Michael Wiciak

# Malloc / Free Implementation

In XV6 Operating System

Michael Wiciak

# Design Decisions

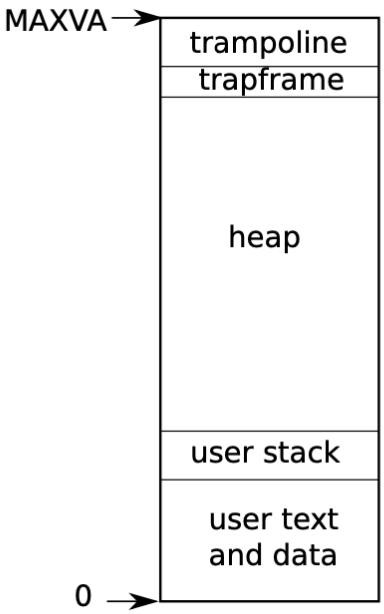


Figure 1: Memory Layout in XV6 (Cox, 2021)

Memory layout in XV6, shown in Figure 1, allows user programs to expand the heap. Malloc will internally control the pointers it returns to ensure integrity and not allow overwriting of used and not freed pointers. To do this, malloc has to store data in contiguous variable length chunks and each has to store metadata shown in the shaded area in Figure 2. The best data structure to store this is a linked list that stores metadata as nodes. If all chunks in the heap are allocated, malloc has to expand the heap and keep track of it. Free will need to access the linked list and change the node data to let Malloc know that data can be overwritten. Malloc will be able to return a pointer to it, allowing the user to write new data there. Since the user specifies how much memory they need, accessing anything outside of the provided pointer will result in segmentation faults.

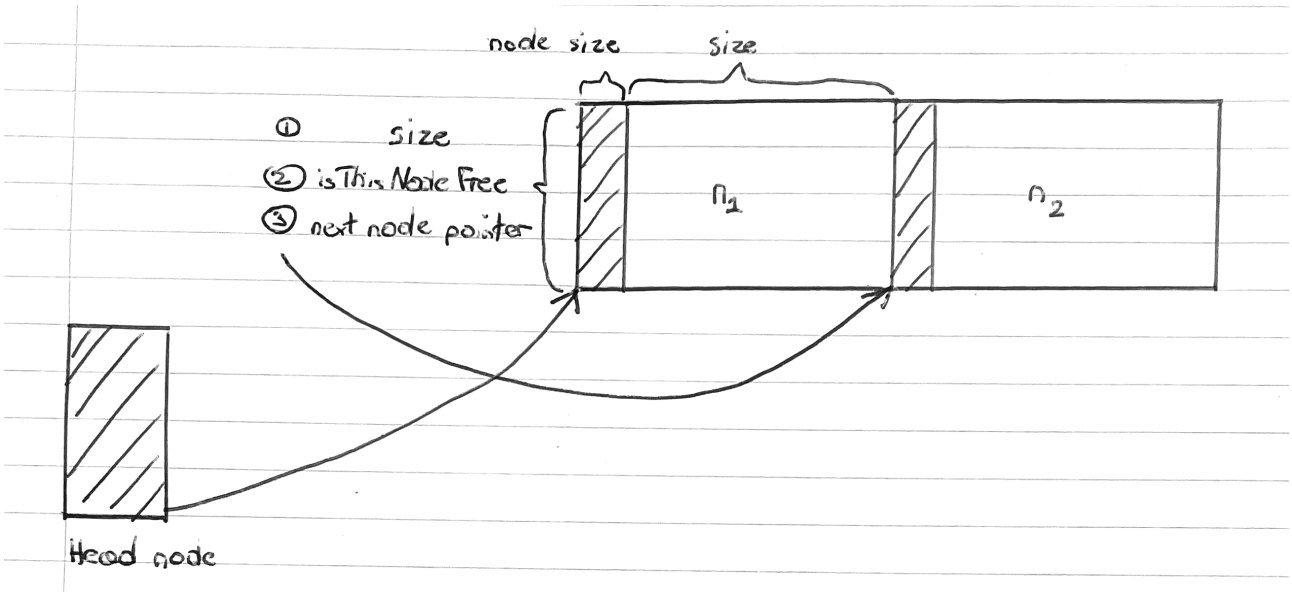


Figure 2: Shows how the linked list structure is used for malloc.

# Implementation

The structure of each chunk allows them to be connected and traversable. For consistency, the size of the node is stored as a macro as all nodes have the same size. Linked List, implemented as a struct, can only be accessed fully by using the head node. It's set as a static variable for security so only Malloc/Free use it.

**Michael Wiciak**

If anything in Malloc fails, like the size provided is negative, a NULL pointer is returned to let the user know it's an error. The linked list header is initialised with size zero and pointing to itself so Malloc will not attempt to change it. The linked list is traversed using a pair of node pointers, initially the head pointer and whatever head points to so NULL. With each iteration, the two pointers change to be the pointers of the next node. This allows the loop to manipulate the past, current and next node safely. The loop exits if we have reached a node which doesn't point anywhere (set to NULL) or wrapped around the heap and returned to the initial head pointer. For readability, I have used the exit flag. The loop attempts to find an already allocated chunk in the heap that has the same or larger size than the one requested by the user and can be overwritten. If it finds it, it returns the pointer to that location and changes the node information so Malloc cannot return that pointer again until it has been freed. This is shown in Figure 3. If this fails, malloc

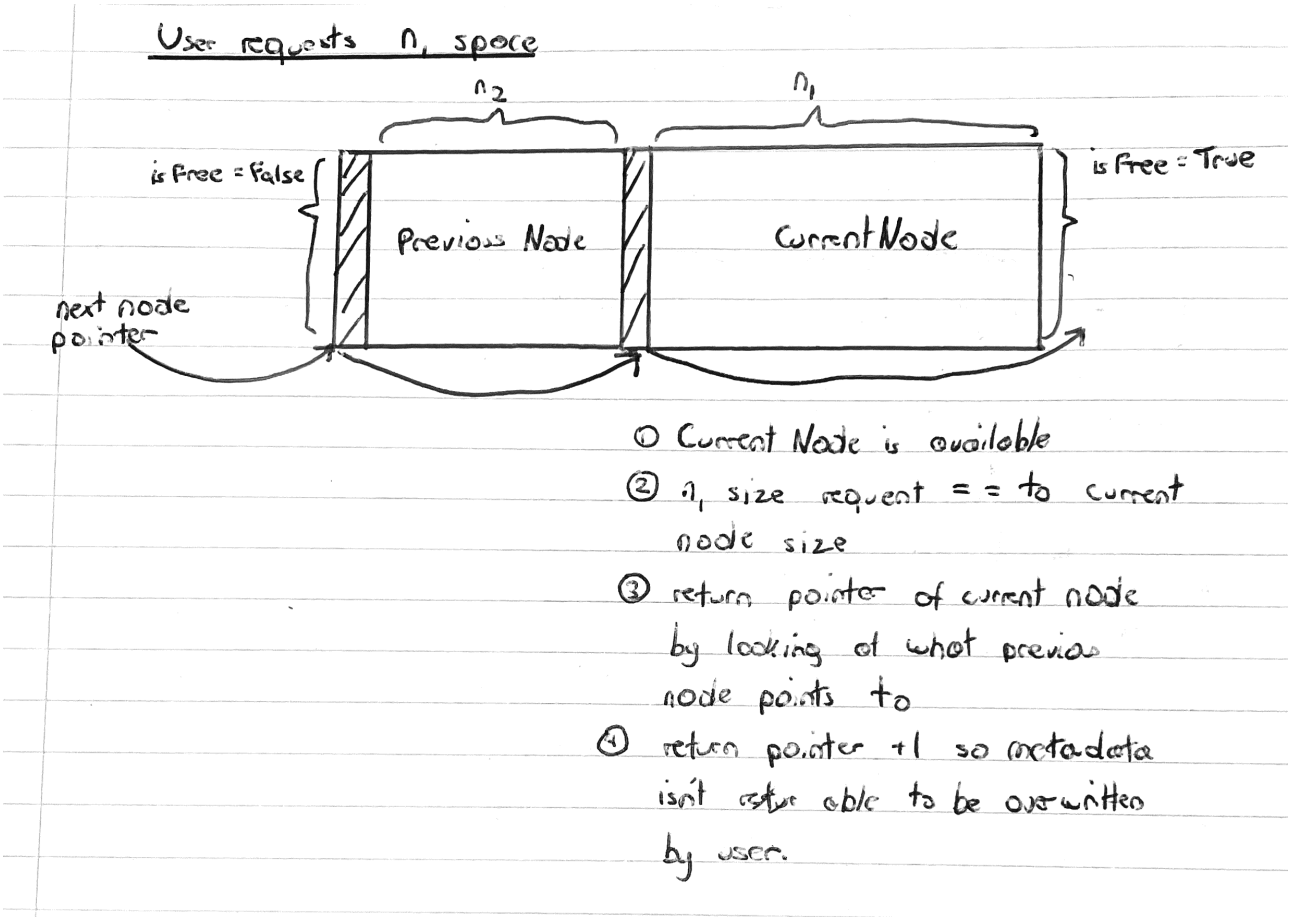


Figure 3: Malloc finding the exact sized chunk it needs.

has to request an extension of heap memory, using the sbrk function (IBM, 2021), from XV6 as all the known chunks are already allocated. A new node is created to match exactly the size requirements of the user plus the space required to store the node, so Malloc doesn't use more heap memory than needed. This new node points to NULL since nothing is ahead of it yet and the previous node now points to this new node. This is shown in Figure 4. If there is no previous node, the header points to the new node. Malloc returns a pointer + 1, meaning it returns a pointer where the user can start to write their data as at the beginning of each chunk without overwriting the metadata of the node.

**Michael Wiciak**

Free takes the pointer and minuses the size of the node as each node in the linked list points to the start of the chunk which includes the metadata. Free traverses through the linked list, like malloc, and checks if any pointer inside the nodes matches the pointer provided. If there is an error or it doesn't find that pointer inside the linked list, it exists without doing anything. If it finds that pointer, it will edit the node metadata so it can be returned by Malloc and overwritten by the user. Optimisation in Figure 5 will attempt to merge any adjacent chunks of memory with one node, so small chunks of memory can be joined and used to store large amounts of memory, reducing the impact of heap memory fragmentation during run-time.

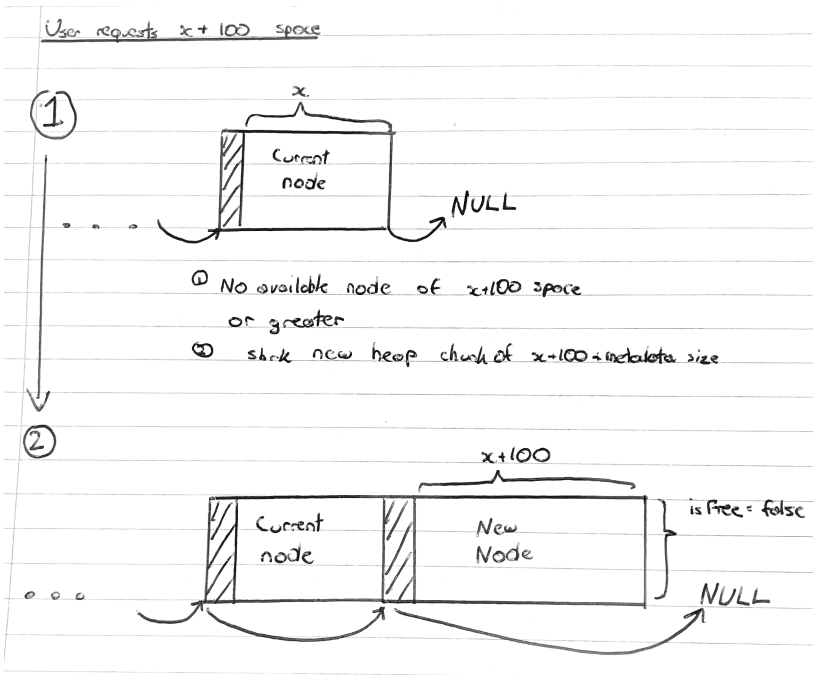


Figure 4: Shows what sbrk does.

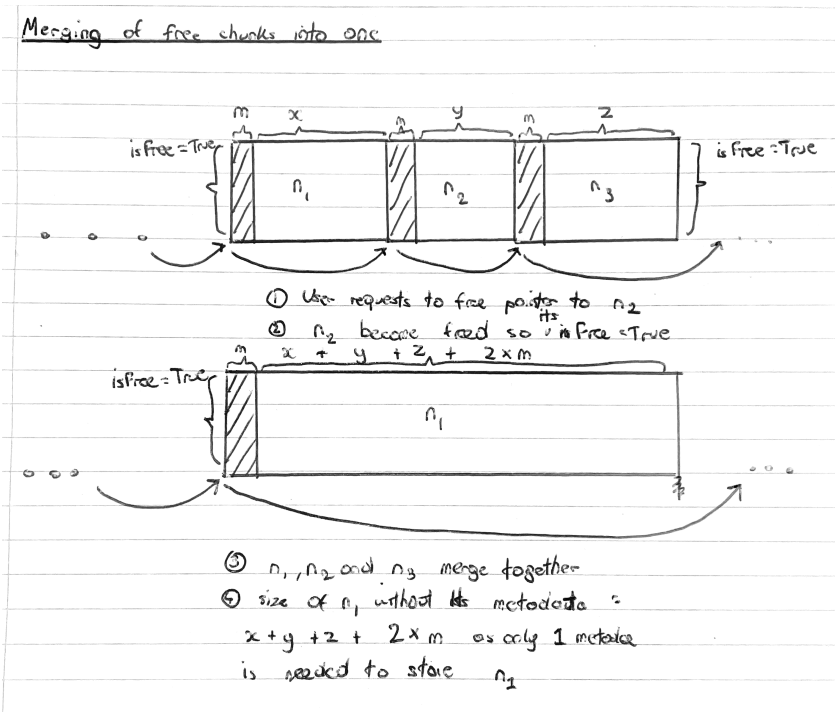


Figure 5: Merging of freed chunks.

# Reflection

I have learned the underlying details of how XV6 works and manages memory, allowing me to better understand how to write future programs that directly access memory and make them more efficient than built-in implementations. Malloc and Free work, as specified in the brief, are optimised to reduce the number of fragmentations and reuse the same chunks of memory, making it more space efficient. Another optimisation could be allowing malloc to split a chunk so each Malloc request returns a pointer to the requested amount of space. The main problem with any heap-based implementation is that there will always be some fragmentation. To combat this, the chunks of memory allocated could be of fixed size regardless of how much space the user requested (Murphy, 2016).

# References

Cox, RC. “xv6: a simple, Unix-like teaching operating system”, pp. 26. 2021.

IBM. “sbrk() — Change space allocation”. IBM. [Online]. 2021. [Accessed 27 November 2022]. Available from: https://www.ibm.com/docs/en/zos/2.4.0?topic=functions-sbrk-change-space-allocation

Murphy, NM. “How to Allocate Dynamic Memory Safely”. BarrGroup. [Online]. 2016. [Accessed 27 November 2022]. Available from: https://barrgroup.com/embedded-systems/how-to/malloc-free-dynamic-memory-allocation