



॥ त्वं ज्ञानमयो विज्ञानमयोऽसि ॥

# Heap Memory Manager with Command-line Debugger

by

**Aimless Bots**

**Department of Computer Science & Engineering  
Indian Institute of Technology Jodhpur**

*A Project Report Submitted to  
Indian Institute of Technology Jodhpur for the project done during our  
course of Operating Systems (CS330)*

## **Supervisor:**

Dr. Ravi Bhandari & Dr. Suchetana Chakraborty

Department of Computer Science & Engineering  
Indian Institute of Technology Jodhpur

May, 2021

# Declaration

We, Aimless Bots, hereby declare that the code implementation of our Project is original and has not been published and/or submitted before.

#	Names	Roll Numbers
1	Manul Goyal	B18CSE031
2	Nishant Jain	B18CSE067
3	Manan Shah	B18CSE030

Date: 9 May 2021

---

# Dedication

We dedicate this report to the Almighty God without whom we can do nothing. We further dedicate it to our parents and guardians for their unceasing and selfless support throughout duration.

# Acknowledgement

We are deeply indebted to our project supervisors Suchetana Chakraborty and Ravi Bhandari, their unlimited steadfast support and inspirations have made this project a great success. In a very special way, we thank them for every support they have rendered unto us to see that we succeed in this challenging study. We would also like to thank them for the assignments which helped to add various features to our project.

We thank IIT Jodhpur for giving us the grand opportunity to work as a team which has indeed promoted our team work spirit and communication skills. We also thank the individual group members for the good team spirit and solidarity.

# Contents

<b>Declaration</b>	<b>i</b>
<b>Dedication</b>	<b>ii</b>
<b>Acknowledgement</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Motivation . . . . .	2
1.3 Problem Statement . . . . .	2
<b>2 Methodology</b>	<b>3</b>
2.1 Introduction . . . . .	3
2.2 The Heap Memory Manager (HMM) . . . . .	3
2.2.1 Page Family Registration . . . . .	3
2.2.2 Virtual Memory Page Organisation . . . . .	4
2.2.3 Block Splitting and Merging . . . . .	8
2.2.4 Internal Fragmentation . . . . .	8
2.2.5 XMALLOC & XFREE . . . . .	11
2.3 Command-line Debugger for Programs using our HMM . . . . .	13
<b>3 Experimental Setup</b>	<b>15</b>
<b>4 Presentation of Results</b>	<b>17</b>
4.1 Introduction . . . . .	17
4.2 deb show mem . . . . .	17
4.3 deb register “[struct_name]” [struct_size] . . . . .	19
4.4 deb allocate “[struct_name]” [quantity] . . . . .	19
4.5 deb free [memory_address] . . . . .	20
4.6 deb show blocks . . . . .	21

<b>5</b>	<b>Limitations, Conclusion and References</b>	<b>22</b>
5.1	Introduction . . . . .	22
5.2	Limitations . . . . .	22
5.3	Conclusions . . . . .	22

# List of Figures

1.1	The role and position of a heap memory manager, with respect to the kernel and user program. The user program requests memory for holding its structs, which are carved out from VM pages requested by the heap memory manager from the OS kernel using <code>mmap</code> and <code>sbrk</code> system calls. . . . .	2
2.1	Linked-list of meta VM pages for storing page families. Structs A, B and C have already been registered. The user application requests to register a new struct named <code>foo</code> of size 20 bytes. Since the right page does not have enough capacity to accommodate a new page family block (shaded in purple), a new VM page is requested by the HMM and added at the head of the list, and the page family is stored in it. Note that each page family block has the same size. . . . .	4
2.2	Organisation of a data VM page. The lowermost part stores information about the page, which is followed by meta blocks and data blocks. Here, it is assumed that the page size is 4096 bytes, each meta block occupies 48 bytes, and the page information occupies 32 bytes. The page is assumed to start at address 0 and the virtual memory address for each block is shown to the left. . . . .	6
2.3	The data VM page list and free block priority queue associated with a page family for struct <code>foo</code> . The page family block (purple) contains pointers to these two data structures. Each data VM page also contains a back pointer to the page family, although not shown here. The largest free block (FB1 here) is present at the head of the priority queue. . . . .	7

2.4	An example of block splitting. The data block (of size 500 bytes) at address 80 is chosen to serve a memory request for 100 bytes. It is split into a data block of size 100 bytes at address 80, which is returned to the user program and marked as occupied, and a residual free block of size 400 at address 180. This residual block is further divided into a smaller free data block of size 352 bytes at address 228, and its guardian meta block at address 180. . . . .	9
2.5	An example of block merging. The user program requests for the data block at address 228 to be freed. Since both the next and previous data blocks (at addresses 628 and 80 respectively) adjacent to this block are free, all the three blocks are merged together into one larger free block of size 4016 bytes. Note that the meta blocks guarding this data block and the next data block are destroyed in the process, and their space is merged into the resultant free data block. . . . .	10
2.6	Examples of hard and soft IF. Assuming the size of page family <code>foo</code> to be $s = 20B$ and meta block size to be $m = 48B$ , the residual block (of size 6 B) at address 4088 suffers from hard IF, because it is not guarded by a meta block, and cannot accommodate one. It can only be reclaimed when the data block at 628 is freed. The free data block (of size 12 B) at address 568 suffers from soft IF, because it is guarded by a meta block, but it cannot accommodate an instance of the page family <code>foo</code> (of size 20 B). It can be reclaimed when either the data block at 628 or the one at 80 is freed. . . . .	11
2.7	Flow diagram of our XMALLOC Implementation . . . . .	12
2.8	Flow diagram of our XFREE Implementation . . . . .	12
2.9	The interaction of the main thread and the CLI thread. . . . .	14
4.1	First line shows the size of virtual page which will be cached by Memory Manger from Kernel MMU. And cursor shows the stating point of CLI where we issue various commands. . . . .	17
4.2	Both the structs, i.e <b>struct a</b> and <b>struct b</b> have been registered till the first debug. No VM pages have been allocated to them as no XMALLOC has been called until the first debug() . . . . .	18
4.3	2 instance of <b>struct a</b> have been allocated using XMALLOC() due to which a VM page has been requested from Kernel MMU and has been appended to the page family of <b>struct a</b> . . . . .	18
4.4	Example of registration . . . . .	19
4.5	Example of allocation . . . . .	20



4.6	Example of freeing memory . . . . .	20
4.7	Example of freeing memory . . . . .	21

# Chapter 1

## Introduction

### 1.1 Background

Virtual memory is a memory management technique which isolates various user processes from one another as well as gives them the illusion of a single large contiguous address space, without them worrying about the actual physical memory allocation which may be smaller and non-contiguous. The virtual address space of a process is typically divided into five segments - the text (or code) segment, the data segment, the BSS segment, and the dynamically expanding heap and stack segments. All the dynamic memory allocation requests by a program are satisfied by the heap segment, which can grow (typically towards higher virtual memory addresses) during run-time to satisfy these requests. In C, the `malloc`, `calloc`, `realloc`, and `free` functions are provided by glibc which are used to allocate and deallocate memory dynamically for use by the user program. These functions essentially hide the low-level details of the heap memory, and its organisation and management from the user program. Internally, these functions request *virtual memory pages* (VM pages) from the OS kernel using `mmap` and `sbrk` system calls, and cache these pages so that memory allocation requests by the program can be satisfied by carving out chunks (or blocks) from these pages. In this project, we have built our own heap memory manager (HMM) for C programs in Linux, essentially replacing the glibc-provided `malloc` and `free` functions. The role of the HMM is shown in Figure 1.1. We have also developed a novel command-line debugging API, which can be easily integrated into any C source program, and display the memory layout and usage statistics at various points during run-time, along with some interactive commands which can be used to test our heap memory manager.

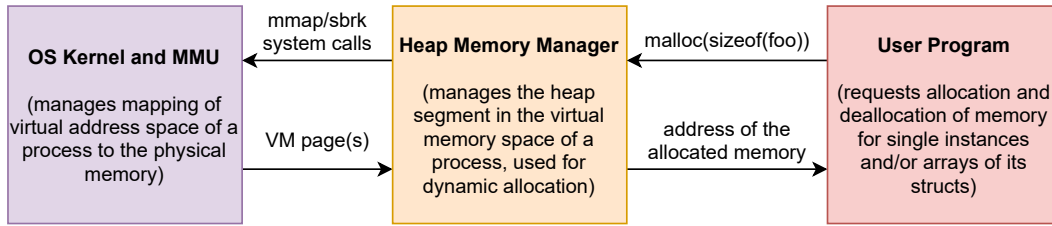


Figure 1.1: The role and position of a heap memory manager, with respect to the kernel and user program. The user program requests memory for holding its structs, which are carved out from VM pages requested by the heap memory manager from the OS kernel using `mmap` and `sbrk` system calls.

## 1.2 Motivation

The motivation behind this project is to learn about the internal working of `malloc` and `free` functions, as well as the system calls used by them internally. Another motivation is to facilitate easy debugging of the programs which use our HMM, by providing a simple and easy-to-integrate CLI with commands to display and manipulate the VM pages cached by the HMM, and their organisation into blocks. This novel feature allows the user to actually see under the hood how the memory is being managed and used at a low-level, how much memory is being used by each `struct` in his program, and allocate and deallocate memory during run-time through commands provided by the CLI.

## 1.3 Problem Statement

Implementing our own Heap Memory Manager using system calls and heap management strategies to create our own version of glibc-provided `malloc` and `free`, called `XMALLOC` and `XFREE`, and ensure easy integration of our Heap Memory Manager with any user application.

Implementing an easy-to-integrate command-line debugger API and various commands for displaying heap memory usage statistics and manipulating heap memory at run-time using command-line.

# Chapter 2

## Methodology

### 2.1 Introduction

This chapter contains a description of the strategies, concepts, and algorithms used by our HMM, as well as some of the issues which arise in our implementation.

### 2.2 The Heap Memory Manager (HMM)

Let us start with the discussion of the internal working of our HMM, the heap memory management strategies used by it, and the API exposed to the user programs. The following subsections explain the various concepts and algorithms used by our HMM.

#### 2.2.1 Page Family Registration

The first step for any user program that wishes to use our HMM is page family registration. Here, we refer to the combination of a struct and its size as a **page family**. Thus, page families store information about the various structs for which the user wishes to allocate memory dynamically. Since a HMM itself cannot use glibc-provided `malloc` and `free`, it requests VM pages to store the registered page families. We call these pages **meta VM pages**, in order to distinguish them from **data VM pages**, which are used to satisfy user memory requests. These pages are organised as a linked-list, and a new VM page is requested and added at the head of the list whenever a new page family cannot be accommodated in the last page. This scheme is shown in Figure 2.1.

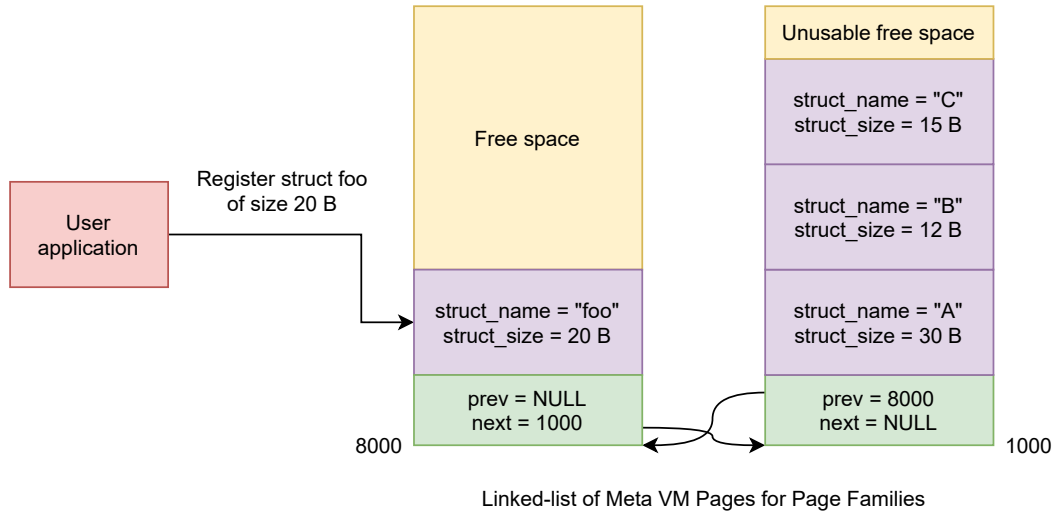


Figure 2.1: Linked-list of meta VM pages for storing page families. Structs A, B and C have already been registered. The user application requests to register a new struct named `foo` of size 20 bytes. Since the right page does not have enough capacity to accommodate a new page family block (shaded in purple), a new VM page is requested by the HMM and added at the head of the list, and the page family is stored in it. Note that each page family block has the same size.

## 2.2.2 Virtual Memory Page Organisation

The HMM, apart from allocating memory for user requests, also needs to store the data structures which are internally required by it to manage memory. Since a HMM itself cannot use glibc-provided `malloc` and `free` to satisfy its own memory needs, it requests and uses VM pages to store these data structures as well. We have categorised the VM pages into two types based on their usage: the data VM pages from which the user requests for memory are satisfied, and the meta VM pages, which are used to store internal data structures and meta-information required by the HMM. The HMM requests VM pages from the kernel by using the `mmap` system call, and returns these pages back to the kernel using the `munmap` system call, when they are not required anymore. We discuss the organisation of the data VM pages in this section (the organisation of the meta VM pages was described in the previous section).

Each data VM page is divided into *data blocks* and *meta blocks*, such that each data block is associated ('guarded') by one meta block. A data block is a contiguous chunk of virtual memory which is actually used to satisfy the

memory requests by the user program. Whenever a memory request is made, a pointer to a data block with size equal to the request size is returned to the user program. A meta block stores various information about the data block associated with it, including the data block size (**db\_size**), whether the data block is free or not (**is\_free**), the offset of the meta block in the VM page, and pointers which point to the next and previous meta blocks in the VM page (**next** and **prev** respectively). The **is\_free** flag is used to indicate that the associated data block is free, and it can be used to satisfy user requests. The **db\_size** attribute is required during deallocation of data blocks, to indicate how many bytes are actually being freed. The **next** and **prev** pointers are needed to facilitate **block merging** and **splitting**, which are discussed later. The organisation of a data VM page is shown in Figure 2.2.

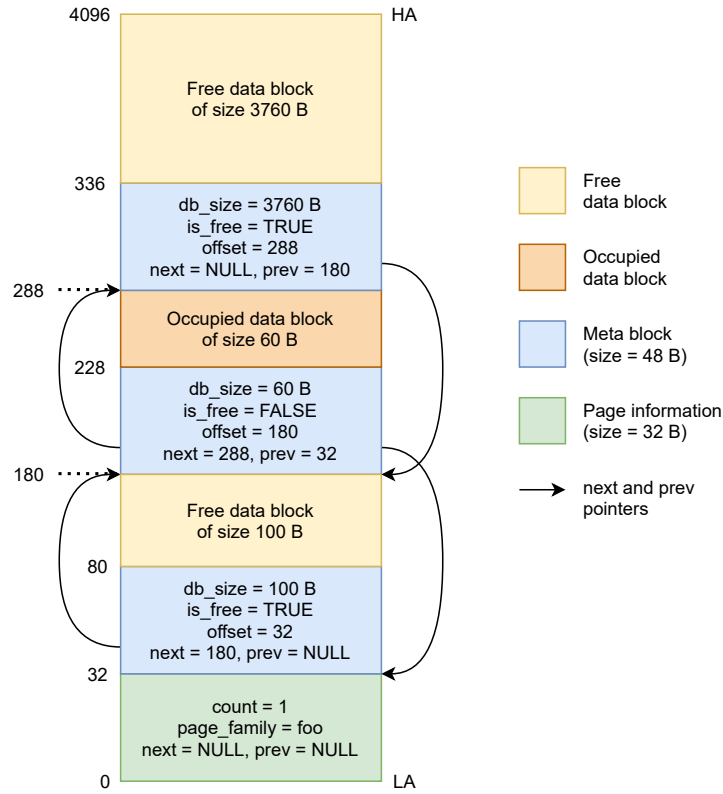


Figure 2.2: Organisation of a data VM page. The lowermost part stores information about the page, which is followed by meta blocks and data blocks. Here, it is assumed that the page size is 4096 bytes, each meta block occupies 48 bytes, and the page information occupies 32 bytes. The page is assumed to start at address 0 and the virtual memory address for each block is shown to the left.

The VM pages requested by the HMM are themselves organised in the form of a doubly-linked list, with the most recently allocated VM page at the head of the list. A data VM page list is associated with each struct (page family) registered by the user program (explained in the previous section). Apart from meta blocks and data blocks, a VM page also contains pointers to the next and previous VM pages in the list, a pointer to the page family with which this page is associated, and an attribute called `count`. Since a single VM page may not be sufficient to satisfy a large memory request (which exceeds the size of the VM page itself, for example, an array of structs), we may need to request more than one **contiguous** VM pages from the kernel for such requests. The attribute `count` stores the number of contiguous VM pages which constitute one single node in the data VM page linked-list. It is

needed during deallocation of the VM pages.

Apart from the page family information, each page family block also stores the pointer to the head of the linked-list of data VM pages (which are used to satisfy memory requests for this page family), and a pointer to a free block priority queue. This priority queue stores pointers to the meta blocks associated with the free data blocks in the data VM page list. The priority is based on the free data block size - the largest free data block is present at the head of the queue. We have implemented the priority queue using a binary heap, so that searching for the largest free block takes  $O(1)$  time, and insertion and deletion of free blocks takes  $O(\log n)$  time (where  $n$  is the number of free blocks). This queue is used to implement the **worst-fit** strategy for allocating free blocks, i.e., choosing the largest free block whenever a request for memory is made. The data VM page list and free block priority queue for a page family for struct `foo` is shown in Figure 2.3.

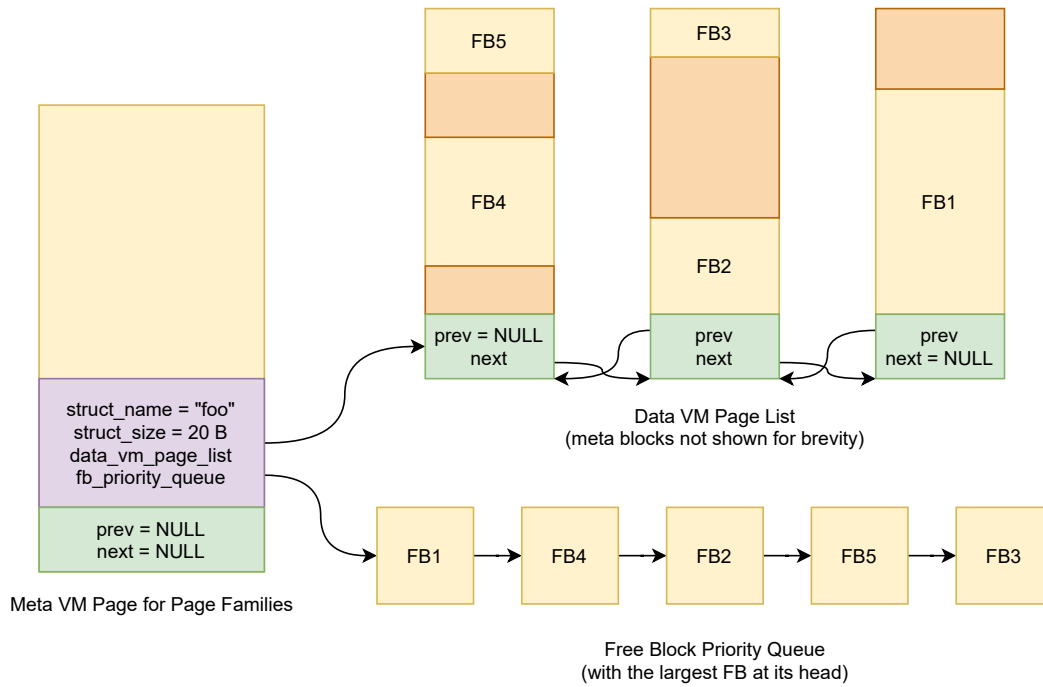


Figure 2.3: The data VM page list and free block priority queue associated with a page family for struct `foo`. The page family block (purple) contains pointers to these two data structures. Each data VM page also contains a back pointer to the page family, although not shown here. The largest free block (FB1 here) is present at the head of the priority queue.



### 2.2.3 Block Splitting and Merging

When the user program makes a memory request for  $n$  instances of some page family of size  $s$  bytes, the HMM needs to allocate a data block of size  $ns$  bytes and return its address to the user program. For this purpose, the HMM uses the free data block present at the head of the priority queue associated with that page family. Assuming that this block has a size  $b > ns$ , the HMM needs to **split** this block into a data block of size  $ns$ , which is present at the lower address and is used for serving the memory request, and a residual free block of size  $b - ns$ , which is present at the higher address. The residual free block is further divided into a data block and its associated meta block (note that if the size of the residual block is less than the size of a meta block, then this is a case of hard internal fragmentation, which is discussed in the next section). This newly created smaller data block is free and can be used for serving future memory requests. The **next** and **prev** pointers, the offsets, and the data block sizes of the appropriate meta blocks as well as the free block priority queue need to be updated so as to reflect the newly created free data block. An example of block splitting is shown in Figure 2.4.

**Block merging** is the process opposite to block splitting, which involves combining two adjacent free blocks into one larger free block. This situation arises when the user program makes a request to free an allocated data block. The program passes the pointer to the data block which needs to be freed, and the HMM determines the size of the data block from the meta block associated with it. The data block is then marked as freed. If the next and/or previous data blocks which are adjacent to this data block are also free, then they should be merged with this block, to create a larger free data block. One larger free block is more desirable than separate smaller free blocks, as the former can be used to serve larger memory requests as compared to the latter. Again, the fields in the appropriate meta blocks and the priority queue needs to be updated to reflect the newly created larger free data block. An example of block merging is shown in Figure 2.5.

### 2.2.4 Internal Fragmentation

Internal fragmentation is a well-known problem in memory management, which occurs when a small extra chunk of memory is allocated along with the actual chunk of the requested size during a memory request. This leads to small unused fragments of memory scattered across the VM page, which could have been used to satisfy a large memory request, had they been adjacent in the VM page, but cannot be used individually due to their small sizes. Since our HMM uses the worst-fit algorithm, it also suffers from internal

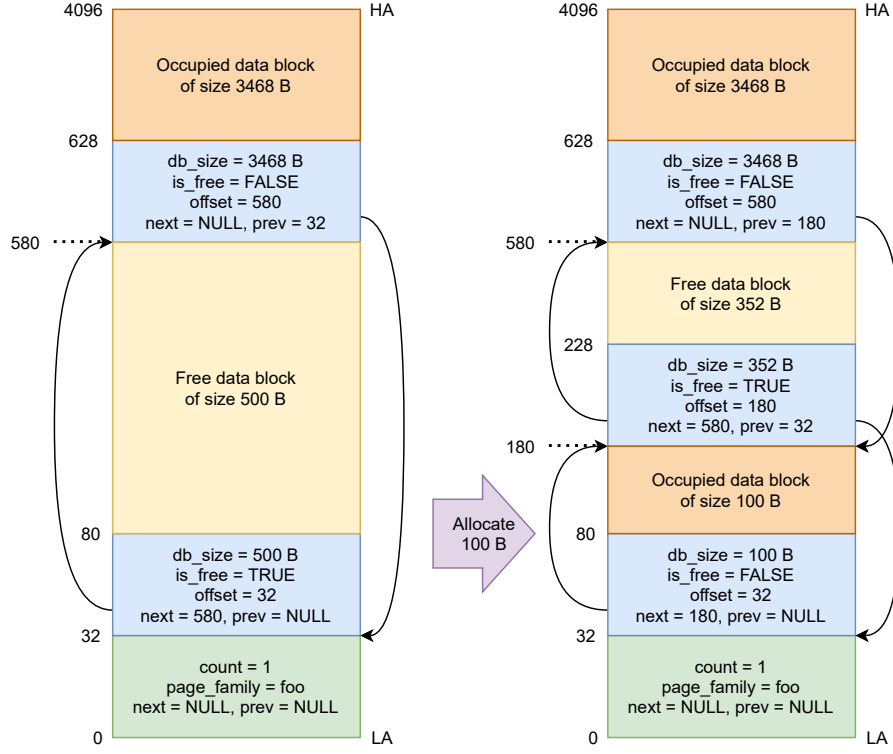


Figure 2.4: An example of block splitting. The data block (of size 500 bytes) at address 80 is chosen to serve a memory request for 100 bytes. It is split into a data block of size 100 bytes at address 80, which is returned to the user program and marked as occupied, and a residual free block of size 400 at address 180. This residual block is further divided into a smaller free data block of size 352 bytes at address 228, and its guardian meta block at address 180.

fragmentation. We can observe two types of internal fragmentation (IF) in our HMM, soft and hard IF. Note that IF can only occur during block splitting.

To define soft and hard IF, let us assume that a free data block of size  $b$  is present in a data VM page which is associated with a page family of size  $s$ , and the size of a meta block is  $m$ . Suppose a memory request of size  $n \leq b$  arrives and this free data block is chosen to serve the request. If  $b = n$ , then no residual free block is created. If  $b > n$ , the size of the residual free block formed after splitting will be  $r = b - n$ . **Soft IF** occurs when  $m \leq r < s + m$ . In this situation, the residual free block can be divided into a free data block and its associated meta block, but the former cannot be used to satisfy any

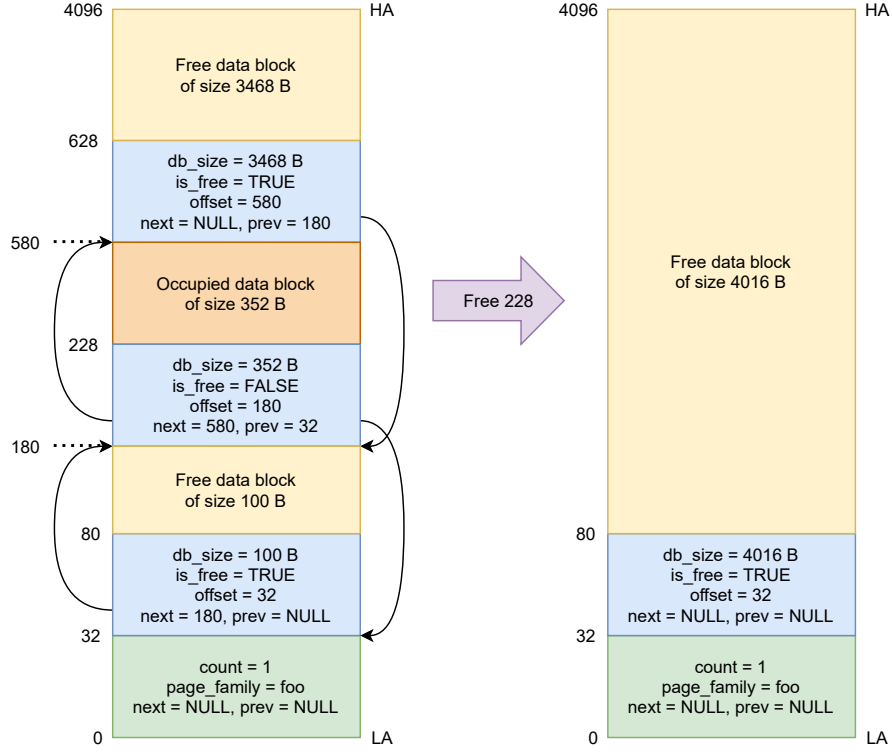


Figure 2.5: An example of block merging. The user program requests for the data block at address 228 to be freed. Since both the next and previous data blocks (at addresses 628 and 80 respectively) adjacent to this block are free, all the three blocks are merged together into one larger free block of size 4016 bytes. Note that the meta blocks guarding this data block and the next data block are destroyed in the process, and their space is merged into the resultant free data block.

requests for the page family of size  $s$ . These  $r$  bytes of the residual free block are therefore wasted, since they cannot be used for serving memory requests for the associated page family. **Hard IF** occurs when  $0 < r < m$ . In this situation, the residual free block is not even large enough to hold a meta block, and therefore, a free data block cannot be created inside this residual block. Note that even if the residual block is large enough to hold an instance of the page family, i.e.,  $s \leq r < m$ , it still cannot be used to satisfy a request since it does not have an associated meta block. Thus,  $r$  bytes are wasted in this case as well. Hard IF, however, has another problem, which is that the space which is wasted in hard IF can only be reclaimed when the data block due to which the unusable residual block was created is freed by the user program. On the other hand, the space wasted due to soft IF can be

reclaimed as soon as any data block which is adjacent (next or previous) to the unusable data block is freed by the user program. Thus, hard IF is more costly than soft IF. An example of hard and soft IF each is shown in Figure 2.6.

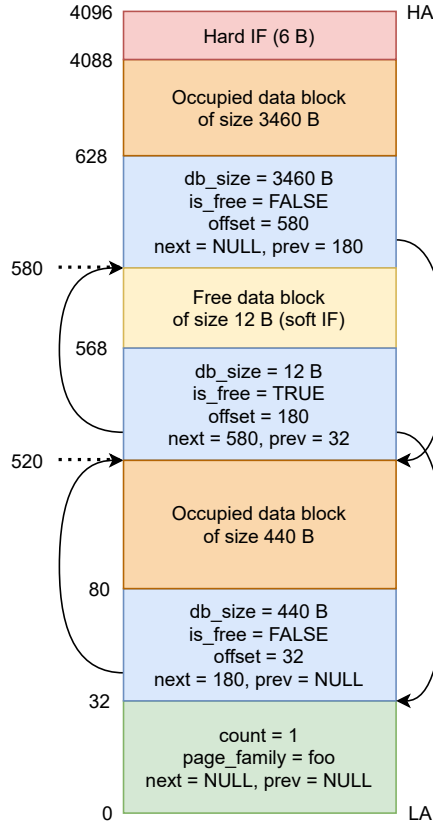


Figure 2.6: Examples of hard and soft IF. Assuming the size of page family `foo` to be  $s = 20B$  and meta block size to be  $m = 48B$ , the residual block (of size 6 B) at address 4088 suffers from hard IF, because it is not guarded by a meta block, and cannot accommodate one. It can only be reclaimed when the data block at 628 is freed. The free data block (of size 12 B) at address 568 suffers from soft IF, because it is guarded by a meta block, but it cannot accommodate an instance of the page family `foo` (of size 20 B). It can be reclaimed when either the data block at 628 or the one at 80 is freed.

## 2.2.5 XMALLOC & XFREE

The flow diagram of `XMALLOC` is shown in Figure 2.7.

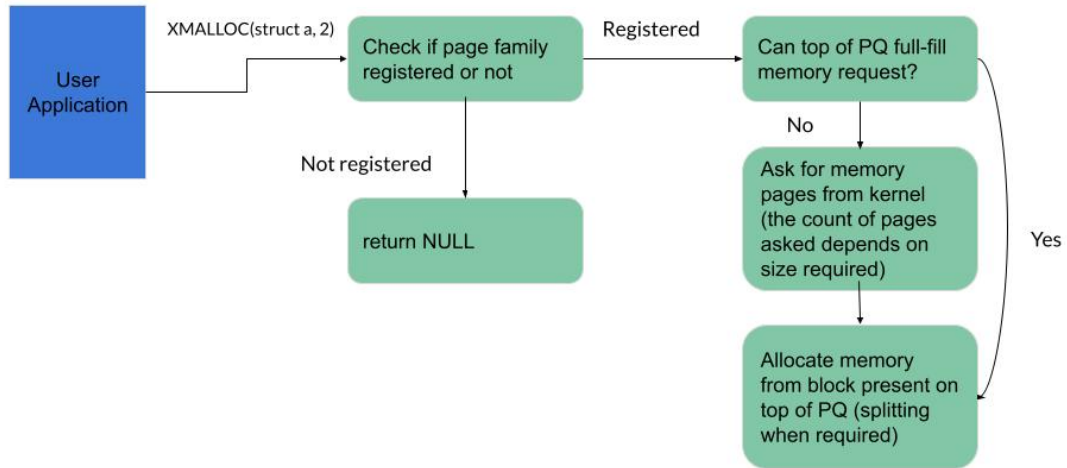


Figure 2.7: Flow diagram of our XMALLOC Implementation

The flow diagram of XFREE is shown in Figure 2.8.

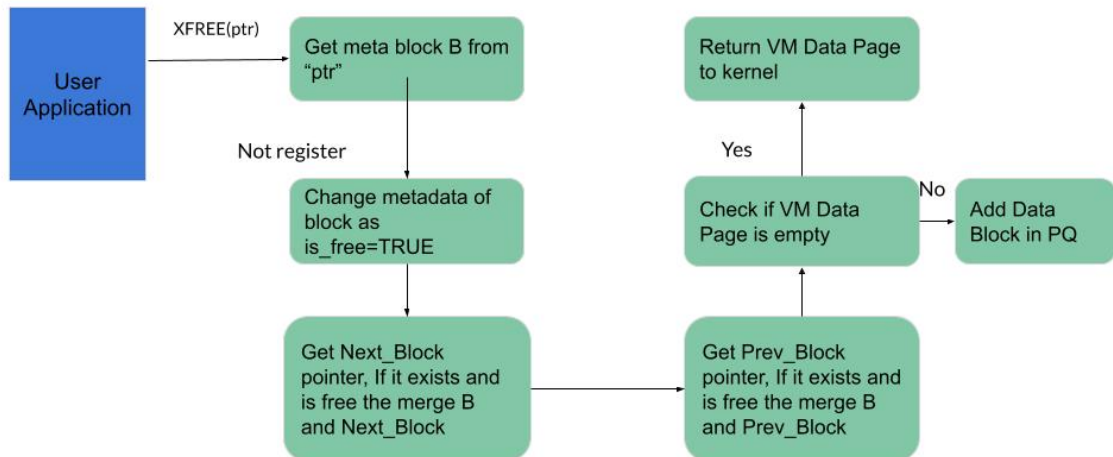


Figure 2.8: Flow diagram of our XFREE Implementation

## 2.3 Command-line Debugger for Programs using our HMM

We have developed an easy to use command-line interface API, which can be easily integrated into any C source program that uses our HMM. The API can be used as follows. The user program first needs to include the required HMM and CLI headers, and initialize them using simple function calls at the start of the main function. The CLI initialisation function spawns a new thread, which we will refer to as the CLI thread, while the user program keeps running on the main thread. The CLI thread is initially waiting on a mutex (binary semaphore) called `cli_mutex`. The programmer can then include `debug()` function calls at various points in the program, called **breakpoints**. Whenever this function is called, the main thread signals the `cli_mutex`, and itself waits on another mutex, called `main_mutex`. Now the CLI thread resumes its execution, and executes the prompt-read-execute cycle, just like a normal command interpreter or terminal. We have provided various commands specially designed for manipulating and displaying the heap memory layout of the process. The output of all the commands represents the current state of the program (after all the statements upto the current `debug()` function call have been executed). After the user issues a special command, `deb continue`, the CLI thread signals `main_mutex` and again waits on `cli_mutex`. Now the main thread can continue execution and subsequently call `debug()` again, as and when needed, and the cycle repeats. This interaction between the main thread and CLI thread is shown in Figure 2.9. All the commands provided by our CLI are explained along with their example usage in Chapter 4.

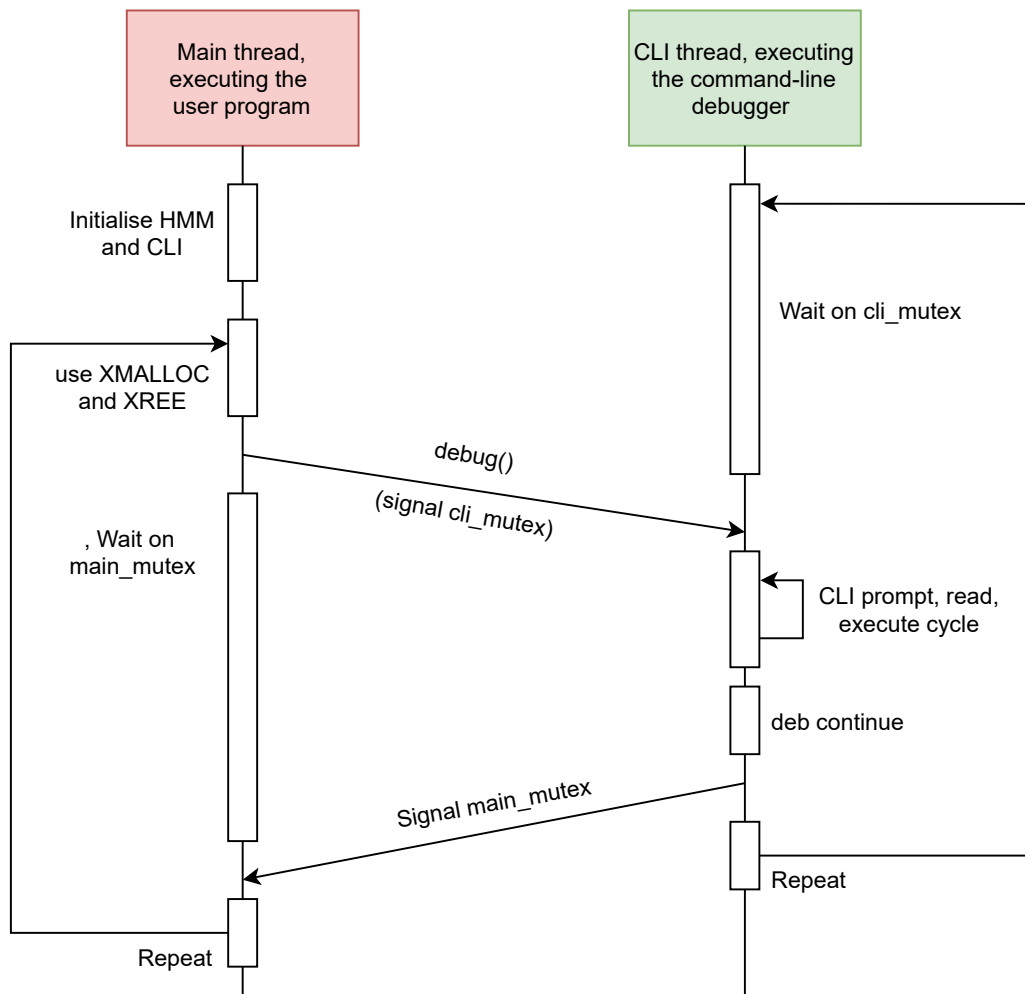


Figure 2.9: The interaction of the main thread and the CLI thread.

# Chapter 3

## Experimental Setup

Listing 3.1: User Space Application

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct a {
    int var1;
};

struct b {
    int var2;
};

int main() {

    int wait;
    void *ptr1 = (void *)malloc(2*sizeof(struct a));

    scanf("%d",&wait);
    free(ptr1);
    scanf("%d",&wait);

    ptr1 = (void *)malloc(3*sizeof(struct a));

    void *ptr2 = (void *)malloc (3* sizeof(struct b));

    return 0;
}
```



Listing 3.2: Integration with CLI and our memory manager

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Header file for memory manager and CLI
#include "memmgr.h"

struct a {
    int var1;
};

struct b {
    int var2;
};

int main() {

    cli_init(); //Initializing CLI
    mgr_init(); //Initializing memory manager

    //Page family registrations of the structs used
    MGR.STRUCT_REG(struct a);
    MGR.STRUCT_REG(struct b);

    // replacement of "wait" in above User Space Application
    // Waits until a command is issued by the user
    debug();

    //Using XMALLOC() instead of malloc()
    void* ptr1 = XMALLOC(struct a, 2);
    debug();

    //Using XFREE() instead of free()
    XFREE(ptr1);
    debug();


    ptr1 = XMALLOC(struct a, 3);
    debug();
}
```

# Chapter 4

## Presentation of Results

### 4.1 Introduction

To show the results, we will use program 3.2. The program has ended its execution before the first use of function **debug()** after it has been run. After running the program interface starts with:



```
Virtual page size = 4096 bytes  
memmgr@debug $
```

Figure 4.1: First line shows the size of virtual page which will be cached by Memory Manger from Kernel MMU. And cursor shows the stating point of CLI where we issue various commands.

So, in this segment, we'll look at the commands used by the CLI and how they're interpreted.

### 4.2 deb show mem

This command shows the current status of all the page families registered in Virtual Memory(VM) Page and also shows the number of VM Pages currently in use. After running the command we get the following output.

```

memmgr@debug $ deb show mem

Page Size = 4096 Bytes
vm_page_family : struct a, struct size = 4

vm_page_family : struct b, struct size = 4

# Of VM Pages in Use : 0 (0 Bytes)
Total Memory being used by Memory Manager = 0 Bytes

```

Figure 4.2: Both the structs, i.e **struct a** and **struct b** have been registered till the first debug. No VM pages have been allocated to them as no **XMALLOC** has been called until the first debug()

To run the program after the first debug() run the command: **deb continue**. This command executes the program till next debug. And after first debug our program allocates 2 instances of **struct a**. So running "deb show mem" again gives the following output:

```

memmgr@debug $ deb continue
memmgr@debug $ deb show mem

Page Size = 4096 Bytes
vm_page_family : struct a, struct size = 4
  VM Page 0:
    next_page = (nil), prev_page = (nil)
    Page family = struct a
    Number of contiguous vm pages = 1 (4096 bytes)
      0x7efff66cb6050 Block 0  ALLOCATED  block_size = 8      offset = 32      prev_block = (nil)      next_block = 0x7efff66cb6088
      0x7efff66cb6088 Block 1  F R E E D  block_size = 3960  offset = 88      prev_block = 0x7efff66cb6050  next_block = (nil)

vm_page_family : struct b, struct size = 4

# Of VM Pages in Use : 1 (4096 Bytes)
Total Memory being used by Memory Manager = 4096 Bytes

```

Figure 4.3: 2 instance of **struct a** have been allocated using **XMALLOC()** due to which a VM page has been requested from Kernel MMU and has been appended to the page family of **struct a**.

This command shows the status of all the VM pages in the page family starting with the first page(index 0) along with fields containing addresses of next and prev page. Figure 4.3 also has a field with name: **number of contiguous vm pages**. This field represents the number of contiguous VM pages allocated to satisfy this request. Following this field is the status of each block in the particular VM page. Fields in the status of each block are: address of the data/free block, whether the memory block is free or allocated, size of the block (in this case it is  $4*2 = 8$  for first block), offset of the meta block associated with this data/free block in this page, address of the next and previous data/free block.

Therefore in this case we get the number of VM pages in use as 1, since it is the VM page used for allocating 2 instances of **struct a**.

A variant of this command is: **deb show mem "[struct\_name]"** where instead of showing memory status for all page families, it shows memory status of the page family of specified structure.

### 4.3 deb register “[struct\_name]” [struct\_size]

An example use of the command is shown in figure 4.4. Also *deb show mem* is used to show the registration of the struct.

```
memmgr@debug $ deb register "struct c" 20
memmgr@debug $ deb show mem

Page Size = 4096 Bytes
vm_page_family : struct a, struct size = 4
  VM Page 0:
    next_page = (nil), prev_page = (nil)
    Page family = struct a
    Number of contiguous vm pages = 1 (4096 bytes)
      0x7f5a06d9c050 Block 0  ALLOCATED  block_size = 8      offset = 32      prev_block = (nil)
    next_block = 0x7f5a06d9c088
      0x7f5a06d9c088 Block 1  F R E E D  block_size = 3960   offset = 88      prev_block = 0x7f5a06d9
    c050 next_block = (nil)

vm_page_family : struct b, struct size = 4

vm_page_family : struct c, struct size = 20

# Of VM Pages in Use : 1 (4096 Bytes)
Total Memory being used by Memory Manager = 4096 Bytes
```

Figure 4.4: Example of registration

### 4.4 deb allocate “[struct\_name]” [quantity]

An example use of the command is shown in figure 4.5. Also *deb show mem* is used to show the allocation of the memory.

```

memmgr@debug $ deb allocate "struct c" 2
Successfully allocated memory for 2 instances of struct c
memmgr@debug $ deb show mem

Page Size = 4096 Bytes
vm_page_family : struct a, struct size = 4
  VM Page 0:
    next_page = (nil), prev_page = (nil)
    Page family = struct a
    Number of contiguous vm pages = 1 (4096 bytes)
    0x7f5a06d9c050 Block 0  ALLOCATED  block_size = 8      offset = 32      prev_block = (nil)
    next_block = 0x7f5a06d9c088
    0x7f5a06d9c088 Block 1  F R E E D  block_size = 3960   offset = 88      prev_block = 0x7f5a06d9
c050 next_block = (nil)

vm_page_family : struct b, struct size = 4

vm_page_family : struct c, struct size = 20
  VM Page 0:
    next_page = (nil), prev_page = (nil)
    Page family = struct c
    Number of contiguous vm pages = 1 (4096 bytes)
    0x7f5a06d9a050 Block 0  ALLOCATED  block_size = 40      offset = 32      prev_block = (nil)
    next_block = 0x7f5a06d9a0a8
    0x7f5a06d9a0a8 Block 1  F R E E D  block_size = 3928   offset = 120     prev_block = 0x7f5a06d9
a050 next_block = (nil)

# Of VM Pages in Use : 2 (8192 Bytes)
Total Memory being used by Memory Manager = 8192 Bytes

```

Figure 4.5: Example of allocation

## 4.5 deb free [memory\_address]

An example use of the command is shown in figure 4.7 by freeing the block of **struct c** allocated in the previous section. Also *deb show mem* is used to show that the memory registered is freed.

```

memmgr@debug $ deb free 0x7f5a06d9a050
Memory freed at address: 0x7f5a06d9a050
memmgr@debug $ deb show mem

Page Size = 4096 Bytes
vm_page_family : struct a, struct size = 4
  VM Page 0:
    next_page = (nil), prev_page = (nil)
    Page family = struct a
    Number of contiguous vm pages = 1 (4096 bytes)
    0x7f5a06d9c050 Block 0  ALLOCATED  block_size = 8      offset = 32      prev_block = (nil)      next_block = 0x7f5a06d9c088
    0x7f5a06d9c088 Block 1  F R E E D  block_size = 3960   offset = 88      prev_block = 0x7f5a06d9c050 next_block = (nil)

vm_page_family : struct b, struct size = 4

vm_page_family : struct c, struct size = 20

# Of VM Pages in Use : 1 (4096 Bytes)
Total Memory being used by Memory Manager = 4096 Bytes

```

Figure 4.6: Example of freeing memory

## 4.6 deb show blocks

This command represents the count of total blocks (TBC), count of free blocks (FBC), count of occupied blocks (OBC), total application memory Usage (AppMemUsage) which is the sum of size of meta blocks + occupied data blocks, number of bytes undergoing hard fragmentation(HARD\_FRAG) and number of bytes undergoing soft fragmentation(SOFT\_FRAG). These fields are shown for each page family registered. The below figure illustrates these fields:

```
memmgr@debug $ deb show blocks
struct a      TBC : 2      FBC : 1      OBC : 1      SOFT_FRAG : 0      HARD_FRAG : 0      AppMemUsage : 56
struct b      TBC : 0      FBC : 0      OBC : 0      SOFT_FRAG : 0      HARD_FRAG : 0      AppMemUsage : 0
```

Figure 4.7: Example of freeing memory

# Chapter 5

## Limitations, Conclusion and References

### 5.1 Introduction

This chapter presents a conclusion to the project, we shall mention the limitations, conclusion and references of our project.

### 5.2 Limitations

During requirement of free data block, we performed worst fit algorithm (i.e. get the data free block having the largest size), the splitting is performed. It may happen that the split may cause partition of data block such that left out partition may not be used to store data, which is called fragmentation. So, our method does not avoid fragmentation. It may lead to various small chunks of memory being unused for certain calling order of `XMALLOC` and `XFREE`.

If various `malloc` calls are made for memory size less than meta block then more than 50% of data memory page will be used to store meta data only, which also shows poor performance.

### 5.3 Conclusions

We have used system calls (*mmap* and *munmap*) which are used by `malloc` and `free` for creating our `XMALLOC` and `XFREE`. For various memory require-

ments (page family registration and priority queue) we have used the memory which we get directly from kernel MMU and not any inbuilt C call. We have successfully implemented merging and splitting of blocks, reclaiming of fragmented memory is must if we cannot avoid fragmentation. Also, we have implemented priority queue and using it implemented worst fit algorithm for returning memory address to **XMALLOC**. The memory manager must be implemented with good efficiency, say if anyone manager using linear like data-structure for priority queue can make the execution time of **XMALLOC**  $O(n)$ ,  $n$  is number of free blocks, but as we have used tree like data-structure for priority queue the execution time of **XMALLOC**  $O(\log(n))$ , making the manager feasible for use.

The command line integration providing a great deal of flexibility, during memory usage statistics helps the user to update the heap memory usage. For example, at the end of routine calls of user defined function the user can debug the memory statistics in command line and if any heap memory is not free/will not be used in future, can be freed making execution of user application smoother.



## References

- (1) <https://www.linuxjournal.com/article/6390>
- (2) <https://man7.org/linux/man-pages/man2/mmap.2.html>
- (3) [https://www.tutorialspoint.com/operating\\_system/os\\_memory\\_management.htm](https://www.tutorialspoint.com/operating_system/os_memory_management.htm)
- (4) <https://www.geeksforgeeks.org/program-worst-fit-algorithm-memory-management/>