Operating System Coursework Report

Introduction

In this report, I will be introducing how I implemented malloc and free in the xv6 operating system. In the xv6 operating system, the malloc and free system calls do not exist, so I created my implementation of it using the underlying system call sbrk() for malloc. This report will explain how I implement these system calls and the design choices I made.

The initial steps I took to implement the functionality were to first find an optimal data structure that could hold the information about a memory block. I chose to use a linked list to keep track of memory as linked lists are built piece by piece meaning that the nodes are dynamically allocated so there is no memory wastage when dealing with linked lists. The node contains the memory address of the start of the memory block, the size of the memory space, the size of the memory requested from that block, free status and a pointer to the next memory block.

Malloc

When we first call sbrk(), we get the first point after the end of the uninitialized data segment which is the start of the heap as shown in Figure 1. This point is called the program break, so increasing or decreasing the program break is essentially like allocating and deallocating memory and this is manipulated by sbrk().

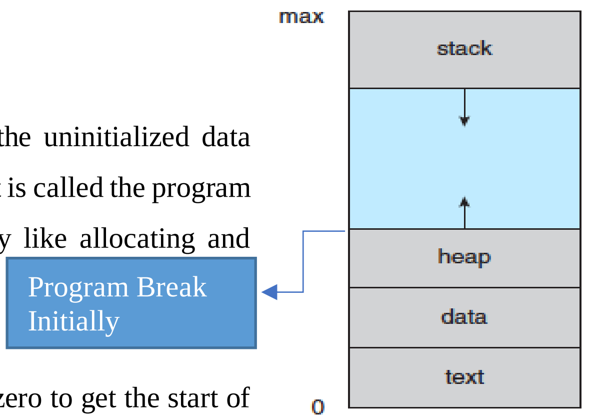

Figure 1 Program Break in relation to Heap in memory

Initially, when malloc is called, we call sbrk(0) with a parameter of zero to get the start of the heap. We then create a node with this value. After that, we call sbrk() again with the parameter of how many bytes we want to allocate. In our case, we would call sbrk(node_size + requested_size), requested_size is the number of bytes the user wants to allocate. This would then lead to the program break increasing by the node_size plus requested_size bytes, therefore, allocating memory. The node is then added to the end of the linked list, and if the linked list is empty then it becomes the head. We then return the node.

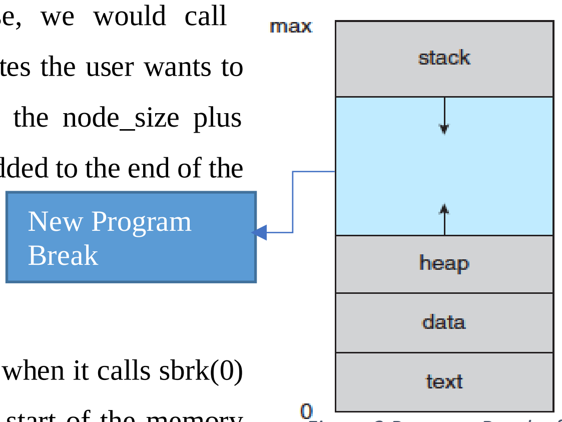

Figure 2 Program Break after sbrk has been called

If we call malloc again then we go through the same steps except that when it calls sbrk(0) now, it will get the new program break's location and set that as the start of the memory block for the new node as presented in Figure 2.

Best Fit Algorithm

When malloc is called, before we start the allocation process, we look for already initialized memory blocks that are free. We use the best-fit allocation algorithm to determine which of the free memory blocks in the linked list should be used. The best-fit allocation has been chosen because it is memory efficient, the program allocates the minimum possible space for the process preventing memory wastage.
The algorithm searches the whole linked list for free memory and allocates the one with the smallest memory block that is adequate.

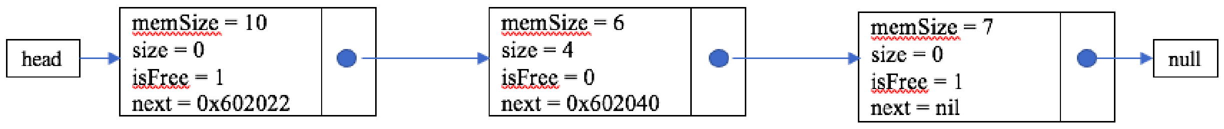

Figure 3 Example of Linked List holding memory

If we call malloc with size 6, then initially the program would go over the linked list see Figure 3. It will first traverse through the first node and check if it's free, if it is then it checks the size difference between the memory space it has and the number of bytes asked for by the user. If it is less than ten thousand then it would set the first node as the best fit node. Ten minus six is four and is less than 10,000 so this node is the provisional best-fit node. Then it traverses to the second node, and checks if it's free, it is not so we skip it. Then we go to the last node and do seven minus six and we get one which is less than four so this is now the best fit node.

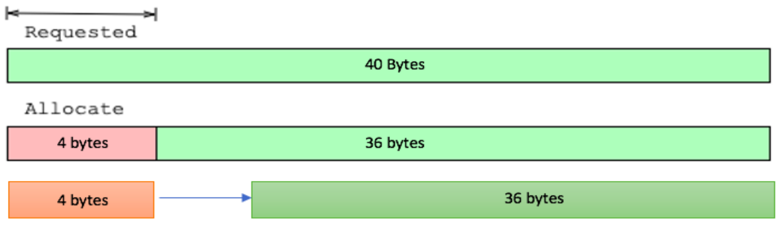

Figure 4 Process for splitting memory

After this process, we check if the best fit node is big enough to be split into two, we check if the node has enough size to hold the requested bytes and the size of a new node with four extra bytes. If it does then it will split the node into two, one containing the exact amount of requested memory and the other containing the rest of the memory. This is done to avoid internal fragmentation and so that most of the memory block is being used up.
The reason both best fit and splitting of memory are implemented it to completely reduce internal fragmentation. As when all the nodes aren't big enough to be split but are bigger than the requested size then the best solution is to pick the smallest memory block out of them.

Free

Free takes in a pointer as a parameter and then it traverses the linked list to find the matching pointer. When found, the mem_block->isFree variable is set to 1 meaning that it is available for allocation now. My implementation of free doesn't actually delete the whole block from memory but rather it just makes it available for use.

When free is called it also searches for contiguous blocks of free memory and groups them together. If we have three memory blocks of 5 and we wanted to allocate 15 bytes of memory normally we would have to create new memory but instead the program groups the three blocks of 5 and allocates it. This makes the mechanism more space efficient and prevents external fragmentation. I implemented this mechanism using recursion, so it goes to the last contiguous free block and adds them, then it works its way back to the first contiguous block making one big chunk of memory.

Reflection

The report went over how malloc and free were implemented in the xv6 operating system. This project taught me a lot about memory management and how the operating system uses the heap to allocate memory to a process. I learnt about the underlying system calls of malloc in brk() and sbrk() and how they work. The most important skill I learnt during this project was how to write efficient code. Taking into account the wastage of memory and the different types of allocation algorithms, I decided that the best-fit algorithm was most suitable for my case. This project also increased my knowledge of working with linked lists and how you can manipulate them. Linked lists are slower than arrays and other data structures but for our program linked list is the most suitable one as we need our list to grow and shrink dynamically so that we are not limited by memory space as we would if we used an array. In the program, we also account for internal and external fragmentation which occurs when the memory allocated is bigger than the memory requested. In our program, we find the best fit memory block and we also split the block to avoid internal fragmentation. When we free memory, the program groups all free contiguous memory together.

If I were to do the program again, I would put more emphasis on time efficiency, as right now my program uses high computation tasks such as the best-fit algorithm and the recursive method to group free contiguous memory. This implementation doesn't allow for fast computation as opposed to if I used first fit which is significantly faster but then would run into internal fragmentation. I learnt that there is no one best solution to the implementation of malloc and there will be tradeoffs in every type of implementation.

References:

- Silberchatz, A., Baer Galvin, P. and Gagne, G., 2022. *Operating System Concepts*. 10th ed. United Kingdom: N/A (Figure 1 & 2)
- Redirect Notice. 2022. Redirect Notice. [ONLINE] Available at: https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.kuniga.me%2Fblog%2F2020%2F07%2F31%2Fbuddy-memory-allocation.html&psig=AOvVaw0N2S7-w3W-kMvLJcacLKqn&ust=1669816603311000&source=images&cd=vfe&ved=0CA8QjRxqFwoTCJCj293F0_sCFQAAAAAdAAAAABAE. [Accessed 29 November 2022]. (Figure 4)