Design Choices Implemented

With my implementation of malloc I first and foremost chose to implement a linked list as opposed to something like a binary tree. This was done because I felt more comfortable working with linked lists and they would make it easier to have metadata about blocks of available memory, such as where it points next; a helpful feature when shuffling the order of the available memory.

Further, my implementation used a first-fit algorithm when assigning processes to available memory. While this would be less efficient when small processes take the memory a larger process could have used, my implementation of free aims to mitigate this by reducing fragmentation of memory.

Lastly, the use of sbrk is required to create a system call to the operating system (OS) to provide more memory. This process takes a lot of time compared to code that doesn't use system calls, however, to minimise the use of sbrk I made sure to allocate 4096 bytes of memory each call. With XV6 having 4096-byte pages, this would make it so that processes are neatly aligned.

Explanation and analysis of the implementation

memory_management.h –

In my memory_management.h I defined the PAGESIZE macro of 4096 bytes and the value of NULL as a void ptr to 0, which XV6 uses as its own equivalent of NULL. I then define a struct called memory_block which will act as a node of a linked list, with the ability to store size, next node in the list and if the node is free.

memory_management.c –

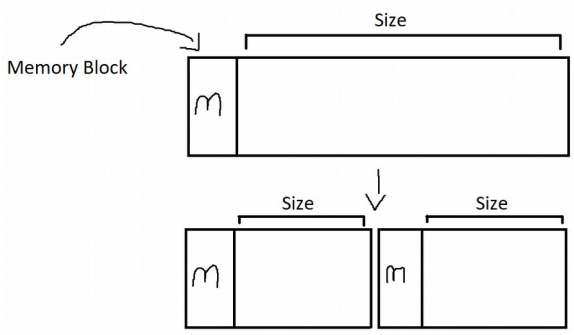
When _malloc() is first called, it initialises a node for a memory block that will be returned. If validation checks pass, it then it gives memory to a global scope head node using the allocateMemory() function which itself will call sbrk for PAGESIZE of bytes and then return this new node in the memory block linked list.

Now that we have a head node, the linked list is traced to find a free block of memory that has sufficient space for the requested size and the size of two memory blocks metadata.

This is because after reading Soshnikov's (2019) use of splitting blocks I wanted to implement the same functionality in my code, of partitioning a block into two parts, the one used and the excess memory. This would have caused a pair of metadata to appear and to why I require there is enough space for a set of metadata on top of the requested space.

partition() works by taking a memory block to be split and the size of the malloc call, that we will split by. The address of the excess memory is calculated by taking the address of the partitioned block, adding its metadata size in bytes and its allocated memory in bytes.

The free memory that will be used by the process calling it becomes the original partitioned block, pointing to the excess memory in the linked list and having its size be that of the partition block minus the size of two headers, and the excess memory size.

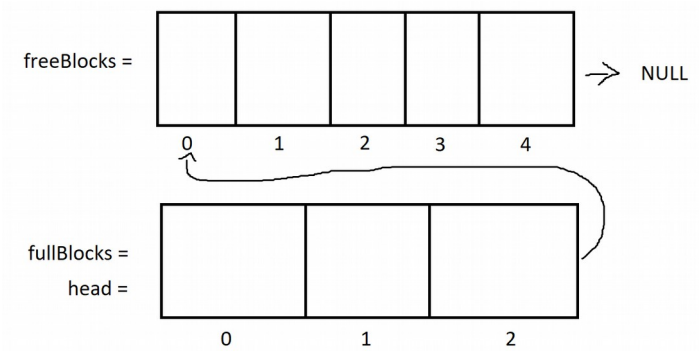


If no appropriate block is found, it will find the tail of memory blocks linked list and then attach a new memory block to it. Then if the size is partitionable as stated above, it will be split. If it does not meet these requirements, it will allocate a new memory block.

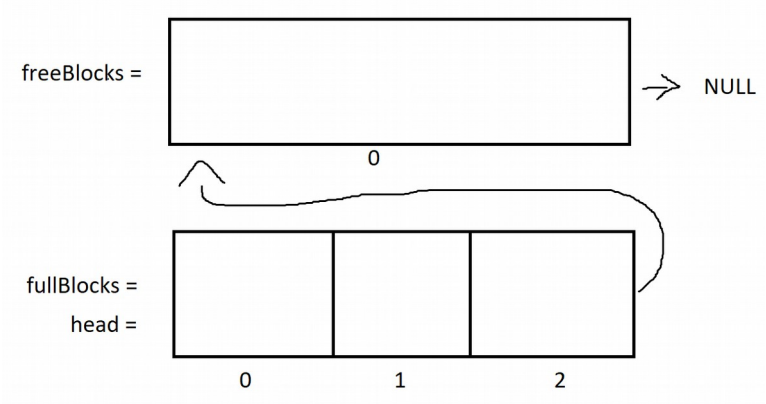
_malloc() will return the memory block + 1 as the actual address we want returned, that of the allocated memory is addressed just after the memory block node.

When _free() is called it will check if the address pointed to minus 1, account for malloc returning the allocated memory not metadata, is already free. If not, then set the flag of the metadata to free. Afterwards, I implement memory defragmentation by creating arrays to store all the free memory blocks and all the full memory blocks. I then once again trace the head node's linked list, adding the free/full memory blocks to their respective arrays.

Now I link all the free memory blocks to each other, and all full blocks together. The head node now becomes the start of the full blocks, and the end of the full blocks, points to the start of the free memory blocks. While the end of the free blocks now points to NULL.



To finish defragmentation, I converge all the free memory blocks at the end of linked list into a single free memory block that can be reallocated, instead of returning this to the heap which would require time costly system calls.



What went well, and improvements to be made

The implementation of defragmentation means my code is a lot more space efficient and potentially time efficient if it can find a suitable block faster. However, in hindsight it could have been even more time efficient, if I put the free blocks at the start of the list so that they are instantly found when requesting to put a new process in, although this would have been at a loss of time when looking for already used processes.

Further, my implementation of partitioning does not adequately work when _malloc requests more data than a single use of allocateMemory() will return. In future implementations, I would aim to make sure it split the process across multiple memory blocks, as currently it will attempt to allocate more memory and then perform undefined behaviour if this is not enough memory allocation.

Overall, I learnt how memory allocators work in real operating systems, and how they are constantly weighing up different advantages / disadvantages for efficiency.

References

Soshnikov, Dmitry. 2019. *Writing a memory allocator*. [Online] [Accessed 30 November 2022]. Available from: http://dmitrysoshnikov.com/compilers/writing-a-memory-allocator/