malloc() and free() in xv6

BY ABDULLA ALI MOLEDINA

Abdulla Moledina [ed20a2am]
STUDENT ID = 201535187

DESIGN DECISIONS:

I used a struct to implement _malloc() and _free() as it allowed me to create blocks of memory with common variables, such as a free variable, a pointer to the next block, and a size variable.

I also decided to use sbrk() with a size of one page in xv6 - 4096 bytes. This is because sbrk() is an expensive call in terms of CPU resources; it must go from user space to kernel space, run the code, and then go back to user space. We want to reduce the number of times sbrk() is called, and 4096 is a big enough size so that sbrk() isn't called often and small enough so that we don't take up all of the memory, but rather only one page size.

CODE IMPLEMENTATION:

memory_management.h:

I defined a struct that is used to contain the variables needed for a block of memory. The 'free' variable is defined as a uint8 data type as it will only use 8 bits (1 byte) of space, rather than 32 bits if it was defined as an int. 'free' only needs one bit (either 1 or 0), hence saving space. I also have an int variable that is equal to the size of the block, and a pointer to the next block.

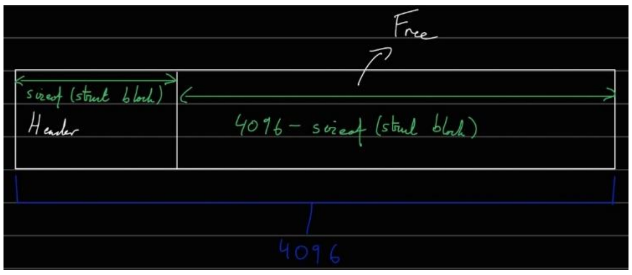
Below the struct I have my function prototypes that are used to run _malloc() and _free().

memory_management.c:

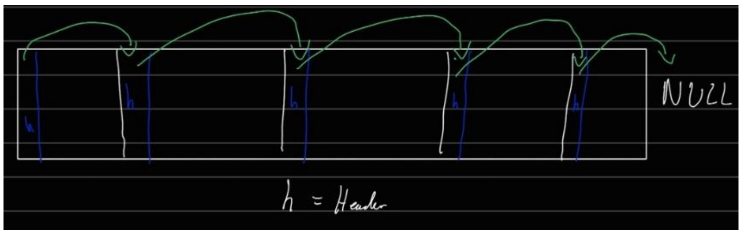
I defined a global head block that will be the head of the linked list and is set to null initially.

When I call _malloc(), I check if the size given by the user is valid. If it isn't, then we print an error message and return out of the function.

We then check if the header block is null. This condition is met, and the code inside is run. We call sbrk(), and check if the call was successful. If it wasn't, our header block will be equal to -1, and so we print an error message and return out of the program. We then set the values of our head block. This is shown in the diagram below:



Next, we start traversing our linked list. I assumed by this point, our total memory has now been split into blocks that are all connected, shown in the diagram below:

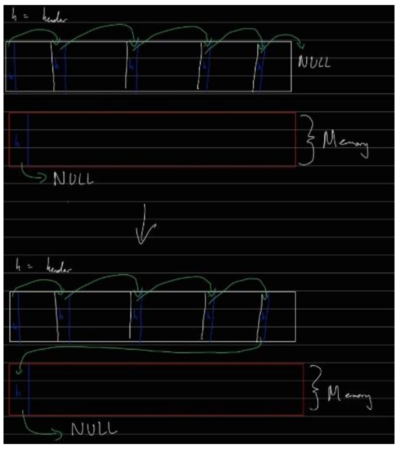


We create an instance of our block variable called freeBlock and make it point to the start of our head block. We traverse through the list while freeBlock is not null. We then check if freeBlock is free and if it is big enough to fit the size we want and one header. If freeBlock meets these conditions, we break out of the while loop. Else, we just move on to the next block and repeat until we either find a suitable block or until we reach the end of the list.

If we reach the end of the list of blocks and have not found a suitable block, then we call our expand() function, and set the result to our freeBlock.

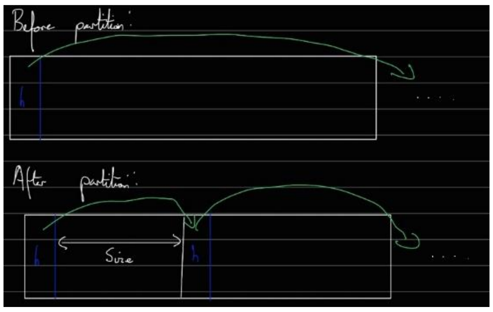
In our expand() function, we create a pointer to our block struct, and call sbrk(). We grow the memory four times the page size, so we have a lot of memory to allocate. We then check if the sbrk() call is successful and set our values for our memory block, like how we set the values for our header in our _malloc() function.

We now must set the memory block we have just created to be the tail of our list. We loop through our list and find the last block. Once we have found it, we make it point to our memory block, which now becomes our last block. We then return the memory block. The diagram below shows the change from our having our memory block to adding it to our list.



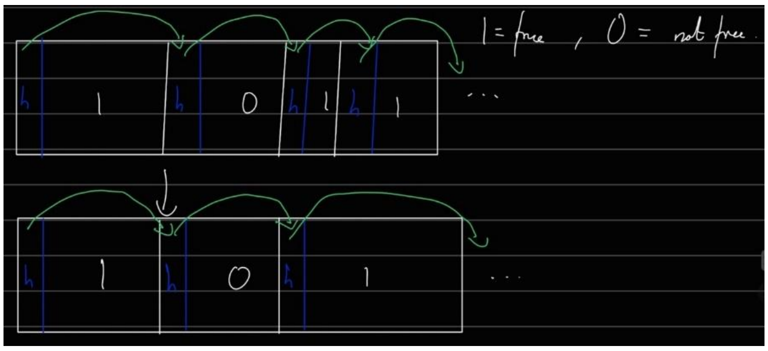
We now check if the block we have is bigger than the size plus one header, because each block must have a header and block to allocate memory, and our partition() will split it into two different blocks. If the condition is met, we call the partition() function (Burelle, 2009).

We take in the block we want to split and the size we want in our parameters. We start by creating a block called split and set its address in memory to the freeBlock's address, plus the size of a header and the size we want. We then set split's size by taking away the size of a header and the size we want from freeBlock’s size. We then make split point to the block that freeBlock is pointing to and set it to be free. We then set freeBlock’s size to be the size we want and pointing to split. Afterwards, we return the new freeBlock. Here's a diagram to show the original freeBlock and the new freeBlock after the partition function is called:



We set freeBlock to not be free and return the freeBlock + 1 at the end. We add one to return the block of memory, not the header (Luu, 2022).

We now need to implement _free(). We check if the pointer given is valid, and if the pointer has already been freed by casting it to a struct header*. If it passes these tests, we then set the block to be free. We now must check if there are any free blocks next to each other, and if so, we must merge them into one. This is shown in the diagram below:



REFLECTION:

This coursework helped me to learn more about using different data structures in C (structs and linked lists) together. It also helped me understand more about memory management in xv6 and apply the reading from the xv6 book into practice.

It took me a while to get started with this coursework, because I didn’t fully understand how to get started, and the stuff I read from chapter 3 of the xv6 book. However, after finding some resources online, I was able to grasp some idea of how to get started.

One thing I will consider next time is trying to use a best fit algorithm. My program uses a first fit algorithm to find free blocks of memory, however, whilst it may be time-efficient, it is not very space-efficient. This could be added by creating another pointer that keeps track of the block that is best fitting for the size we want.

REFERENCES:

Burelle M. 2009. A Malloc Tutorial. Laboratoire Syst`eme et S´ecurit´e de l'EPITA (LSE). pp 9 – 10.

Luu, D. [No Date]. Dan Luu's Malloc Tutorial Website. [Online]. [Accessed 24th November 2022]. Available from: https://danluu.com/malloc-tutorial/