XV6 Memory Management Implementation

Design Decisions

Having no idea where to begin, I read through *Hack the VM: malloc, the heap & the program break*[1]. After reading this I had a rough idea of how malloc works, which system calls will be of use and consequently I made a start.

To begin with, I implemented a simple malloc function. It had no way of checking for free blocks in the heap, it simply made a call to sbrk() to add more memory to the end of the heap. This solution makes no attempt to optimize memory usage. I needed a way of keeping track of blocks.

I then read through *OSTEP - Free-Space Management*[2], which introduced me to new ideas - Using a linked list to keep track of free blocks, different memory allocation algorithms and the coalescence of contiguous free blocks.

My first instinct was to implement this “Free List” idea, but I came to the conclusion that coalescence would become complicated when trying to work out which free blocks are contiguous. I instead used a linked list that contains *all* blocks, it contains a flag that tells us if the block is free or allocated. Each block points to the next contiguous block, thus coalescence is easier to implement.

Once I had implemented a linked list, I had l to decide on an allocation strategy. After another read through of *OSTEP - Free-Space Management,* I settled on the *First Fit* method. The main reason was speed, since we don’t have to search through the whole list every time we wish to find a suitable free block. The problems with this method arise when the beginning of the list fragments into lots of smaller blocks leaving unusable gaps. Since our blocks are kept track of in a linked list contiguously, we don’t have a free list that can become convoluted so coalescence is still simple to compute.

The last part that I wanted to implement was the coalescence of contiguous free blocks. I wanted to make sure that when this function was called, it merged all contiguous free blocks in one go.

Explanation of Code

I shall start with an explanation of the struct that defines the blocks. It has three attributes: A flag which is either a 1 or a 0 to inform us whether the block is allocated or free; A size field which contains the size of the data; and a pointer to the next contiguous block.

This struct acts as the header of a block. Please note that the size attribute is the size of the data allocated, *not including the size of the header.*

The flag allows an easy implementation of _free since we can just change the flag and that block is now treated as a free block.

---

[1] (Hack the virtual memory, malloc, the heap & the program break, https://blog.holbertonschool.com/hack-the-virtual-memory-malloc-the-heap-the-program-break/)

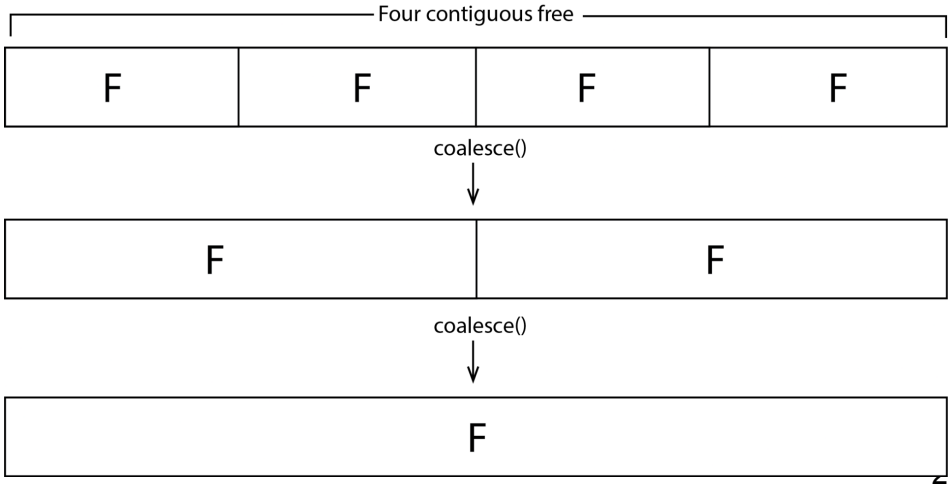
[2] (OSTEP, Free-Space Management, https://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf)

Now onto the juicy part, the malloc implementation. I split up important code into functions for the sake of readability and modularity. These include a function for returning the last block in the linked list, implementing the first fit allocation strategy and coalescence.
Starting in _malloc, first things first is to check the validity of the size parameter. If the user has passed a value that's 0 or less, simply return NULL.
Next we face two possibilities: 1 - This is the first call to _malloc, hence there will be no heap and the head of the linked list will be NULL or 2 - This is not the first time _malloc has been called.
In scenario 1 we must add some memory to the heap using the sbrk() system call, which expands the program break by the parameter passed to it and then returns the previous program break. We can call sbrk() with the size passed into _malloc but we must also add enough space to hold the header of the block, which is just the size of the block struct. Since this is the first call to sbrk() it will return the beginning of the heap. So the head of our linked list is now pointing to the beginning of the heap, perfect. After we have initialized the heap, we can assign the attributes to the appropriate values and then return a pointer to the beginning of the data, not the beginning of the block, so we return the address of the beginning of the heap + the size of the header.
In scenario 2 we must first join together any contiguous free chunks via a call to coalesce(). This ensures that free spaces are merged and not treated as individual blocks. We then search the list to see if there's an appropriate sized block using the first fit method. If a block is found, the data address is returned. If there's no appropriate block, we must add memory to the heap and create a new node to the list at the end.

The first fit function comes with some nuances. I made the decision that a block is only "suitable" if the size we want to allocate is less than the size of the block, the remaining size after allocation should contain enough space for a header + 1. This prevents blocks of smaller size being allocated and leaving empty, unretrievable space between blocks. This means, when a suitable block is found, a new free node can be created in the space between the allocated block and the next block in the list.

There were some design decisions that I decided to implement in the coalesce function that were of my own invention. The function loops through the list, checking each node with the next node along. If both nodes are free, we can merge nodes. Once nodes are merged, the function calls itself. This can be explained with the use of a diagram.

As you can see, after the first call, the four free nodes have been changed to two nodes. If we didn't have a recursive function, the function would stop here, but there's still



contiguous free blocks. Therefore we keep calling coalesce until all free blocks are merged. One drawback of this method is that with a large heap, recursively calling this function will become costly, especially with lots of smaller contiguous free blocks.

Finally we dive into _free(). This function is simple but there are some important parts to note. The pointer passed as the parameter refers to the beginning of the data, but we want to access the header of the block, so we must take away the size of a header struct from the parameter. We can then search through the list for the matching address and once found, the flag is changed. If it is never found we simply exit the function.

Reflection

Starting this project, I had limited knowledge of the memory of a process, was unaware of the various methods and intricacies of allocating memory and had never used a linked list. The task was daunting but after doing careful research, it seemed more approachable. I am happy with my code. I think it's easy to read and has suitable modules. If I was to do this assignment again I would first of all, start five weeks ago, secondly implement a sophisticated memory allocation method such as *Best Fit* and thirdly do more research into proper coalescence methods rather than making up my own.
Overall, I have enjoyed this assignment and have learnt a great deal.