Htoo Aung (sc21hta)

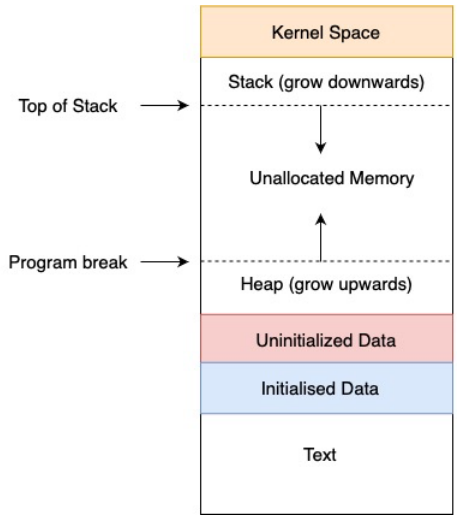
Xv6 Memory Management Report

Design Decisions

A linked-list "**struct** mem_Chain(int size)" is used to represent the heap as the size of this data structure can expand or contract on run-time with efficient insert and deletion techniques.

How does the memory allocation work for the first time? In the heap, after the program break has been set, the compiler will look for any memory-address between previous program break and newly-allocated program-break[1].

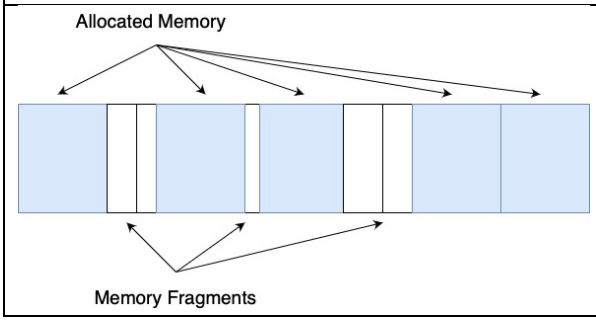
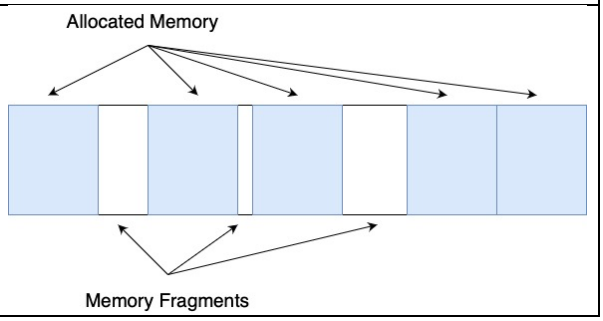
sbrk() function modifies the program break by expanding it, where the heap grows upwards (i.e., in a typical memory-layout on Linux/x86-32)[2]. If a call to sbrk() is successful, it returns the pointer to the previous program break [3][4].



A typical memory layout

In this implementation, "**void** _free(void *ptr)" doesn't lower the program-break, however, it adds a memory-block to "**MEM_START**(ptr)" (i.e., a global-pointer to a linked-list of freed blocks) to ensure that they are recyclable for subsequent malloc-requests()[3]. It also checks whether there are adjacent freed blocks and merges them, thus minimizing sbrk() calls.

Coalescing prevents tiny adjacent memory-block from fragmenting on the free-list (i.e., memory fragmentation), which would not be space efficient otherwise, as they cannot accommodate memory for slightly larger future memory-block-requests.

Before Coalescing:	After Coalescing:
	

Explanation

Functions:

void add(**struct** mem_Chain *chain):

Takes a memory-block/chain, adds to the linked-list and positions the leading-node appropriately.

void delete(**struct** mem_Chain *chain):

Discards a memory-block or memory-chain from a supplied pointer-variable to the structure positions the leading-node appropriately.

void search_coalesce_setBreak**()**:

First it searches vacant-blocks so that it can be coalesced. It does 3 validations:

- If the ending memory-block terminates where the program break is it returns **NULL** pointer.
- If there are adjacent free-blocks merging takes place (i.e., to deal with memory fragmentation).
- if the last memory-block on the free list terminates where program break begins, and discards if the size is appropriate to (i.e., to limit the number of sbrk() function calls)

struct mem_Chain *partition(**struct** mem_Chain *chain, **int** size):

Splits "chain" into a smaller memory block whose block is a reduce size of "size" of its original size

struct *_malloc(**int** size):

It searches the list of memory-blocks previously freed in order to find the memory-block whose size is **>=** the requested block.

- If requested memory-block is equal to one of those freed blocks, this address is returned.
- If requested is greater than those freed-blocks **"partition"** takes place.

This is because, the only eventuality when OS should facilitate sbrk() function call is when the system has requested more memory-blocks than size of the addresses that are stored in the memory-chain that contains all previously freed memory-blocks.

void *_malloc(**void** *ptr):

Firstly, it checks whether the **ptr** points to any random memory-address outside the heap. Adds **ptr** to a free-list. Calls search_coalesce_setBreak**().**

Reflection

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ memory_management
Integers:
1->0x0000000000004018
2->0x000000000000401C
3->0x0000000000004020
4->0x0000000000004024
5->0x0000000000004028

Charatcers:
A->0x0000000000004048
B->0x0000000000004049
C->0x000000000000404A
D->0x000000000000404B
E->0x000000000000404C
$
```

The block of memory returned by malloc() is always aligned according to their byte boundary-size for any primitive C datatype. For example, **sizeof**(int) is 4 bytes, **sizeof**(char) is 1 byte in xv6 and most 32-bit/64-bit machines.

Commenting line 251 to line 280 and uncommenting the main function provides the following output on the right, which shows that the address increases by 4 bytes for **int** and 1 byte for **char.**

Memory-leak test (uncommenting the main function and commenting line 251 to line 261) as shown on bottom-right also demonstrates that heap grows steadily, until its size reaches the maximum-limit (i.e., when **leak** is too large) in the virtual memory [2]. This is a typical behaviour expected in dynamic memory allocation
n, provided that the memory is not freed after usage.

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ memory_management
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x0000000000000220 stval=0x0000000000000000
$
```

All these tests confirms that memory is allocated contiguously, and void ***malloc**(int size) is functioning in the same manner as void ***malloc**(size_t size) from **stdlib.h**.

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ memory_management
Integers:
1->0x0000000000004018
2->0x000000000000401C
3->0x0000000000004020
4->0x0000000000004024
5->0x0000000000004028

1->0x0000000000004018
intArr[0] = 1

Before freeing:
intPtr->0x0000000000000002

usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x0000000000000006 stval=0xfffffffffffffffa
$
```

Free-test as shown on the left indicates that accessing a memory pointer that has been freed but not reallocated will give a **"usertrap()"** Xv6, but in conventional C-programming this should return a **"Segmentation fault (core dumped)".**Therefore, it can be assumed that free works on a basic level, although more rigorous testing may be required to confirm that its behaviour resembles to that of **free()** from **stdlib.h.**

Skills acquired: gdb, C-programming (linked-lists, structures, dynamic-memory allocation and pointers), Debugging, xv6, Memory-Management of Operating Systems, Makefiles, Vim/Nano, Time-management, Open-source software development.

Author's note: The main function that I have used in my final testing phase has been included in the submission. Please feel free to verify my claims by uncommenting each individual test separately and their relevant pointers and variables and compile and integrate the program into xv6.

References

[1] Dan Luu. Malloc Tutorial. August 21, 2019. [Online]. [Accessed 21st November 2022]. Available from: https://danluu.com/malloc-tutorial/

[2] Michael Kerrisk. The Linux Programming Interface, A Linux and UNIX System Programming Handbook, 2010, Chapter 6 pp.119. [Online]. [Accessed 24th November 2022]. Available from: https://sciencesoftcode.files.wordpress.com/2018/12/the-linux-programming-interface-michael-kerrisk-1.pdf

[3] Michael Kerrisk. The Linux Programming Interface, A Linux and UNIX System Programming Handbook, 2010, Chapter 7 pp.139-146. [Online]. [Accessed 24th November 2022]. Available from: https://sciencesoftcode.files.wordpress.com/2018/12/the-linux-programming-interface-michael-kerrisk-1.pdf

[4] Marwan Burelle. Laboratoire Syste`me et Se´curite´ de l'EPITA (LSE). A Malloc Tutorial*. February 16, 2009, [Online].[Accessed 25th November 2022]. Available from: https://wiki-prog.infoprepa.epita.fr/images/0/04/Malloc_tutorial.pdf