

This would reduce fragmentation and instead do the opposite, combine 2 or more "chunks" adjacent in memory and reduce both the wasted space and fragments at the end of the re-allocation.

Left Over Memory

Fragmentation in turn becomes a contributor to the amount of wasted memory. The worst fit strategy compounds this waste by searching for the memory "chunk" of the highest size, resulting in larger left-over fragments in freed memory (in relation to the other 2 strategies).

In comparison to that, best fit theoretically (in a real-world circumstance with unpredictable amounts and sizes of "to be allocated" elements) result in the least fragmentation. However, the amount of waste would not differ much from worst fit. In worst fit, the wasted space comes from the fragmented memory "chunk" that the "to be allocated chunk" does not need; whereas in best fit, the equivalent wasted memory comes from the larger memory chunk that was simply not used. The waste comes from different places, but the amount of waste is relatively the same.

The only strategy out of the 3 that could differ in resulting waste is first fit because it does not follow the same type of condition as the other two (however negligible it might be).

Conclusion

When it comes to managing memory in operating systems its always important to establish the time vs (waste of) dynamic. Which is valued over the other; realistically it is unlikely both can be satisfied, usually one must give for the other to excel. The determinate for this relationship is usually the business decision of the system users or the product owner.

Furthermore, a tipping point to push for one value over the other is cost. Time, cost, and resources create a delicate triangle where none can be fully satisfied. One will affect the other in a complete cycle, however both time (to allocate/deallocate) and (waste of) resources can be affected by the cost (budget) of development.

From the results of the experiment, the 3 suggested strategies yield similar results with first fit edging out the other 2; mostly due to its speed over best fit and worst fit. In other criteria, first fit either exists within the same relative range as the other 2 strategies or is the exact same. Considering the static data used to test; some results could fluctuate minorly.

This, however, does not mean that the suggest first fit strategy is the best algorithm. There are many other strategies that could be tested to find an overall “winner”; such as last fit and many others. In addition, as stated in the analysis; there is room for numerous improvements to the three strategies; these same suggestions can be applied to any future strategies.

It is also important to consider the context of the allocation. For example, if it is known that all data will be of the same type and therefore size; first fit is heavily preferred over the other 2 as iterating through the entire list would be pointless (best fit would be the same as first fit and the same as worst fit). Thus, the ability to break early would save time with no other drawbacks. However, when the memory sizes fluctuate, it can be argued that best or worst fit is preferred. The choice between best and worst fit can be better concluded from further testing with a larger set of criteria to determine an extensive list of pros and cons.

Appendix

Appendix 1

Initial allocation	90,000 elements - 30,000 Integers - 30,000 bools - 30,000 long long its
Deallocation	45,000 elements
Re-allocation	30,000 elements - 10,000 Integers - 10,000 bools - 10,000 long long its

Appendix 2

Time comparisons of strategies.

	Time (Seconds)
First Fit	
Best Fit	
Worst Fit	

Wasted fragment comparison of strategies.

	Fragments left in freed list
First Fit	6349000
Best Fit	6349000
Worst Fit	6349000

Wasted memory space comparison of strategies

	Space left in freed list
First Fit	6369000
Best Fit	6499007
Worst Fit	649907