Joshua Silva
ECE 2230
Spring 2024
MP4

Performance Evaluation

- Data Gathered From Unit Driver 4 with Different Conditions

Unit Driver 4 for Perf. Eval.			
Search Policy	SearchLoc	Runtime (ms)	Free Blocks
First_Fit	Head_First	0.869	9
First_Fit	Head_First	0.934	9
First_Fit	Head_First	0.836	9
First_Fit	Rover_First	0.898	9
First_Fit	Rover_First	0.932	9
First_Fit	Rover_First	0.866	8
Best_Fit	Head_First	0.955	8
Best_Fit	Head_First	0.951	7
Best_Fit	Head_First	0.89	7
Best_Fit	Rover_First	0.953	8
Best_Fit	Rover_First	0.915	8
Best_Fit	Rover_First	0.888	8
Worst_Fit	Head_First	0.925	9
Worst_Fit	Head_First	0.905	9
Worst_Fit	Head_First	0.914	9
Worst_Fit	Rover_First	0.954	9
Worst_Fit	Rover_First	0.9	9
Worst_Fit	Rover_First	0.987	9

- Testing Setup & Methodology
  - Unit Driver 4 was used to test out Search Combinations. Although initially used to prove the special case that allocations and frees can be done at random and still maintain functionality, the greater complexity of the driver makes it more suited to test out the differences between search locations and policies.

I used a random number generator and inserted them into random allocations, such as 2 * random & ¼ random, to better replicate random situations the user might put the system through. The driver allocates 4 different times, with frees scattered about, then allocates another 4 times, again with frees in between. This is done to better test out the search policies, as there will be more free blocks that could potentially hold new memory & show the benefits of certain search policies.

From there, I ran through each combination of Search Policy and Combination to test out the runtimes and the amount of free blocks at the end of runtime. Runtime is gathered to determine if there is any hindrance to performance with any searchtype. Free Blocks are recorded to see if a Search Policy is better at managing memory & preventing fragmentation than other methods.

- Search Policy Comparison
  - Looking at our data above, we can see differences in effectiveness between each search policy. Best_Fit is the most effective way to manage our memory, as it is regularly able to make use of existing free blocks better than its counterparts, leading to less free blocks and better preventing fragmentation.

    First_Fit, according to our data above, is not the best at preventing fragmentation, but it has a slightly better runtime than its counterparts. It is somewhat hard to measure this, as the complexity is O(n) due to its sequential nature (meaning we would need many more allocations to better test runtimes), we can still see that First_Fit runs about 0.1 milliseconds faster than than other Search Policies. This is probably due to the fact that First_Fit looks for the first match, so it could have a time complexity of O(1) even with a large free list, while the other search policies will always have a time complexity of O(n) due to having to search for the best/worst free block.

    Worst_Fit is the worst search policy. It was unable to ever go below 9 free blocks, our maximum amount for the other search policies. It also consistently had the worst runtimes, never going below 0.9 milliseconds. Although, its runtimes are close to Best_Fit. As said earlier, both Best_Fit & Worst_Fit have to search through the entire free list to find the smallest and largest free block available in the free list.

- <u>Search Starting Comparison</u>
  - Overall, the start of our search had little effect on runtimes or the possibility of fragmentation. As seen in our data above, the runtimes between starting at the head or rover barely differ. Best_Fit and Worst_Fit should not be affected at all, since they will always run through the entire free list no matter what. First_Fit would be the only policy affected by our decision of search location, and our design could have some impact on the effectiveness of first fit.

    Our design takes the allocated memory from the right side of the free block, ensuring that we do not need to update the original header chunk for the free block. We also insert new pages of memory at the beginning of the free list, meaning the largest free blocks will tend to gather at the beginning of the free list. So theoretically, starting at the rover with First_Fit could lead to less chances of fragmentation, as First_Fit is less likely to use the largest pages. This is seen with one of runs with First_Fit with Rover_First, in which it managed to use 8 free blocks instead of 9.

- <u>Conclusion</u>
  - Overall, changes to our Search Policies and Search Locations have an impact on the effectiveness of our free list. Best_Fit proves that it is the best at avoiding fragmentation, being more likely to get rid of smaller free blocks completely, instead of splitting up larger free blocks. Worst_Fit was shown to be the worst at preventing fragmentation, as it consistently had the most amount of free blocks compared to the other methods. First_Fit was shown to have a slightly better runtime, although we would require many more allocations to prove this (besides we might run out of memory if we run too many allocations).

    Search Location had little effect on Best_Fit & Worst_Fit, as they will always have to go through the entire free list. But, due to our design & data gathered, we can see that Search Location is more likely to prevent fragmentation with First_Fit. As it is more likely for free blocks to gather at the beginning of our free list due to our handling of calling morecore, so First_Fit starting at the rover could lead to it searching smaller free blocks first.