When programming there are two groups of memory the operating system will sort the program's memory into. Initialized memory also known as the stack and uninitialised memory also known as the heap (Lanza, 2018). The size of the stack is defined on compile time whereas the size of the heap is decided during run time. In general OS place the heaps and stacks on opposite ends so that as they grow and shrink they move towards the middle of the memory address space. This way there are less conflicts. However this task's focus is on creating a heap memory management scheme for the xv6 operating system. Since the size of the heap is unpredictable, the operating system cannot guarantee the heap is allocated entirely contiguous memory. So a linked list is used to represent the heap as shown in *figure 1* below. It starts with a base_head that then points to the next head until the terminating NULL pointer is reached. That means that each used and free block of memory has its own head with its own information pointing to the next one in line.Another reason a linked list is used is that even if the memory is contiguous the links define borders between different used blocks and between used and free blocks.

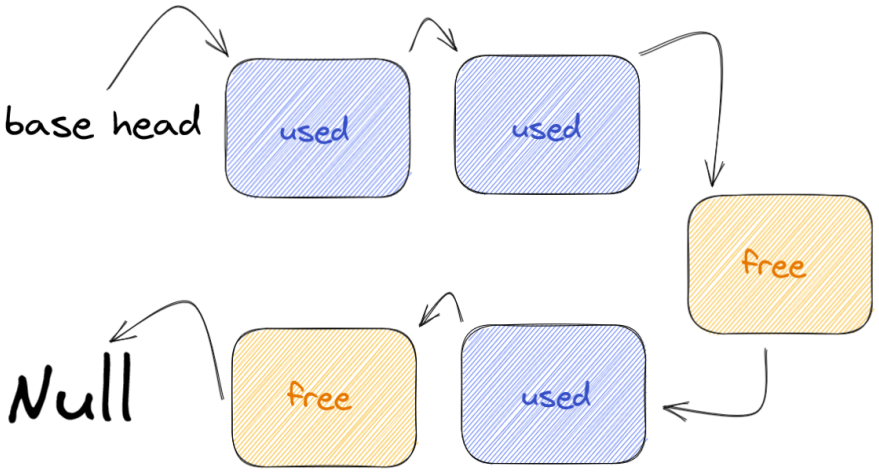

Figure 1

Every block has a header, so it is important to design the data structure containing all the header information. The main information is, if it's a free block, the size of the body, and the next block in the list. One consideration when making the header data structure, it would be beneficial to make it a consistent size and a size that is a multiple of 2. I arbitrarily set my size to 16 bytes and did so by making a union with a uint16 and a struct (Kernighan B. W., Ritchie D. M., 1988). By never setting the uint16 it only sets the union's size and the struct is the one with all the information.

Before allocating any memory _malloc() performs some sanity checks. First _maloc() checks if the user submitted a valid size. And second more importantly _malloc() checks if the linked list has been initialised. Once the checks are passed this implementation of _malloc() allocates memory using the best fit method. This method goes through the entire linked list to find the free block which closest matches the requested size and allocates that block (Arpaci-Dusseau A. C., Arpaci-Dusseau R. H., 2018). This method is beneficial as it packs the linked list as tightly as possible making it more memory efficient. However this method has a performance cost as it must iterate through the entire linked list every time _malloc() is called. Since the specs say that the focus of the allocator is to be memory efficient this time penalty is justifiable. Once the linked list is iterated through either no block is found or a best fit block is found. If a block that is

big enough is not found then more_mem() is called but if a best fit block is found then the butcher() method is called. These two helper functions will be discussed later in the report.

When using more_mem() to call sbrk() to ask for more memory, memory and time considerations need to be made. The smallest addressable unit of memory is 1 page or 4096 bytes (Cox R., Kaashoek F. and Morris R., 2021). First sbrk() returns the exact amount of bytes requested, however the OS will have to allocate an entire page as it cannot address a smaller size. So to not fragment the pages I round up the user requested size to the nearest page. Calling sbrk() also costs a significant amount of time as it needs to go from user to kernel space and back. Less sbrk() calls would be beneficial so in addition to rounding up to the nearest page I also allocate an additional page to the heap when more_mem() is called. An additional consideration is that because more_mem() most likely generates a free and a used block it will be beneficial for the heap to free the free block and potentially merge it with a prior one (Kernighan B. W., Ritchie D. M., 1988).

To cut one big free block into a free and a used section the butcher() is called. To do that the free block must be big enough to account for not only the *size* argument but also any overhead. First bit of overhead is the head of the *size* used section, but the free section also has overhead. It needs at least to fit its head in there in order to not break the list and utilise all memory in the heap. *Figure 2* shows that if all the overhead is not accounted for it causes internal fragmentation. That is when memory inside the free block is not sufficient to hold all the overhead (Silberschatz A., Galvin P.B. and Gagne G., 2018).

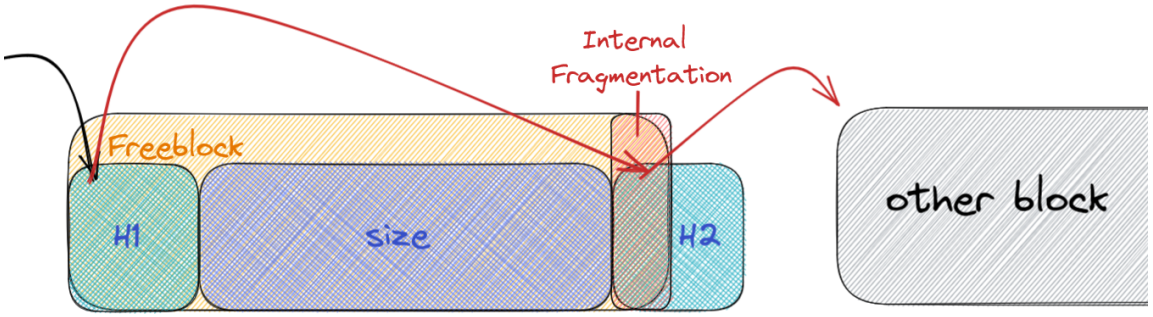

Figure 2

Since the size of each header is 16 bytes a free block needs to be 32 bytes bigger than the user requested size to be able to be utilised in the heap. If the second header is not accounted for then those 16 bytes can no longer be accessed in the heap and over time will clog up the system's memory. So this overhead is accounted for when calling butcher(). However this means that almost good enough free blocks will be skipped making the heap slightly less memory dense.

To free a block in the implementation is trivial, as only the boolean value in the header must be changed from false to true. However in order to free efficiently additional considerations need to be taken. First undefined behaviours such as freeing an already freed block and freeing a null pointer are accounted for. Then the heap is traversed to verify if the free block exists in the heap. This verifies that the user has not given a faulty pointer that is not part of the heap. Second type of considerations are memory efficiency considerations.

Take the scenario, three small consecutive blocks are freed. The then user requests a block larger than the individual free blocks but smaller than the three combined, as shown in *figure 3*. This is one example of external memory fragmentation.

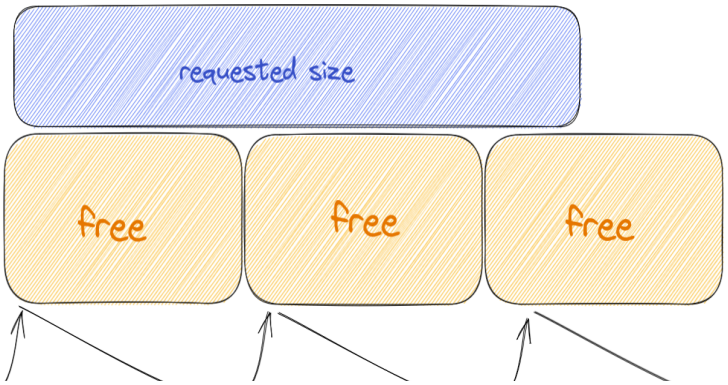

Figure 3

External memory fragmentation is when the cumulative free space in the list is big enough to store a given size but the free space is broken up so it can't be allocated . However *Figure 3* is an example of an external fragmentation problem that can be fixed using compaction. Compaction simply means that consecutive free blocks are combined into the largest possible free block. When calling _free() it always checks its left and right neighbouring blocks to see if any merging can happen. This however assumes that the neighbouring blocks in the linked list are also neighbours virtual memory address space (Arpaci-Dusseau A. C., Arpaci-Dusseau R. H., 2018). This is because a linked list does not guarantee contiguous memory between blocks, which is a pro normally but now is a con. Another example of external memory fragmentation is when a used block is in the middle of two free ones. If the free ones were compacted then the user requested size would fit however they cannot be compacted. This is because until _free() is called a used block must not be moved around as it is used by the user (Arpaci-Dusseau A. C., Arpaci-Dusseau R. H., 2018).

I learned a lot during this exercise. First I strengthened my knowledge of pointer arithmetic and understanding memory spaces. Second main thing was I had very tangingle experiences tackling internal, external fragmentation and really had a front row perspective on what makes those issues so challenging to deal with.

There are a couple of things I would do if I were to extend this exercise. First I might look into making my singly linked list into a doubly linked one. This way I will not have to traverse the entire heap every time I need to find the previous and next block of a given block. In the beginning I assumed that best fit would be the most efficient way to pack the heap, however from my explorations into fragmentation I now understand that depending on the OS it can create a substantial amount of external fragmentation (Silberschatz A., Galvin P.B. and Gagne G., 2018). This is because it creates many small free blocks which may rarely be used. In the future I would like to test how worst fit compares as it creates the largest possible free blocks which I think would be better at creating less external fragmentation.

References

Arpaci-Dusseau A. C., Arpaci-Dusseau R. H., 2018, Free Space Management. *Operating Systems: Three Easy Pieces.* [Online]. :Arpaci-Dusseau Books. 1-3, 11. [30 November]. Available from: https://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf

Cox R., Kaashoek F. and Morris R., 2021, *xv6: a simple, Unix-like teaching operating system*. [Online]. 31, 37.[30 November 2022]. Available from: https://minerva.leeds.ac.uk/ultra/courses/_539722_1/outline/file/_10443653_1

Kernighan B. W., Ritchie D. M., 1988, 8.7 Example - A Storage Allocator. *The C Programming Language.* [Online]. Edition 2. United Kingdom: Pearson, 185-190. [30 November 2022]. Available from: https://github.com/AzatAI/cs_books/blob/master/The.C.Programming.Language.2nd.Edition.pdf

Lanza, Medium, 2018, *Stack vs Heap. What's the difference and why should I care?*. [Online]. [30 November 2022]. Accessible from: https://nickolasteixeira.medium.com/stack-vs-heap-whats-the-difference-and-why-should-i-care-5abc78da1a88

Silberschatz A., Galvin P.B. and Gagne G., 2018, Chapter 9 Main Memory. *Operating Systems Concepts*. [Online]. Edition 10. United States of America: Laurie Rosatone, 359. [30 November 2022]. Available from: https://www.academia.edu/42880365/Operating_System_Concepts_10th_Edition