Lab 4 Report

Luke Seewald 20663616
Michael Rokas 20668160

Problem Statement

The problem assigned to us for this lab has to do with memory allocation. We are given a fixed amount of space to manage and our control data must be kept within this fixed space. This means that in reality there is less space available to the user as some is used for control. Another restraint is that we must allow for dynamic allocation, where we do not know the size that is going to be needed until it is requested. It is also a requirement that we do not move elements around after allocating. This means that how we choose where to put the data is important because it could cause more or less fragmentation. It also means that in order to make the most efficient use of the space we should coalesce two adjacent free spaces into one block of free space whenever possible. Finally it is important that we minimize wasted space, whether that be from fragmentation or from unnecessary control data (coalescence and choosing a good strategy for choosing where to allocate the block are the biggest assets in minimizing wasted space).

Data Structure and Algorithm Design

The data structure used for control of blocks in both the worst and best fit is a doubly linked list. Each node of the linked list contains the following parameters: pointer to the previous node, pointer to the next node, pointer to the start address of memory it is controlling, size of memory it is controlling and its state (free or allocated). The start address in the memory is always directly after the linked list node, and directly after the block is the linked list node for the next data block. An example of how this structure would appear visually can be seen in figure 1. This choice of data structure has its advantages and disadvantages. One of the pros being that we have faster process times as we store more data that we could do without but for the sake of simplicity and speed we keep the extra data. Every node uses 40 bytes on most machines, this is more efficient compared to a bitmap for large sized memory, for example a block of memory 1MB in size can be controlled by only 40B of memory although at the same time it still requires 40B for small blocks of memory as well, for example a 8B block of memory still requires 40B of memory to control it making it less favorable to a bitmap in this case. There are some small improvements that can be made to this design as well, mostly centered around making the size of a control node smaller. Firstly, if we make the linked list singly linked we would save memory by not requiring the extra pointer, although it would take extra processing time to find the previous nodes. We could also take advantage of the fact that ints can be negative and use positive and negative signs on the size parameter to represent free or allocated which would allow us to eliminate the state parameter to make our node smaller. It would also be possible to remove the start parameter because we know the start address is always directly after the node.

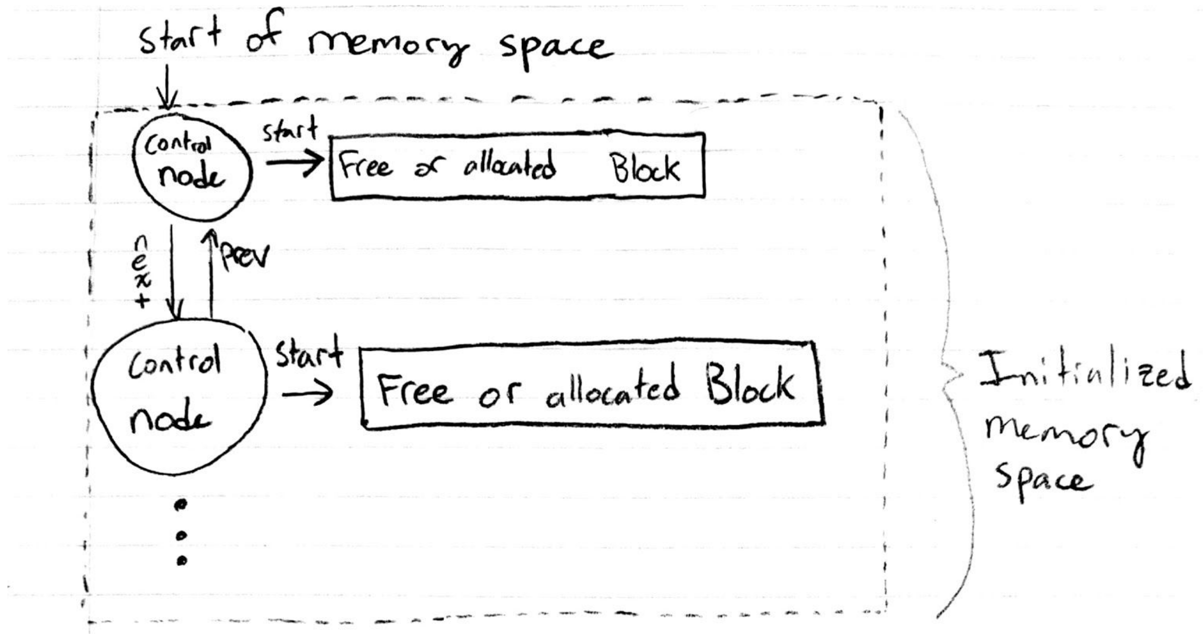

Figure 1: Illustration of the memory control data structure

In our initialization functions we check if the given size is large enough to fit at least one control node. If it is not then return -1, if it is then we set our global pointer to that memory space using `malloc()` of the given size. We then initialize the first node of the control linked list at the beginning of the memory space, setting `prev` and `next` to `NULL`, the size to be the given size subtract the size of a control node, the state to be 0 (free) and the start address to be the start address of the memory space plus the size of a control node.

Both the best and worst fit allocation algorithms work similarly with the exception of how they choose where to allocate new block as later discussed. The first thing we do is check the passed size and make sure it is 4 byte aligned, if it is not then we round up to the nearest 4 bytes (this is done with a ternary operator and some simple math) then we made two local functions to be called by the respective allocation algorithms, `find_best_fit()` and `find_worst_fit()`. In these functions we set up a node pointer called traverse to go through the linked list of all the nodes as well as a node that will point to the best or worst fit space. Here we are not looking for the requested space plus the size of our control data as we are going to return the address of an already existing control node which we can then change the state of. In the `find_worst_fit()`, as we traverse through the linked list we keep track of the biggest free space. If we find a larger free space (that is still larger than the requested size) then that becomes the biggest free space and so on. Once we have reached the end of the linked list, we return the node pointing to the largest free space. If no space was found to be big enough then we return `NULL`. In `find_best_fit()`, we do the same process but instead check for the smallest space that is still larger than the requested size. Once returning from these two functions we do a null check and return if nothing was found. Otherwise, now that we have the node with some amount of free space, we check the edge case where the size of the free space is exactly the size of the requested space. If so, then we just change the state of that node to be allocated and return the start address to the user (which is the address of the pointer plus the size of our control data). Otherwise, we check how much space will be left if we tried to make another control node to keep track of the leftover free space. If this value is greater than 0, then we change the returned node's state to allocated, create the node keeping track of the free space (initialize it with state being free ect.), insert it into the list and then return the start address for the requested space. If it is not greater than zero that means there is internal fragmentation as there is not enough space to make another control node. To solve this we simply give the user this small extra space and the internal fragmentation goes unseen by the user.

The deallocation algorithms are identical for both best and worst fit. The overall approach we took for deallocation was to merge free space from the right into the left if both were free space. This was because if we merged left into right and we merged the start node into another node, we would lose reference to the start node as it is expected to be at the very beginning of the memory space. The user calls the dealloc functions, passing in the pointer to the start address that we gave them. Then we define a pointer to where our respective control node would be, which is the pointer they passed to us minus the size of our control node. We do a null check for safety, if it was null then we return. Next we traverse our list of nodes to see if it is a block of memory that we actually passed to the user, if it is not then we return. If it is, then we check the previous(left) node (make sure it is not null) and see if it is free. If it is, then we merge the two nodes by calling the merge function that we made (we do not set the state of the current node to be free since it will be removed by the merge anyways). In this merge function we set the size of the left to be the size of the right plus the size of a control node (since the right node is going to be removed). Then we remove the node and return the remaining (the left) node. Next, within this same if statement, we check to see if the next (right) node is free as well (also make sure next is not `NULL`) if it is then we merge the two nodes and return. Now, if the previous node was not free, we check the next (right) node to see if it is free. If it was then we set the current node to be free and then merge the two nodes and return. Otherwise if neither the previous or next (left or right) nodes were free then we just change the state of the current node to be free and return.

Our utility functions are the same (only difference is where we start the traverse pointer, at which global variable). We initialize a count variable and a traverse pointer and go through the linked list. Every time we come across a node that is free and the space is less than the passed size, we increment the counter. Once we have reached the end of the list, we return the count.

Testing

We have 9 tests to show basic functionality that are used on both worst fit and best fit to make a total of 18. The purpose of each test can be seen in table 1 below.

Test 1	Fills memory space completely, deallocates two blocks of size 200 and 220 then allocates 100 to show that worst fit puts it in the 220 block (for best fit it goes into the 200 block).
Test 2	Fills the memory space completely and then deallocates the blocks from left to right and shows that as we free the space, it coalesce.
Test 3	Fills memory space completely with 4 blocks, frees blocks 1 and 3 and then frees block 2 to show coalescence.
Test 4	Tests that we do 4 byte alignment and that we can handle not having enough space left for a leftover node so we give it to the user (but don't tell them about the extra space).
Test 5	Allocates 3 blocks of memory then asks for another one which is too large for the space left, shows that we don't fulfil that request.
Test 6	Allocates a bunch of blocks of memory and leaves two blocks under 100 bytes of free space, then runs the utility functions and shows that they work (outputs 2 blocks).
Test 7	Tries to initialize our memory space with 30 bytes, which is less than a control node. Therefore it fails as expected.
Test 8	Allocates some memory and actually uses it for some operations (added two ints and then deallocated one of them, made sure the other didn't change).
Test 9	Allocates some memory and then tries to deallocate a block but gives the wrong start address (it fails as expected) and then tries again with the correct start address (which works).

Table 1: Explains the specifics of each test case

Analysis

In order to analysis the pros and cons of choosing a best fit or worst fit allocation algorithm we ran two main tests. One to show which algorithm provides better performance for large memory allocations and another to show which allows for less amounts of external fragmentation. The details of Each of these tests are shown below.

The first test demonstrates how each of the algorithms perform when allocating data blocks. The data space was originally initialized to be in a state where it was almost empty with the exception of a 4 byte divider of allocated memory splitting the space into two equal halves. This can be seen in figure 2. Then we allocated 16 bytes 20 times, when doing this it can be seen in figure 3 that the best fit algorithm placed all 16 blocks in the first half (leaving the second empty). And the worst fit algorithm dispersed the 16 blocks equally across the two halves. After doing this we tried to allocate a 1kB block, in the best case scenario we were successful as the second half of the memory space could accomodate for a block this large, although the worst fit algorithm was not capable of allocating the 1kB as both halfs had under 1kB of space. From this test it can be concluded that a best fit algorithm will perform better when there is potential of large blocks needing to be allocated.

The reason the best fit was able to accommodate the large data and the worst fit was not is as follows. When allocating new memory a best fit approach will always allocate memory in the smallest space possible(that is still large enough for the requested space), this means that the smaller free memory space will get even smaller although the larger free memory space remains untouched and still ready to hold a large amount of data. On the other hand the worst fit will always put the data into the larger memory space which as seen in this analysis will tend to try to keep all free memory spaces at a similar size, if one memory space is significantly larger it will be used up, so it comes closer to the average free memory size, and the smaller space doesn’t get too small. The advantage of this is discussed in the second analysis.

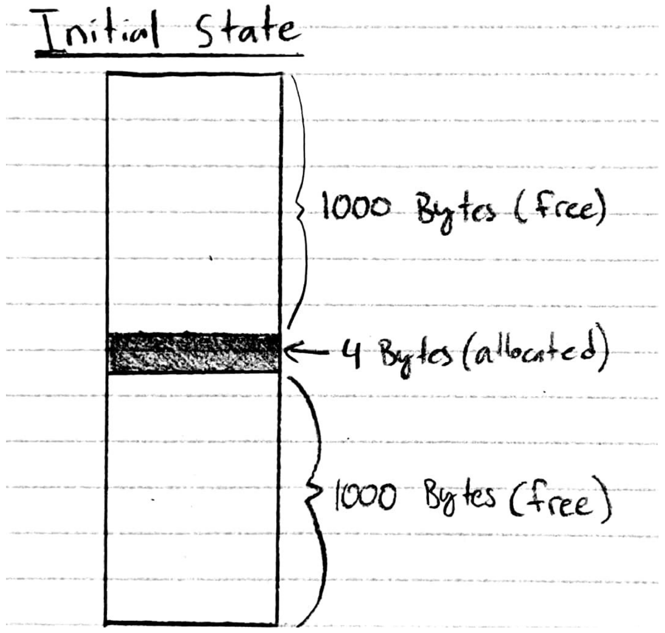

Figure 2: Illustrates the initial state of the memory space for the first analysis test. *(memory for the control is not illustrated)*

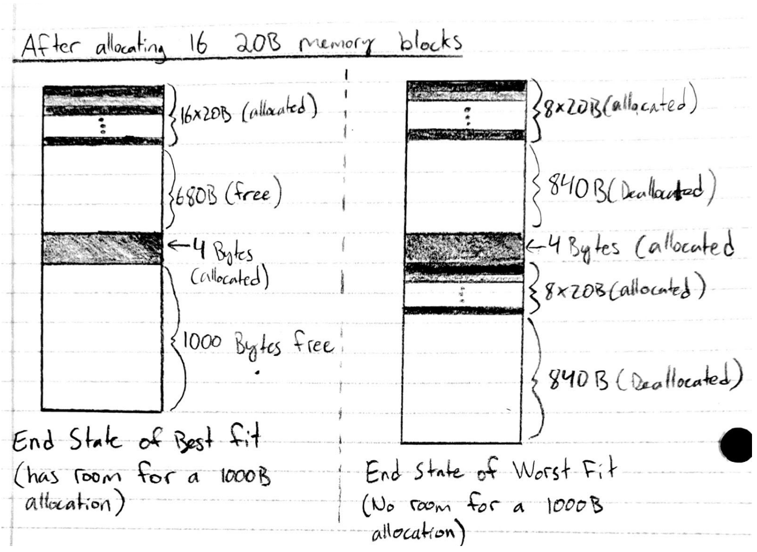

Figure 3: Illustrates the final state of both the best fit and worst fit memory space after the first analysis test. *(memory for the control is not illustrated)*

The second analysis test was designed to show which algorithm produces more external fragmentation (i.e free memory spaces that are so small that they have minimal if not no use to the user). In this test case both the worst and best fit memory spaces were initialized(done via allocations and then deallocating) to have many empty spaces of two sizes, 12B and 100B. This can be seen in figure 4. In total there were four spaces of size 12B and four of size 100B. We then attempt to allocate 8B four times with both the best and worst fit algorithms and they both succeed as can be seen in figure 5. After the completion of these allocations we run the external fragmentation counter on both the best and worst fit memory spaces, looking for memory less than 8 bytes, as there is not much that can be done with 8 bytes aside from declaring simple primitive data types. This test returned 4 cases of fragmentation in the best fit memory space and no fragmentation in the worst fit case. This shows that the worst fit algorithm is better at minimizing external fragmentation.

The reason worst fit provides less external fragmentation is due to the fact it always picks the largest memory space. Best fit will always try to fit the memory into the smallest space possible which can cause memory to fit very tightly into memory spaces, although if the memory does not fit in exactly it can create very small memory spaces as seen here called external fragmentation. These memory spaces are often too small to be used and are therefore a waste of space. The worst fit will always put memory in the largest space so although it shrinks large spaces, the space left over is likely to be large and still usable which means external fragmentation is less of an issue.

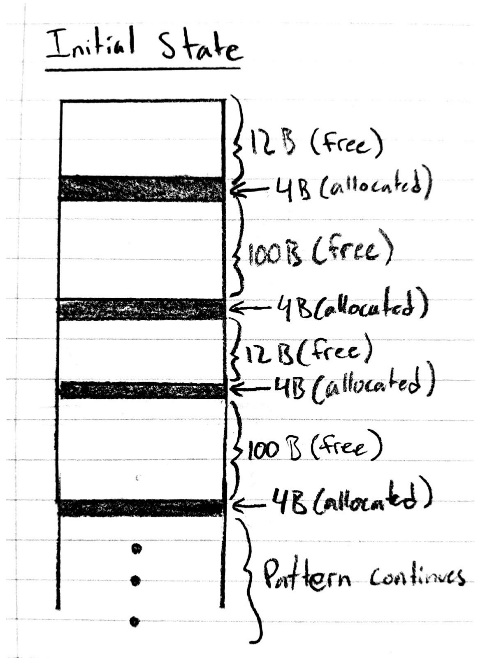

Figure 4: Illustrates the initial state of the memory space for the second analysis test. *(memory for the control is not illustrated)*

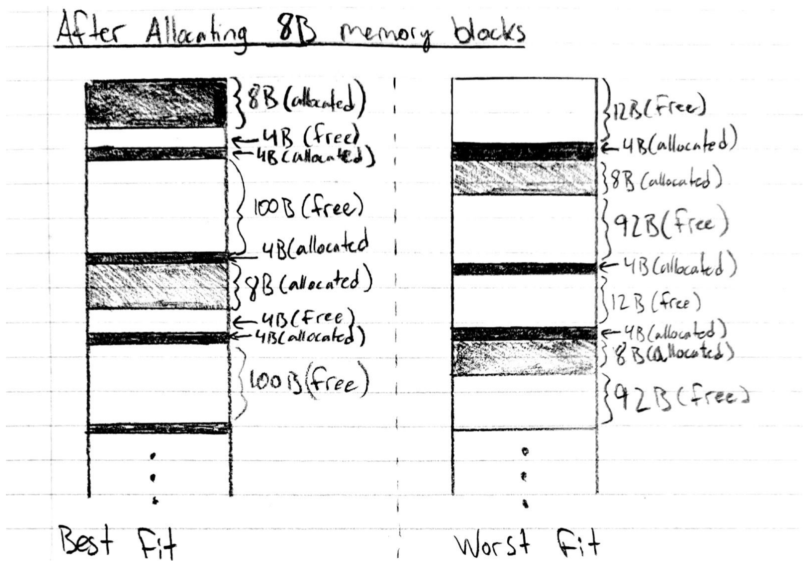

Figure 5: Illustrates the final state of both the best fit and worst fit memory space after the second analysis test. *(memory for the control is not illustrated)*

The final analysis test that was run was the extensive use test. This test makes three thousand calls to allocate and deallocate random sized blocks with both best and worst fit, keeps track of how many times both algorithms failed and how much fragmented data spaces under 16B each algorithm makes. This test was run 500 time and the average amount of fragmentation and failed allocations were recorded. The results are shown in table 2. This large data collection correlates very closely to the previous two analisis. It can be seen at earlier examples the best fit algorithm it created 3 times the amount of fragmented data spaces on average due to trying to fill tight spaces. Although as also shown in previous cases the best fit algorithm failed 12% less than the worst fit test, due to its ability to preserve large sized blocks. This final test was very useful in the confirmation of the previous observations made in our analysis.

	Average number of fails	**Average number of fragmented spaces under 16B**
Best Fit Algorithm	67	3
Worst Fit Algorithm	76	1

Table 2: shows results of excessive use test

Conclusion

There are many tradeoffs to consider when designing a memory allocation system, primarily in structure of control data, and algorithm of choice. Using Dynamically sized control data structures can provide better performance when allocating fewer large blocks but can begin to suffer if more small sized blocks, whereas using a structure such as a bitmap can provide a more consistent performance regardless of usage. When choosing an algorithm it is important to consider how the memory system will be used. If larger memory is to be used for example loading images or large documents into memory a best fit algorithm may be optimal so that more large data blocks can be loaded at once with less of a concern of losing small amounts of data to fragmentation. Although if the data being used is allocated in smaller block sizes and there is less memory space available (i.e a embedded system) it may be ideal to use a worst fit algorithm, as less large data blocks will be needed but losing small sections of data to fragmentation is more costly as there is less memory space.