

LINUX MEMORY MANAGER

Project report submitted in partial fulfillment of the
requirement for the degree of Bachelor of Technology
in

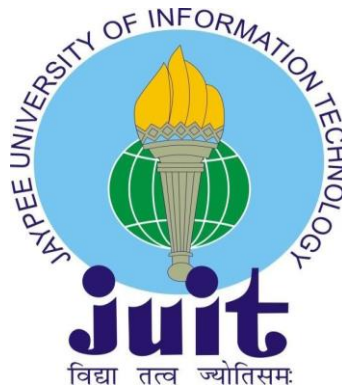
Computer Science and Engineering

By

KASHIK BAGLWAN 191255

Under the supervision of
Dr. DEEPAK GUPTA

to



Department of Computer Science & Engineering and
Information Technology

**Jaypee University of Information Technology Waknaghat, Solan
173234, Himachal Pradesh**

Candidate's Declaration

I hereby declare that the work presented in this report entitled “**Linux Memory Manager**” in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science and Engineering** submitted in the department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from August 2022 to May 2023 under the supervision of **Dr. Deepak Gupta** of Computer Science and Technology. The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Kashik Baglwan (191255)

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

Supervisor Name: Prof. Dr. Deepak Gupta

Designation: Professor

Department Name: Computer Science and Information Technology

Dated: 01-05-2023

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT
PLAGIARISM VERIFICATION REPORT

Date:

Type of Document (Tick): PhD Thesis M.Tech Dissertation/ Report B.Tech Project Report Paper

Name: _____ Department: _____ Enrolment No _____

Contact No. _____ E-mail. _____

Name of the Supervisor: _____

Title of the Thesis/Dissertation/Project Report/Paper (In Capital letters): _____

UNDERTAKING

I undertake that I am aware of the plagiarism related norms/ regulations, if I found guilty of any plagiarism and copyright violations in the above thesis/report even after award of degree, the University reserves the rights to withdraw/ revoke my degree/report. Kindly allow me to avail Plagiarism verification report for the document mentioned above.

Complete Thesis/Report Pages Detail:

- Total No. of Pages =
- Total No. of Preliminary pages =
- Total No. of pages accommodate bibliography/references =

(Signature of Student)

FOR DEPARTMENT USE

We have checked the thesis/report as per norms and found **Similarity Index** at(%). Therefore, we are forwarding the complete thesis/report for final plagiarism check. The plagiarism verification report may be handed over to the candidate.

(Signature of Guide/Supervisor)

Signature of HOD

FOR LRC USE

The above document was scanned for plagiarism check. The outcome of the same is reported below:

Copy Received on	Excluded	Similarity Index (%)	Generated Plagiarism Report Details (Title, Abstract & Chapters)	
Report Generated on	<ul style="list-style-type: none"> • All Preliminary Pages • Bibliography/Images/Quotes • 14 Words String 		Word Counts	
			Character Counts	
		Submission ID	Total Pages Scanned	
			File Size	

**Checked by
Name & Signature**

Librarian

Please send your complete thesis/report in (PDF) with Title Page, Abstract and Chapters in (Word File) through the supervisor at plagcheck.juit@gmail.com

ACKNOWLEDGEMENT

Firstly, I would like to express our heartiest thanks and gratefulness to almighty god for his divine blessing makes it possible to complete the project work successfully. I am really grateful and wish our profound indebtedness to Dr. Deepak Gupta, Professor of the department of CSE & IT, Jaypee University of Information Technology, Waknaghat. Deep knowledge & keen interest of my supervisor has helped me a lot to carry out this project. His endless patience, scholarly guidance, continual encouragement, constant and energetic supervision, constructive criticism, valuable advice, reading many inferior drafts and correcting them at all stages have made it possible to complete this project. In this unique situation, I might want to thank the various staff individuals, both educating and non-instructing, which have developed their convenient help and facilitated my undertaking.

Finally, I must acknowledge with due respect the constant support and patients of my parents.

Regards,

Kashik Baglwan 191255

CSE | JUIT

Table of Contents

1	Chapter 1-Introduction	1
1.1	Introduction	1
1.2	Problem Statement	2
1.3	Objectives	3-5
1.4	Methodology	7
1.5	Organization	8
2	Chapter 2-Literature Survey	9-16
3	Chapter 3-System Development	17
3.1	Architecture and Design	17
3.2	Algorithm	18-20
3.3	Analytical Development	20-21
4	Chapter 4-Performance Analysis	22
4.1	VM Page De(Allocation)	22-24
4.2	Page Family Registration	24-26
4.2.1	Page Family Instantiation	26-28
4.3	Meta Blocks and Data Blocks	29-30
4.4	Block Merging and Splitting	31-32
4.4.1	Block Merging	32-34
4.5	VM Page Management	35-37
4.5.1	VM Page Deletion	38

4.5.2	VM Page Insertion	38-39
4.6	Free Data Block Management	39-41
4.7	Final Push-Implement Xmalloc	42
4.7.1	Xmalloc Algorithm Discussion	42
4.7.2	Xmalloc Implementation	42-45
4.7.3	Hard Internal Fragmentation	46
4.8	Testing Our Project	47
4.8.1	Explanation of Output	47-50
4.9	Implementation of Xfree	51
4.9.1	Algorithm Flowchart	51
4.9.2	Implementing Xfree	52-57
5	Chapter 5-Conclusions	58
5.1	Conclusions	58-59
5.2	Future Scope	60
6	References	61

LIST OF FIGURES

S.No.	Title	Page
1	Fig 1.3. Hard and Soft If	6
2	Fig 1.4. Flow Chart of Project Phases	8
3	Fig 2.1. Heap Block	10
4	Fig 2.2. Heap Blocks Rearrangement	11
5	Fig 3.1. Project Design	18
6	Fig 4. Code De(allocation)	23
7	Fig 4.1 Output	24
8	Fig 4.2. Example of User Application	25
9	Fig 4.2.1. Code Snippet Phase 2	27
10	Fig 4.2.2 Code Snippet Phase 2	27
11	Fig 4.2.3 Code Snippet Phase 2	28
12	Fig 4.3.4 Code Snippet Phase 2	28
13	Fig 4.3 Meta Block	30
14	Fig 4.3.1 Meta Block	30
15	Fig 4.4 Xcalloc	31
16	Fig 4.4.1 Block Merging	33
17	Fig 4.4.2 Code Snippet Block Merge	34
18	Fig 4.5 VM Page Management	36
19	Fig 4.5 a Mm.h File	36
20	Fig 4.5 b Looping Macros	37
21	Fig 4.5 c Iterating Across	37

22	Fig 4.5 d Allocations Deallocations	37
23	Fig 4.5.1 VM Page Insertion	38
24	Fig 4.5.2 VM Page Deletion	39
25	Fig 4.6 Block Management	41
26	Fig 4.7 Flow Chart	42
27	Fig 4.7.2. Hard and Soft IF	44
28	Fig 4.8.1 Output of Xcalloc Implementation	46
29	Fig 4.9 Flow Chart	50
30	Fig 4.9.2 a Code Xfree	51
31	Fig 4.9.2 b Output Xfree	57
32	Fig 4.9.2 c Output Xfree	57

ABSTRACT

LMM acquires and frees memory from kernel space on behalf of the application in virtual memory pages. The system calls `mmap()` and `munmap()` were used for this purpose. Cache vm pages and use them as a reservoir for future memory requests issued by application until the future memory pages are completely exhausted. The vm side kernel when the application frees enough memory that the vm side has noy occupied or allocated for use by the application. Linux memory management subsystem is responsible, as the name implies, for managing the memory in the system. This includes implementation of virtual memory and demand paging, memory allocation both for kernel internal structures and user space programs, mapping of files into processes address space and many other cool things. The goal of this project is to help you control the memory requirements of your processes by creating your own heap memory manager. By reducing or eliminating internal and external fragmentation issues and avoiding pointless system calls from allocating/deallocating memory, the memory manager enhances process speed. The LMM may also show users statistics regarding memory utilization by each process structure. These statistics can be used to examine how much memory your application uses and offer suggestions for improving how much memory your processes need. Her LMM can also identify memory leaks.

CHAPTER-1 INTRODUCTION

1.1 Introduction

This project is regarding designing your own heap memory manager which will manage the process's memory requirement. Memory manager will get rid or minimize the internal and external fragmentation problems, and boost performance of the process by preventing unnecessary system calls for allocating/releasing the memory. Not only that, linux memory manager (LMM) can display the user with the statistics regarding the memory usage by each structure of the process. From this stat, application memory usage can be analyzed and can provide hints to further optimize the memory requirements of the process. Memory leak can also be discovered using Linux Memory Manager (LMM).

LMM acquires and frees memory from kernel space on behalf of the application in virtual memory pages. The system calls `mmap()` and `munmap()` were used for this purpose. Cache vm pages and use them as a reservoir for future memory requests issued by application until the future memory pages are completely exhausted. The vm side kernel when the application frees enough memory that the vm side has occupied or allocated for use by the application.

Linux memory management subsystem is responsible, as the name implies, for managing the memory in the system. This includes implementation of virtual memory and demand paging, memory allocation both for kernel internal structures and user space programs, mapping of files into processes address space and many other cool things.

Linux memory management is a complex system with many configurable settings.

The goal of this project is to help you control the memory requirements of your processes by creating your own heap memory manager. By reducing or eliminating internal and external fragmentation issues and avoiding pointless system calls from allocating/deallocating memory, the memory manager enhances process speed. The LMM may also show users statistics regarding memory utilization by each process structure. These statistics can be used to examine how much memory your application uses and offer suggestions for improving how much memory your processes need. Her LMM can also identify memory leaks.

LMM that, on behalf of applications in virtual memory pages, requests and releases memory from kernel space. For this, the system calls `mmap()` and `munmap()` were employed. `vm` pages should be cached and used as a until all of the `vm` pages are used, the application's memory requests will be stored in a reservoir. When the programmer releases enough memory that the `vm` side has not used or allocated for use by the application, the `vm` side exits to the kernel.

1.2 Problem Statement

This project is regarding designing own Heap memory manager which will manage the process's memory requirement. Memory manager will get rid or minimize the internal and external fragmentation problems, and boost performance of the process by preventing unnecessary system calls for allocating/releasing the memory. Not only that, Linux memory manager (LMM) can display the user with the statistics regarding the memory usage by each structure of the process. From this stat, application memory usage

can be analyzed and can provide hints to further optimize the memory requirements of the process. Memory leak can also be discovered using Linux Memory Manager (LMM).

The memory manager will be able to:

- Allocate and free the memory.
- Catch memory leaks.
- Address memory fragmentation problems.

1.3 Objectives

The main objective of this project is to successfully create a linux based heap memory manager in c.

The memory manager will be able to:

- Allocate and free the memory.

This will be done using `xmalloc()` and `xfree()` functions for allocation and freeing respectively. The application will generate a `xmalloc()` call to our Linux memory manager, the LMM will allocate the required memory by the application in the form of blocks (meta and data blocks).

- Catch memory leaks.

Our LMM will be able to detect any memory leaks in the system. Memory leaks usually occur when the user the LMM creates memory in the form of virtual memory pages and forgets to return it to the kernel. The same can occur if the memory allocated to the application by our LMM is not freed or returned back to it.

- Address memory fragmentation problems.

LMM will also address the issues related to fragmentation hard internal fragmentation (not enough space for meta block) and soft internal fragmentation (not enough space for data block).

The case one and case two both show the problem of internal fragmentation the data block which is grey in color is block is subjected to the problem of internal fragmentation it means that this portion represents some memory which is internally fragmented and is not usable and the same goes with the second diagram.

So, briefly describe the difference between the two internal fragmentation is that when you take the free data block which undergo split of size equal to 66 bytes then the internal fragment in memory which we opt in was eighteen bytes and this internally fragmented memory was guarded by the meta block. So, when internally fragmented memory is guarded by a block then such type of fragmentation is called soft internal fragmentation.

Whereas if we assume that the free data block which undergoes split was of size forty-six bytes and twenty bytes of memory was claimed by the application then after the split the amount of memory which was left was not big enough to accommodate it in a meta block. So, it produced the internally fragmented memory which was not guarded by a meta block. Such type of internal fragmentation is called hard internal fragmentation.

When our free data block undergoes a split, internal fragmentation can be produced and that internal fragmentation could be either soft internal fragmentation or it could be hard internal fragmentation.

soft internal fragmentation happens as a result of splitting free data block of size 66 Bytes a new meta block MB two dash was produced because internally fragmented memory was guarded by a new meta block. So, the link between these meta blocks will be arranged as usual. In other words, the soft internally fragmented memory is just treated as any other free data block.

However, it's a different thing that the size of this internally fragmented memory is so small that it cannot be used for any memory allocation by the application for the given page family. You can see in the diagram number one there is no special change in the linkages between the meta blocks but this is not true with the hard internal fragmentation. In the hard internal fragmentation, the linkages are between meta blocks since this is internally fragmented memory and it is not guarded by its own meta block therefore the meta blocks on the either side of this internally fragmented memory that is meta block number three and meta block number two will have linkages bypassing this hard internal fragmented memory.

So, when we will guide an api to handle the blocks splitting, we need to consider that if fragmentation is produced whether the fragmentation is a type soft internal fragmentation or whether the fragmentation is of type hard internal fragmentation and accordingly we have to manage the linkages between the meta blocks and this virtual memory data page.

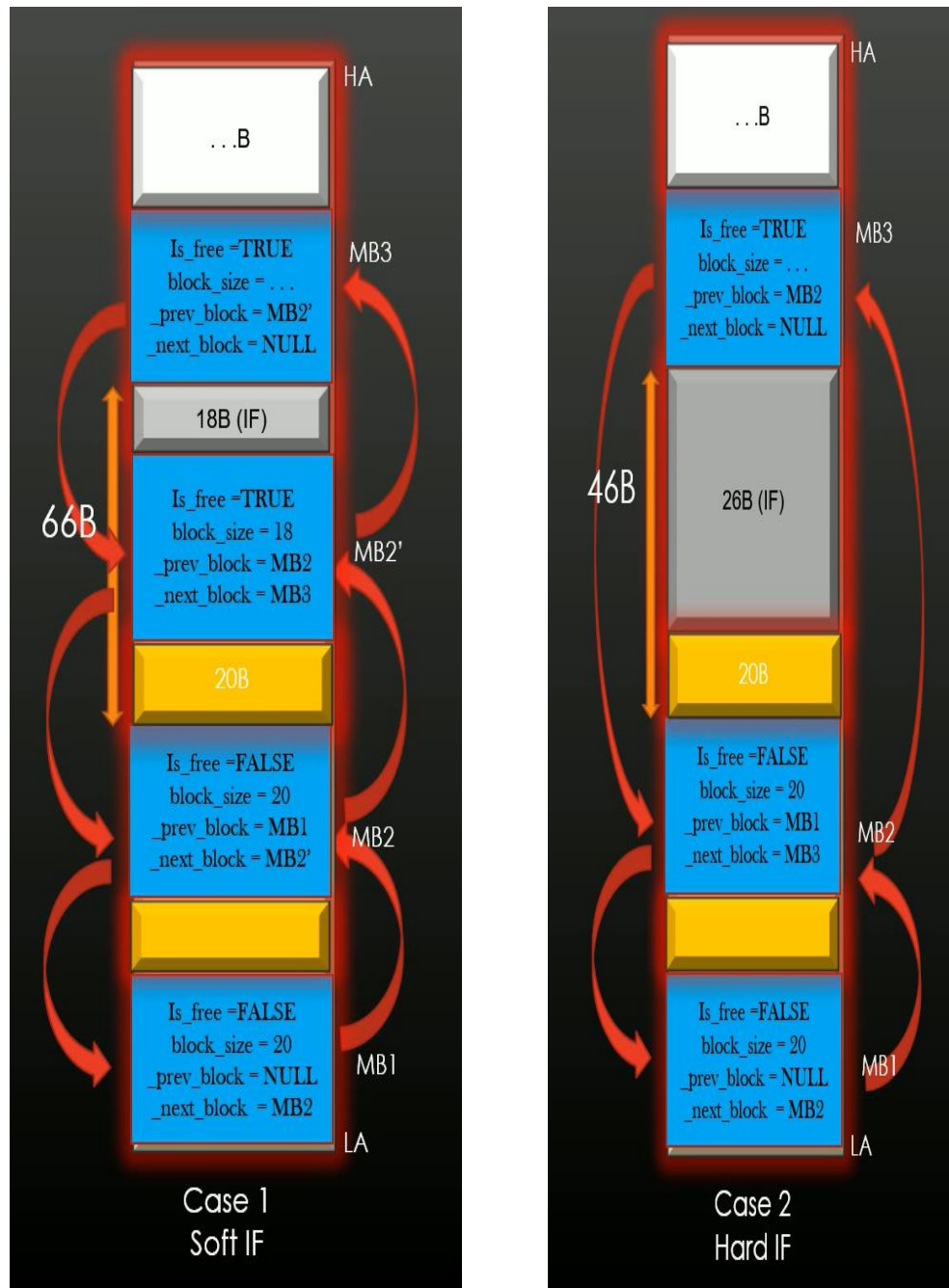


Fig 1.3b Hard and Soft If

1.4 Methodology

Methodologies used in this project are purely based on the knowledge of operating systems. Apart from this the concepts of data structures and been implemented in c language to meet the needs and requirements of the project.

The Linux memory manager constantly contacts with the system's kernel for the allocation and deallocation of virtual memory pages for fixed sizes. Instead of dealing with small data block or independent data blocks, The LMM requests for complete virtual memory pages from the kernel and then performs multiple functions to split the pages into chunks of memory as per the requirements of the application which is using our Linux memory manager.

The application invokes the xalloc() calls to the LMM and LMM registers the size and name of the structures of the application in the form of page families, this process keep happening as many times as the xalloc() function is invoked. As a result the virtual memory pages keep registering the page families starting from bottom to top in a contiguous manner in order to provide information to the LMM to allocate memory to the application. These page families are accessed within the virtual memory pages with the help of the linked lists and a pointer to them. The Linux memory manager will also deal with the problems of fragmentation (hard and soft internal fragmentation).

The use of priority queues has been made to know the availability of the free blocks and their particular sizes. This was done for the management of free blocks in the virtual memory pages and how the new memory request

for the application has to be carried out (from the data block of largest free memory- naïve approach).

The concept of meta blocks and data blocks has been used for the allocation of memory to the application in the form of blocks of memory/data. The task of a meta block is to keep a record of the following or succeeding data block. This information includes the Boolean expression for Is_free, the size of the data block, the previous and the next pointers to the meta blocks.

1.5 Organization

The project has been organized in 10 phases and the methods used are various data structures such as linked list, doubly linked list, priority queues, etc. to fulfill needs of those particular phases.

The following flow chart depicts the different phases of the project and their order of progress in the respective shown manner.

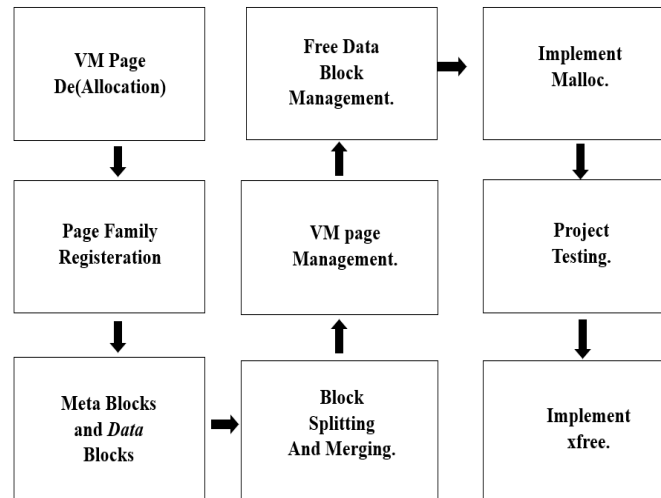


Fig 1.4 Flow chart of project phases

Chapter-2 LITERATURE SURVEY

Heap memory is also known as dynamic memory. Alternative to heap memory, sometimes known as "dynamic" memory, is local stack memory. Local storage is mostly automated. Local variables are automatically created after a function is called and released after the function completes. Heap memory has a number of variations.[1] In C or Java, programmers can directly request memory allocation by using the new operator. The size of this "block of memory" is usually determined by the size of the object you are automatically creating. This memory (object) block is allocated and will stay that way until it disappears for whatever reason. In some languages items on the heap only disappear when the programmer explicitly requests releasing.

The programmer therefore has more responsibility but also more control over the memory because it requires active management.[5] The principal cause of problems, known as "memory leaks," is deleting without releasing all references to a memory location. (In reality, many commercial C programmers include memory leaks that cause them to crash eventually.) java automatically deal spaces memory via garbage collection. to address the root of this problem. The drawback is that garbage collection is a laborious procedure that occurs sporadically.

The management of heap memory is done by heap memory manager. The main memory operations performed by heap memory manager:

- Allocation (using malloc and calloc)
- Deallocation (using free)
- Reallocation (using realloc)

A program's available memory space is called a heap. For use within the heap, a software may allocate memory "blocks" or sections of memory. A

program explicitly requests the allocation of a block of a specific size by invoking a heap allocation procedure. This new operator is available in C++ or Java. A reference to the memory block that the allocation function created on the heap, which was typically the size of the desired object, was returned. Consider a software that sends out three requests for memory allocation and allots space on the heap for three different gif pictures, each of which needs 1024 bytes. Three allocation requests later, the memory appears as follows:

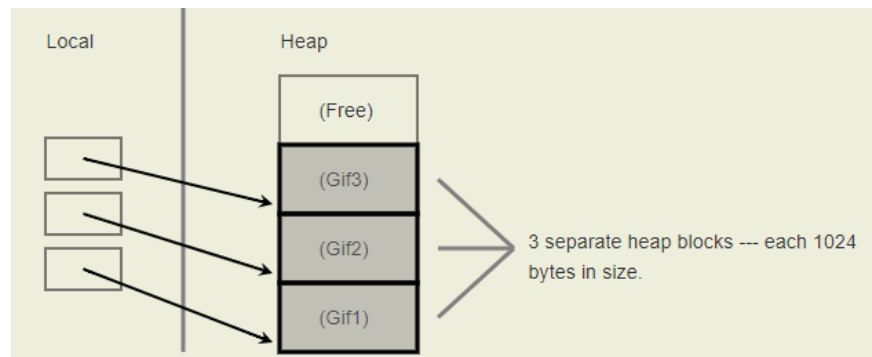


Fig.2.1 Heap block

Each allocation request returns a reference to the new block to your program along with a contiguous area of the heap of the requested size.[2] A program always manipulates its heap blocks by reference because each block is always referenced by reference and thus always acts as a "pointer". Since heap block references by convention lead to the block's base, they are sometimes referred to as "base address" pointers (lowest address byte). In this illustration, three blocks are progressively allocated from the heap's end, and each block is given the desired 1024 bytes. [5]In fact, as long as the blocks don't overlap and are at least the specified size, the heap manager can allocate them anywhere in the heap. A portion of the heap is always reserved for the program and is hence "in use." Other regions are "free" and available to satisfy allocation requests since they have not yet been defined. The heap manager keeps track of which portions of the heap are used and

for what purposes at any given moment using its own internal data structure. Each allocation request from the free memory pool is approved by the heap manager, who also updates private data structures to track which heap regions are in use.

Some programming languages demand that a memory block be explicitly released when a programmer has finished utilizing it. The block is identified as unused in this instance. When a space in Java is not mentioned, it is often "enabled." [5] This instructs java's garbage collector to clear this space. Implicit garbage collection releases unneeded memory blocks from the heap. The block's previously occupied memory space is now free and available to be used to fulfil upcoming allocation requests, according to the heap manager's update to private data structures. When the second of three blocks is released by the trash collector, the heap will appear as follows:

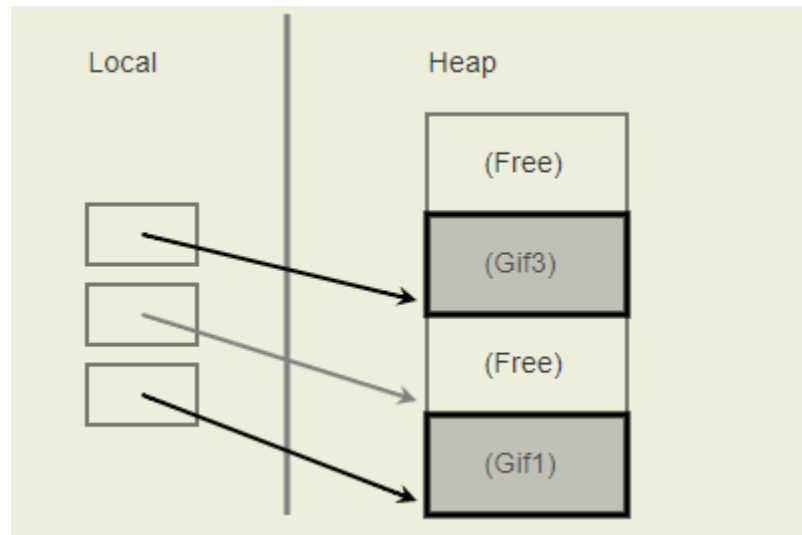


Fig.2.2 Heap block rearrangements

The reference keeps pointing to a block that has previously been released after being set free. Freed points are no longer accessible to programmers.

Programmers must make sure that outdated references to deallocated blocks are not followed in languages (like C++) that lack explicit deallocation and garbage collection.[2] The pointer is shown in grey because of this. Although present, the pointer cannot be used. To inform the garbage collector that an item is no longer in use, Java code naturally sets a reference to null or refers to another object. This is a major factor in the safety of Java references compared to C++ pointers.

The identification and correction of programming errors in large-scale software systems is a crucial resource-demanding procedure that may seriously jeopardize the initial production schedules and jeopardize the final quality of the delivered software products. [6]Memory corruption tends to spread quickly during execution, dispersing inaccurate material broadly and naturally interfering with program units' proper behavior, even in confined regions that only influence data content on a small scale. Such gradual data infection, which results in a cascade of incorrect data processing and disturbs the normal system behavior, may continue in silence for a protracted length of time before it is externally detectable.[4] After recurring system executions have confirmed the presence of symptoms for erroneous program behavior, the tracing of the offending code is started as a methodical debugging procedure. There are actually two crucial problems in this process that might make debugging procedures a nightmare:

- In most instances, the time lag between running malicious code and seeing fault symptoms is (a) non-zero, (b) unstable, and (c) not connected in a way that aids in bug discovery.
- When debugging at the source code level, the problem can only ever be replicated during regular program execution (example. real-time systems, multi-threaded systems).

The described work in this regard focuses on effectively automating the detection process utilizing built-in memory inspection and monitoring methods while decreasing the time between dynamic memory corruption episodes and runtime detection. Ignore it. [2]The proposed heap manager is an adapter to the native heap manager that incorporates extensive memory corruption prevention, and it resolves a few challenging programming issues to assist the practical deployment of the in development of real-world systems. The following are a some of the sophisticated features absent from the current heap manager recipe:

- Accommodating user address displacement for class instance arrays that are dynamically allocated by the compiler.
- Support for defect monitors, which simultaneously scan heap-memory areas for errors, thereby doing away with the necessity for manual insertion of thorough validation checks into the application source code.
- Delivery of a technique using delayed memory disposal and content monitoring to efficiently catch post-disposal memory overwrites.
- Defensive copies are used to encapsulate the capacity to detect corruption in the internally preserved memory-block information.
- Support for concurrent logging tools with minimum impact on allocation processes and a focus on off-line data analysis.
- Allocation and disposal operators were completely re-implemented to minimize conflicts in the event that third-party libraries already provided overloaded versions.

The proposed heap manager makes only minor changes to the source code from the perspective of programming usability, just needing allocation expressions to use the enhanced macro versions.[1] Using standard text-editor features, this task is easy automatable. For instance, in the MS Visual

Studio™ IDE, replacing the regular expressions `new:b+;(i);[.+];` (dynamic arrays), `new:b+;(i);(.*)`; (dynamic instances with function `Object() { [native code] }` arguments), and `new:b+;(i)`; (dynamic instances with default function `Object() { [native code] }`) with `DNEWARR(1,2)`, `DNEWCLASS(1,2)`, and `DNEW(1,` respectively, substitutes the defensive versions of the standard allocation expressions automatically.[4] The provided defensive features can then be implemented in the source code depending on the specific requirements for memory-error checking, such as adding explicit validation tests before pointer use, confirming the size of heap-memory regions before content write, or turning on particular defect monitors. The fact that text context tags before and after user memory blocks offer information on the source file, source string, allocation representation, and memory size () of the area of memory being scanned also makes it possible to scan memory more effectively. provide. Additionally, the additional contents of the text tag are used to immediately identify any memory space allocated within a program () that does not employ allocation macros (for instance, third-party code or files that have not yet been refactored).

The context tag "\$,0, \$,80" is present in an 80-byte block that was allocated using the default `new` expression, for instance.

The memory management system provided by the compiler is basically a base layer protected by the given heap manager.[5] The most well-known heap-memory checking feature included in current compilers is the delayed detection of post-disposal overwrites for recycled blocks by a check done before their reallocation. The latter is achieved using the following technique: memory blocks that have been disposed of are painted with a standard byte pattern, and when they are allocated again, their contents are

compared to this byte pattern; any discrepancy signals a post-disposal overwrite. It is obvious that even while this approach can identify overwrites, it cannot guarantee that the fault is always found (only if the offending block is finally recycled), and it does not provide direct defect detection (there is no indication of when the overwrite happened).

[2]The original heap manager used by C++ compilers is said to directly contain the capabilities included in the claimed defensive heap manager. In this manner, once the standard allocation functions include the defensive functionality and the code generation caters to introduce explicit validation tests for pointers engaged in de-reference (i.e., arrow or dot operator) expressions, the need for deployment of the allocation macros and the injection of explicit pointer validation checks in the source code is effectively eliminated. Since the development complexity of heap-manager adapters is significantly increased when advanced features for intensive bug defense are accommodated, as is the case with the reported development recipe, it is thought to be more effective if bug defense is eventually encapsulated in the standard compiler functionality.

The challenge of monitoring memory policies inside the memory management framework has been addressed, and we have developed a memory management interface that permits a variety of new memory technologies that are entering the market. The proposed method is fully implemented in user space. To a higher level in the software stack, the interface exposes the variety of functionality made accessible by linux system calls. When compared to a solution without buffer reuse, our technique reduces the total number of necessary system calls, improving overall speed. The technique allows for the efficient separation of client-specific requirements from the reused buffers.

Supports through a decorator interface allocation tracking, accounting, and profiling for all customers. [3] The middleware solution is heavily used by the HPC community to streamline scientific applications. These middleware options are anticipated to leverage the mankind interface for memory allocation, enabling user applications to benefit from cutting-edge memory technologies and policies without requiring changes to the application software or operating system.

Chapter-3 SYSTEM DEVELOPMENT

3.1 Architecture and Design

Now let us try to understand the overall design of our project in a little bit broader perspective. So, you can see that we can have any random user space process or application. Any C program can be served as a User space application and we have a standard glibc C library in which we have a standard implementation of malloc, free and other memory management related functions. Therefore, it's a standard C library. The third component of our project is the kernel memory management unit which is the part of the operating system. The virtual address space of the application running in user space resides in the kernel space.

It is the responsibility of the memory management unit in the kernel space to allocate and to deallocate virtual Memory pages to the user space application

We have three components in our project.

- user space application.
- The standard C library or in other words we can call this a standard C library as memory allocator in the context of our course.
- The third component is the kernel memory management unit for memory allocation and deallocation.

So, in a simple C program where you can always request memory of X bytes. There is no restriction on the value of X. You can request 10 bytes, 12 bytes, 14 bytes depending on the need. And it is the responsibility of the standard C library or in other words the memory allocators which is malloc or free to satisfy the need of the user space application.

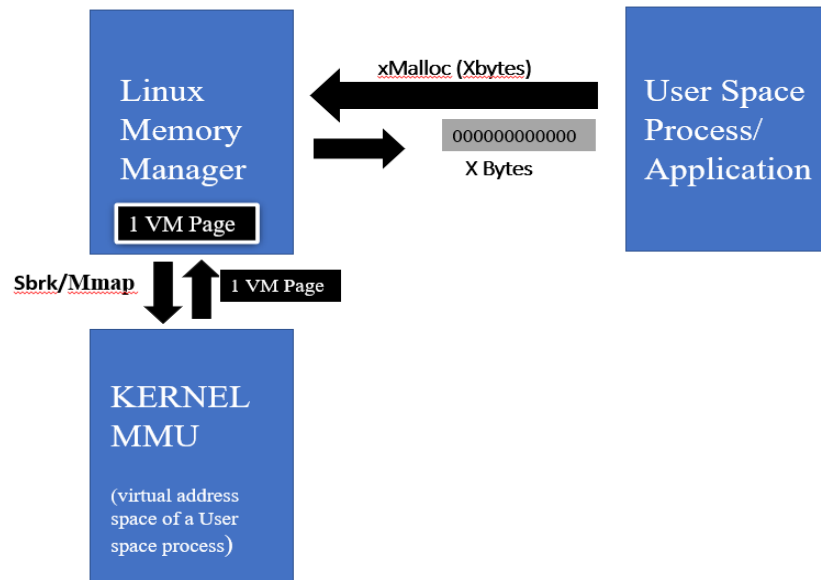


Fig.3.1 Project Design

3.2 Algorithm

In the diagram shown above, the Malloc implementation of the standard C library is providing Xbytes of memory to the user space application. So, memory allocation and deallocation between standard C library and kernel Management Unit happens only in units of pages sizes.

using System calls such as Sbrk and Mmap the Malloc and other related functions which are implemented in the standard C library request integral number of virtual memory pages from the kernel MMU. The question is that why not the standard C library he can request any x number of bytes from the kernel MMU for allocation or deallocation.

The answer is the system calls as sbrk or mmap are expensive system calls. The idea is that that the standard C library caches the vm page allocated by

kernel mmu and then allocates small chunks from this virtual memory page to a process. The virtual memory page is then cached by the standard C library.

And from this cached virtual memory page the standard C library then provided the number of bytes which are requested by the user space application. It means that when the user space application again issues the request to allocate a memory of say y bytes and if there is a scope that y bytes of memory can be provided from vm page, then this is standard C library does not request the kernel mmu to allocate another virtual memory page because the vm page which is already cast by the standard C library is sufficient enough to satisfy the new request of bytes to the use of space application. So this is how the whole algorithm works.

When a standard C library detects that process has freed all memory in a given virtual memory page. Glib c returns the page to the kernel mmu and this release of the memory is achieved using(sbrk/munmap) system calls. If user space application invokes certain free calls one after the another such that there is no memory left to allocate in vm page which is being used by this user space application then in such a condition the standard C library can return this vm page back to the kernel space.

Now defining the main objective of this project, in this project we will replace the standard memory management unit of the standard C library that is the set of memory related functions which are implemented in standard C library such as malloc or free with our own Linux memory manager

In other words we will have to implement our own version of malloc called xmalloc So basically we are reinventing the wheel instead of using standard C library for memory allocation to the user space application. We will write

our own memory manager which will have its own implementation of memory management related api such as malloc and free and provide the memory management services to the user space application.

3.3 Analytical Development

The project deals with analyzing issues related to memory leaks, fragmentation, memory allocation and deallocation to the user space applications.

If you develop a memory manager that dynamically allocates memory, you're also responsible for tracking any memory that you allocate whenever a task is performed, and for releasing that memory when you no longer need it. If you fail to track the memory correctly, you may introduce “memory leaks” or unintentionally write to an area outside of the memory space.

Conventional debugging techniques usually prove to be ineffective for locating the source of corruption or leaks because memory-related errors typically manifest themselves in an unrelated part of the program. Tracking down an error in a multithreaded environment becomes even more complicated because the threads all share the same memory address space.

The analysis tools support these four tasks:

- monitoring memory and resource consumption — you can view real time data about memory usage for a system or a single process and about resource usage for a process; this feature helps you determine which processes should be further analyzed or optimized
- analyzing heap memory usage — you can see which areas of a program use the most dynamic (heap) memory, to learn which sections of code need to be optimized to improve performance

- finding memory corruption — you can find and fix the bad memory operations that cause a program to crash or behave improperly
- finding memory leaks — you can find and fix memory leaks, to improve an application's long-term stability and performance.

Chapter-4 PERFORMANCE ANALYSIS

4.1 Virtual memory page de(allocation)

This is basically the first phase of the development of linux memory manager. Here the linux memory manager interacts with the kernel to allocate and de-allocate the virtual memory pages in accordance with the requirements of the user space application.

The user space application will request say x bytes from our linux memory manager, the linux memory manager will in turn request a vm page from the kernel mmu of fixed size. After receiving the page it will then divide the vm page into memory blocks and allocate the memory required by the user space application.

In the same pattern if the user space application has freed all the memory which is no longer in use of the application (`xfree()`), the vm page is now empty and is being returned to the kernel mmu. These functions are performed with the help of two major system calls, i.e. `Xcalloc()` and `xfree()`.

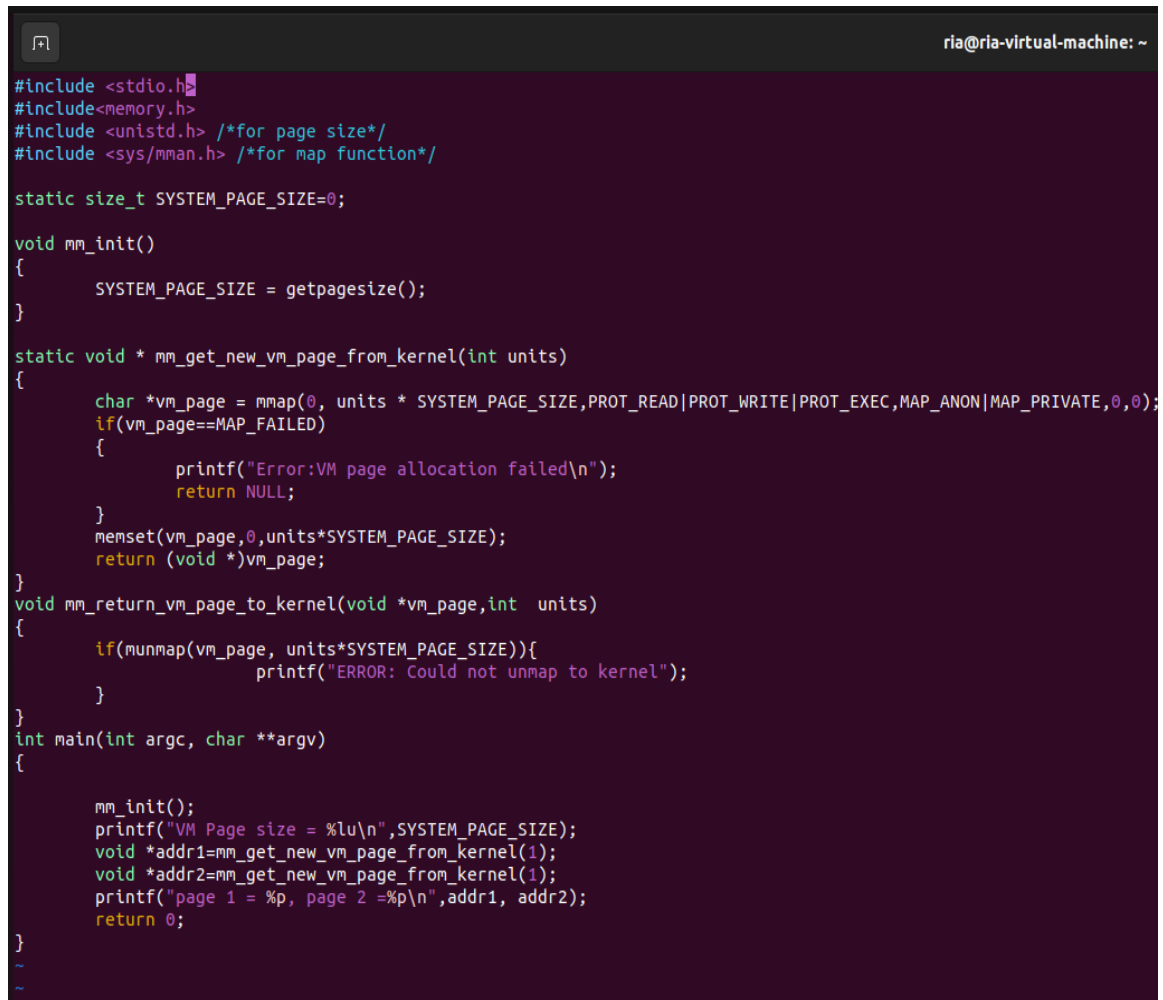
Let us now take a look at the api to implement these system calls.

`System_page_size` defines the size of the vm page extracted from the kernel mmu.

The argument to the function `mm_get_new_vm_page_from_kernel(int units)`, defines the number of pages to be requested from the kernel by the LMM. Similarly, the pages have to be returned to the kernel, in a bottom-up fashion.

In the first phase we have shown the implementation of mmap and munmap system call to allocate and deallocate the vm pages.

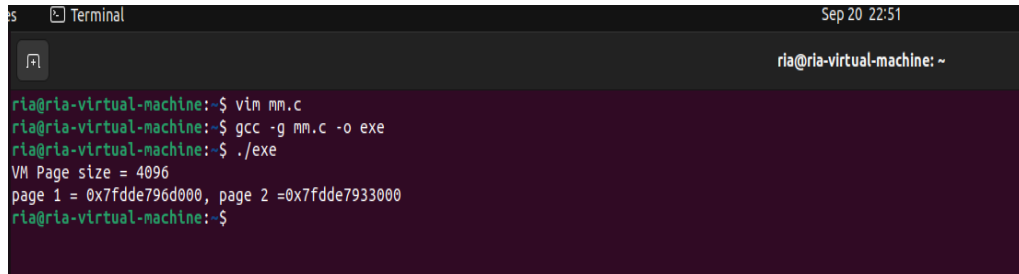
Below is the screenshot attached of the linux terminal (vim) where the c code has been implemented followed by the output of the code written.

A screenshot of a Linux terminal window with a dark purple background. The terminal shows the implementation of a simple memory management library. The code includes headers for stdio, memory, unistd, and sys/mman. It defines a static variable for system page size and implements functions for initialization, getting new VM pages from kernel, and returning them to kernel. The main function demonstrates the usage of these functions. The terminal output shows the VM page size and the addresses of two allocated pages.

```
ria@ria-virtual-machine: ~  
#include <stdio.h>  
#include <memory.h>  
#include <unistd.h> /*for page size*/  
#include <sys/mman.h> /*for map function*/  
  
static size_t SYSTEM_PAGE_SIZE=0;  
  
void mm_init()  
{  
    SYSTEM_PAGE_SIZE = getpagesize();  
}  
  
static void * mm_get_new_vm_page_from_kernel(int units)  
{  
    char *vm_page = mmap(0, units * SYSTEM_PAGE_SIZE, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_ANON|MAP_PRIVATE, 0, 0);  
    if(vm_page==MAP_FAILED)  
    {  
        printf("Error:VM page allocation failed\n");  
        return NULL;  
    }  
    memset(vm_page,0,units*SYSTEM_PAGE_SIZE);  
    return (void *)vm_page;  
}  
  
void mm_return_vm_page_to_kernel(void *vm_page,int units)  
{  
    if(munmap(vm_page, units*SYSTEM_PAGE_SIZE)){  
        printf("ERROR: Could not unmap to kernel");  
    }  
}  
  
int main(int argc, char **argv)  
{  
  
    mm_init();  
    printf("VM Page size = %lu\n",SYSTEM_PAGE_SIZE);  
    void *addr1=mm_get_new_vm_page_from_kernel(1);  
    void *addr2=mm_get_new_vm_page_from_kernel(1);  
    printf("page 1 = %p, page 2 =%p\n",addr1, addr2);  
    return 0;  
}  
~  
~
```

Fig .4.1 Output

The output of the above code snippet is as follows.



```
ria@ria-virtual-machine:~$ vim mm.c
ria@ria-virtual-machine:~$ gcc -g mm.c -o exe
ria@ria-virtual-machine:~$ ./exe
VM Page size = 4096
page 1 = 0x7fdde796d000, page 2 =0x7fdde7933000
ria@ria-virtual-machine:~$
```

Fig. 4.1.2 Output of above code

The output generated depicts the size of the vm page requested from the kernel which comes out to be a standard 4096 bytes.

The next line of the code shows the no of vm pages and their respective addresses.

4.2 Page Family Registration

Family registration means that user space application which is relying on our Linux memory manager for memory allocation and deallocation, such an application during its initialization phase is supposed to tell our Linux memory manager the details of the structures which application is using. These are those the structures which application wishes to perform dynamic memory allocation and deallocation.

So, in the diagram below you can see that we have a user space application on the right-hand side and we have Linux memory manager library which is integrated with our application. In the registration process user space application is telling the Linux memory manager the information about the structures which application is using. let us suppose that the application is using the structure person, occupation structure and the student structure and against each of these a structure is the size of the structure So, the application has to tell the name of the structure and the

size of the structure in the registration process to the Linux memory manager. Application may be using hundreds of a structure.

This registration of structure information with Linux memory manager is essential because Linux memory manager needs to know how many bytes to be allocated to the application, if application issues request for memory allocation for a structure.

It is for this reason that user space application has to convey the structures and their corresponding sizes to the Linux memory manager at the time of initialization of the application. Information comprises of name of the structure and the size of the structure.

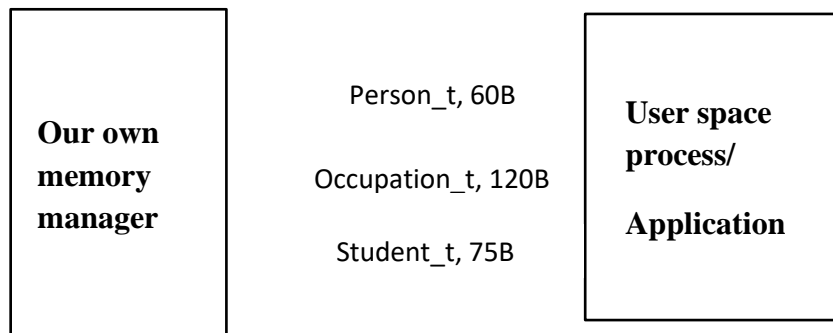


Fig 4.2 Example of user application

Our Linux memory manager needs to have a memory in order to store this structure information, that memory will come from our Linux memory manager itself that has the responsibility to allocate and deallocate memory for the user space application. But what will our Linux memory manager do if it has to use some memory in order to store this structured information. The entity which is responsible to provide the memory to the application itself needs memory to operate.

Linux memory manager uses vm pages as requested from the kernel to store the registration information. We have already done in the phase one

that how our Linux memory manager can request vm pages from the kernel those virtual memory pages will be used by our Linux memory manager in order to store applications structure information. So, this process is called Page family registration. Structure name plus the size of the structure together is called Page family.

4.2.1 Page Family instantiation

It basically means that how our user space application will interact with our LMM. This will be done with help of an api *mm_instantiate_vm_page()*. Where it will accept two arguments i.e., the name of the structure and the size of the structure.

Whenever the application wants to access the memory from the LMM, it will invoke this api and the LMM will accept the above-mentioned arguments for the page family registrations.

All the data structures, looping macros and also the api's have been defined in the mm.h (header file) for the internal usage of the LMM. Apart from this we have created another file uapi_mm.h which will act as an interface header file between the LMM and the user application. To make use of our LMM the application has to include the uapi_mm.h in its particular code.

```

typedef enum{
    MM_FALSE,
    MM_TRUE
} vm_bool_t;

typedef struct block_meta_data_{
    vm_bool_t is_free;
    uint32_t block_size;
    uint32_t offset; /*offset from the start of the page*/
    pthread_t priority_thread_glue;
    struct block_meta_data_ *prev_block;
    struct block_meta_data_ *next_block;
} block_meta_data_t;
GLTHREAD_TO_STRUCT(pthread_to_block_meta_data,
    block_meta_data_t, priority_thread_glue, pthread_ptr);

```

Fig.4.2.1 Code snippet phase 2

```

#define offset_of(container_structure, field_name) \
    ((size_t)&(((container_structure *)0)->field_name))

/*Forward Declaration*/
struct vm_page_family_;

typedef struct vm_page_{
    struct vm_page_ *next;
    struct vm_page_ *prev;
    struct vm_page_family_ *pg_family; /*back pointer*/
    uint32_t page_index;
    block_meta_data_t block_meta_data;
    char page_memory[0];
} vm_page_t;

#define MM_GET_PAGE_FROM_META_BLOCK(block_meta_data_ptr) \
    ((vm_page_t *)((char *)block_meta_data_ptr - block_meta_data_ptr->offset))

#define NEXT_META_BLOCK(block_meta_data_ptr) \
    (block_meta_data_ptr->next_block)

```

Fig 4.2.2 Code snippet phase 2

```

/*
 * Public APIs Exposed to the Application using Memory Manager
 */

/*Printing Functions*/
void mm_print_memory_usage(char *struct_name);
void mm_print_block_usage();

/*Initialization Functions*/
void
mm_init();

/*Registration function*/
#define MM_REG_STRUCT(struct_name) \
    (mm_instantiate_new_page_family(#struct_name, sizeof(struct_name)))

/*Allocators and De-Allocators*/
#define XALLOC(units, struct_name) \
    (xalloc(#struct_name, units))

#define XFREE(ptr) \
    xfree(ptr)

#endif /* __UAPI_MM__ */
~

```

Fig. 4.2.3 Code snippet phase 2

```

ria@ria-virtua
#include <stdint.h>

void *
xalloc(char *struct_name, int units);

void
xfree(void *app_ptr);

void
mm_instantiate_new_page_family(
    char *struct_name,
    uint32_t struct_size);

```

Fig.4.2.4 Code snippet phase 2

4.3 Meta Block and Data Block

This is the phase 3 of our project where we talk in detail about the meta block and data block present in the vm pages. Basically, a virtual memory page is divided into chunks of memory called *data blocks* which have to be assigned to the user space application. These data blocks are preceded by the *meta blocks* which are nothing but all the information regarding the succeeding data block. It consists of a Boolean for *Is_free*, which tells if the data block present is free or not, the next information is regarding the size of the data block and then the pointers to the previous and next meta blocks.

The pointer to next meta block is null if the data block succeeding that meta block is the topmost block of the vm page. Similarly, the pointer the previous meta block is null if the data block succeeding that particular meta block is the bottom most block of the virtual memory page.

Meta Block data structure is as follows:

```
Typedef struct block_meta_data{  
  
Vm_bool_t is_free; //free or allocated  
  
Uint32_t struct_size; //size of the block  
  
Struct block_meta_data_ *prev_block; //ptr to the next meta block  
downward in data VM page  
  
Struct block_meta_data_ *next_block; //ptr to the next meta block upward  
in data VM page  
  
} block_meta_data_t;
```

```

ria@ria-virtual-machine: ~
/*Forward Declaration*/
struct vm_page_family_;

typedef struct vm_page_{
    struct vm_page_ *next;
    struct vm_page_ *prev;
    struct vm_page_family_ *pg_family; /*back pointer*/
    uint32_t page_index;
    block_meta_data_t block_meta_data;
    char page_memory[0];
} vm_page_t;

#define MM_GET_PAGE_FROM_META_BLOCK(block_meta_data_ptr) \
    ((vm_page_t *)((char *)block_meta_data_ptr - block_meta_data_ptr->offset))

#define NEXT_META_BLOCK(block_meta_data_ptr) \
    (block_meta_data_ptr->next_block)

#define NEXT_META_BLOCK_BY_SIZE(block_meta_data_ptr) \
    (block_meta_data_t *)((char *)block_meta_data_ptr + 1) \
    + block_meta_data_ptr->block_size)

#define PREV_META_BLOCK(block_meta_data_ptr) \
    (block_meta_data_ptr->prev_block)

#define mm_bind_blocks_for_allocation(allocated_meta_block, free_meta_block) \
    free_meta_block->prev_block = allocated_meta_block; \
    free_meta_block->next_block = allocated_meta_block->next_block; \
    allocated_meta_block->next_block = free_meta_block; \
    if (free_meta_block->next_block) \
    free_meta_block->next_block->prev_block = free_meta_block

#define mm_bind_blocks_for_deallocation(freed_meta_block_top, freed_meta_block_down) \
    freed_meta_block_top->next_block = freed_meta_block_down->next_block; \
    if (freed_meta_block_down->next_block) \
    freed_meta_block_down->next_block->prev_block = freed_meta_block_top

vm_bool_t
mm_is_vm_page_empty(vm_page_t *vm_page);

```

Fig.4.3 Meta block

```

static inline block_meta_data_t *
mm_get_biggest_free_block_page_family(
    vm_page_family_t *vm_page_family){

    glthread_t *biggest_free_block_glue =
        vm_page_family->free_block_priority_list_head.right;

    if (biggest_free_block_glue)
        return glthread_to_block_meta_data(biggest_free_block_glue);

    return NULL;
}

vm_page_t *
allocate_vm_page();

```

Fig.4.3.1 Meta block

4.4 Block Merging and Splitting

Now as we are entering into the phase 4 of our project, which includes block splitting and block merging. Just as we know that our LMM requests a fresh vm page from the kernel, it is completely free of size 4096 Bytes. Now as the user application asks for memory allocation for say a structure (foo_t) of size 20 Bytes, this particular free data block of 4096 bytes will be split into 2 blocks with their respective meta blocks.

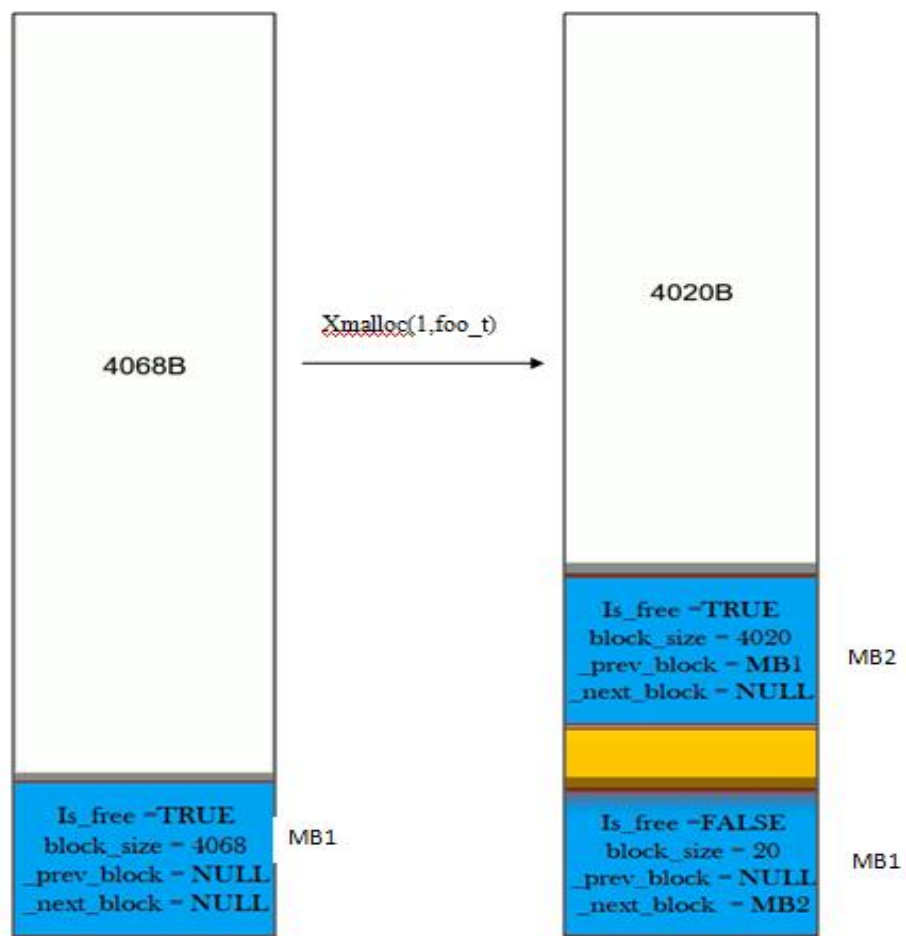


Fig. 4.4 Xcalloc

As it can be observed in the above diagram, the white blocks represent the free data blocks, blocks which have not yet been allocated. On the other hand the yellow blocks represent the data blocks which are not free or the amount of memory which has been allocated to the user space application. The blue blocks are the meta blocks which contain all the necessary information regarding the data blocks. When a free data block experiences a split, a new meta block is also created for that particular data block.

Therefore, it can be concluded that,

- for every malloc, a free block is splitted into allocated block and remaining area left is a smaller free block,
- If vm page does not have free block to satisfy malloc request, a new data vm page is requested from kernel and same exercise is repeated.

4.4.1 Block Merging

Whenever the application wants to release the block of memory, the linux memory manager may require to perform block merging. The diagram on the left hand side shows the snapshot of the virtual memory page.

Let us say that at this point of time the data block MB4 and MB1 is a free data block. All of the data blocks which are in yellow colour are the data blocks which are being used by the application. That is, they are allocated data blocks. Now suppose the application does not need the data block MB5 anymore and therefore the application issues the request to free the memory. Here we have not shown the blue coloured meta blocks but assume that they are always there and also assume that the size of the meta block is negligible. We have done this in order to simplify our calculations and understand the concept of block merging in a simplified way.

Suppose the application does not need the data block MB5 anymore and it wants to free the data block and for that purpose the application issues the request Xfree, passing the pointer to the data block. Finally, our vm page would look like shown in the figure

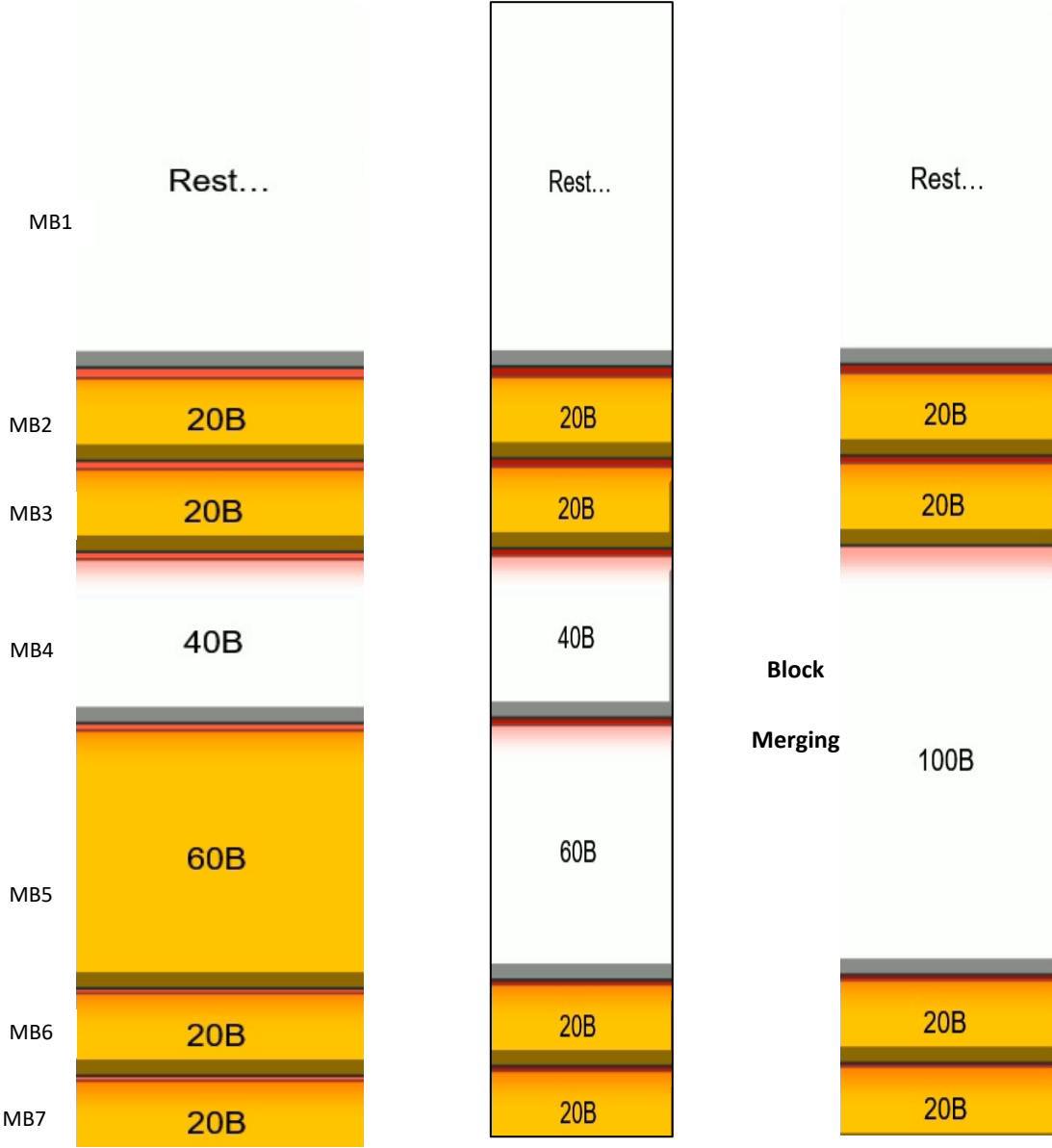


Fig. 4.4.1 Block merging

As you can see, the data block mB5 has changed to white, indicating that it has been released after the data block. The block merging method needs to be run after that because it detects when there are two or more consecutive data blocks in a virtual memory page that need to be combined to make a single, larger free data block.

Therefore, in this instance, the data blocks MB4 and MB5 will be combined to create the data block MB45. Block merging is the process of combining free contiguous data blocks into two; this is prohibited in a virtual memory page. Due to the free data blocks in the centre diagram, this rule is actually more likely to be violated.

Therefore, in order to follow this rule, we must combine these two free data blocks to create MB45, a single larger free data block. In the third vm page, you can observe that no two free data blocks are sequential to one another.

Please find below the code to perform the union of two free data blocks.

```
static void
mm_union_free_blocks(block_meta_data_t *first,
                    block_meta_data_t *second){

    assert(first->is_free == MM_TRUE &&
           second->is_free == MM_TRUE);

    first->block_size += sizeof(block_meta_data_t) +
                        second->block_size;
    remove_glthread(&first->priority_thread_glue);
    remove_glthread(&second->priority_thread_glue);
    mm_bind_blocks_for_deallocation(first, second);
}
```

Fig. 4.4.2 Code snippet block merge

4.5 VM Page Management

Entering into the phase 5 of our project Linux memory Manager i.e. the Virtual page management. This section talks about how the LMM manages and organizes the vm pages which are requested from the kernel. After the LMM accepts the vm page from the kernel it needs to perform certain operations such as,

- Allocation of memory to the application
- Deallocation of the memory from the application
- If a vm page is completely free then return it back to the kernel memory management unit.
- If a vm page is exhausted, the request another vm page from the kernel.
- Maintain collection of vm pages.
- Collect certain statistics. (These statistics will give us idea about the memory usage of an application)

Data structure needed to manage and organize the vm pages in LMM is *vm_page_t* + which is used to represent single data vm page.

We know that our LMM requests the vm pages from the kernel when the vm page already present in the LMM is completely exhausted. Every vm page has to appended at the beginning of the list. All these vm pages and the pointer to the page family registered are connected to each other with the help of a doubly linked list. The pointers next and previous are used to organize virtual memory pages in the form of a doubly linked list. The bottom part of each data virtual memory page will be used to store the objects of type virtual memory page.

Declaration: mm.c

```
vm_bool_t
mm_is_vm_page_empty(vm_page_t *vm_page){

    if(vm_page->block_meta_data.next_block == NULL &&
        vm_page->block_meta_data.prev_block == NULL &&
        vm_page->block_meta_data.is_free == MM_TRUE){

        return MM_TRUE;
    }
    return MM_FALSE;
}
```

Fig.4.5 VM page management

Define: mm.h

```
#define MARK_VM_PAGE_EMPTY(vm_page_t_ptr)
    vm_page_t_ptr->block_meta_data.next_block = NULL;
    vm_page_t_ptr->block_meta_data.prev_block = NULL;
    vm_page_t_ptr->block_meta_data.is_free = MM_TRUE
```

Fig. 4.5 a mm.h file

Macros in mm.h for iterating over virtual memory pages which are present in a doubly linked list,

```

#define N_PAGE_FAMILY_PER_VM_PAGE \
    (GB_SYSTEM_PAGE_SIZE/sizeof(vm_page_family_t))

#define VM_PAGE_FAMILY_RESIDUAL_SPACE \
    (GB_SYSTEM_PAGE_SIZE - (N_PAGE_FAMILY_PER_VM_PAGE * sizeof(vm_page_family_t)))

#define ITERATE_PAGE_FAMILIES_BEGIN(first_vm_page_family_ptr, curr) \
{
    uint32_t count = 0; curr = NULL;
    for(curr = (vm_page_family_t *)first_vm_page_family_ptr;
        count != gb_no_of_vm_families_registered;
        count++, curr++){
        if(count == N_PAGE_FAMILY_PER_VM_PAGE){
            curr = (vm_page_family_t *)((char *)curr +
                VM_PAGE_FAMILY_RESIDUAL_SPACE);
            count = 0;
        }
    }
}

#define ITERATE_PAGE_FAMILIES_END(first_vm_page_family_ptr, curr) \
}}

```

Fig. 4.5 b Looping macros

Macro for iterating over vm page per family,

```

#define ITERATE_VM_PAGE_PER_FAMILY_BEGIN(vm_page_family_ptr, curr) \
{
    curr = vm_page_family_ptr->first_page;
    vm_page_t *next = NULL;
    for(; curr; curr = next){
        next = curr->next;
    }
}

#define ITERATE_VM_PAGE_PER_FAMILY_END(vm_page_family_ptr, curr) \
}}

```

Fig. 4.5 c Iterating macros

```

vm_page_t *
allocate_vm_page();

#define MARK_VM_PAGE_EMPTY(vm_page_t_ptr) \
    vm_page_t_ptr->block_meta_data.next_block = NULL; \
    vm_page_t_ptr->block_meta_data.prev_block = NULL; \
    vm_page_t_ptr->block_meta_data.is_free = MM_TRUE

#define MM_GET_NEXT_PAGE_IN_HEAP_SEGMENT(vm_page_t_ptr, incr) \
    ((incr == '+') ? ((vm_page_t *)((char *)vm_page_t_ptr + GB_SYSTEM_PAGE_SIZE)) : \
    ((vm_page_t *)((char *)vm_page_t_ptr - GB_SYSTEM_PAGE_SIZE)))

```

Fig. 4.5 d Allocations deallocations

4.5.1 VM Page Deletion

Returning the vm page back to the kernel,

```
mm_vm_page_delete_and_free(
    vm_page_t *vm_page){

    vm_page_family_t *vm_page_family =
        vm_page->pg_family;

    assert(vm_page_family->first_page);

    if(vm_page_family->first_page == vm_page){
        vm_page_family->first_page = vm_page->next;
        if(vm_page->next)
            vm_page->next->prev = NULL;
        vm_page_family->no_of_system_calls_to_alloc_dealloc_vm_pages++;
        mm_return_vm_page_to_heap_segment(vm_page);
        return;
    }

    if(vm_page->next)
        vm_page->next->prev = vm_page->prev;
    vm_page->prev->next = vm_page->next;
    vm_page_family->no_of_system_calls_to_alloc_dealloc_vm_pages++;
    mm_return_vm_page_to_heap_segment(vm_page);
}
```

Fig. 4.5.1 VM page insertion

4.5.2 VM Page Insertion

Using the allocate function to request a fresh vm page from the kernel and declaring it in the mm.c file, file m am dot c and i have already defined the prototype of this function in the file mmd attach. So now i have already explained that what this function is supposed to do the very first thing that we need to do is to request a fresh new watch on memory page from the kernel.

So, for that you can simply invoke this api and we are requesting only one world trial memory page from. We are not requesting any giant virtual memory page which is a concatenation of two or more virtual memory

pages. We will discuss more about giant virtual memory pages when we will be modifying our linux manually.

```
vm_page_t *
allocate_vm_page(vm_page_family_t *vm_page_family){

    vm_page_t *prev_page =
        mm_get_available_page_index(vm_page_family);

    vm_page_t *vm_page =
        mm_get_available_page_from_heap_segment();

    vm_page->block_meta_data.is_free = MM_TRUE;
    vm_page->block_meta_data.block_size =
        MAX_PAGE_ALLOCATABLE_MEMORY;
    vm_page->block_meta_data.offset =
        offset_of(vm_page_t, block_meta_data);
    init_glthread(&vm_page->block_meta_data.priority_thread_glue);
    vm_page->block_meta_data.prev_block = NULL;
    vm_page->block_meta_data.next_block = NULL;
    vm_page->next = NULL;
    vm_page->prev = NULL;
    vm_page_family->no_of_system_calls_to_alloc_dealloc_vm_pages++;
    vm_page->pg_family = vm_page_family;

    if(!prev_page){
        vm_page->page_index = 0;
        vm_page->next = vm_page_family->first_page;
        if(vm_page_family->first_page)
            vm_page_family->first_page->prev = vm_page;
        vm_page_family->first_page = vm_page;
        return vm_page;
    }

    vm_page->next = prev_page->next;
    vm_page->prev = prev_page;
    if(vm_page->next)
        vm_page->next->prev = vm_page;
    prev_page->next = vm_page;
    vm_page->page_index = prev_page->page_index + 1;
    return vm_page;
}
```

Fig. 4.5.2 VM page deletion

4.6 Free Data Block Management

Entering into the phase 6 of our project which will deal with the management of a free data block in the virtual memory page. This section will talk about how to choose a free data block for memory allocation to our application. Question arises, which vm page to be chosen for meeting the

xalloc() request from the user space application? The worst fit case would be selecting the page which has the largest free data block available.

The way by which we can find the biggest free data block amongst the various virtual memory pages is by using the data structure, priority queue. Each page family must maintain a max-priority queue of free data blocks. Priority queue of free data blocks the free data blocks will be arranged in this max priority queue in the decreasing order of their block sizes, meaning the largest free data block will be the head of the queue

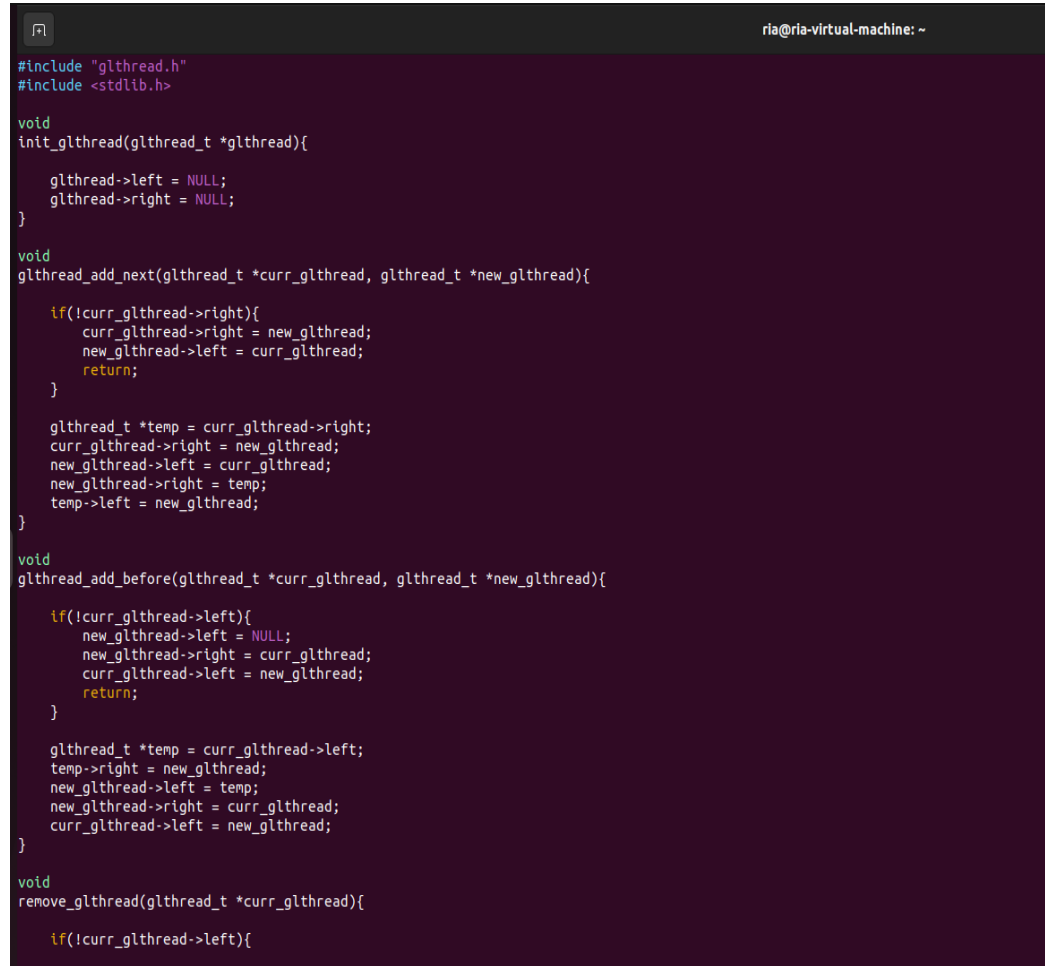
The solution to this question is that that each page family that in this particular data structure must maintain a max priority queue of free data blocks. It means that we will maintain a doubly linked list of free data blocks and these data blocks will be arranged in the decreasing order of their block sizes. Now the Page family data structure have to have an additional member which will point to the head of this priority queue. this additional member is nothing but we will call it as free block priority list head. This is the new member of the data structure Page family which will point to the head of this doubly linked list.

Now let us see with this concept in place how our Linux memory manager will be able to satisfy the malloc request issued by the application. Now having received this malloc request our Linux memory manager will pick up the first free data block which is present in this priority queue and that free data block will be used for memory allocation to the application.

Now a new largest free data block will be created as a result of splitting the older free data block and the other blocks will be rearranged in the priority queue such that the fresh newly created block is inserted in this doubly linked list at an appropriate place.

The time complexity to process the malloc request issued by the application by our Linux memory manager is **order of N** because choosing the largest free data block from the head of this linked list is order of one. But once this free data block is split it up to form a smaller free data block and the insertion of this free data block in this doubly linked list is an order of N operation.

But remember here we are implementing the priority queue using doubly linked list, we implement this prior to queue using max heap data structure. Then we can improve the time complexity of memory allocation to the application by log base 2 which shall be the great improvement.



```
ria@ria-virtual-machine: ~
#include "gthread.h"
#include <stdlib.h>

void
init_gthread(gthread_t *gthread){
    gthread->left = NULL;
    gthread->right = NULL;
}

void
gthread_add_next(gthread_t *curr_gthread, gthread_t *new_gthread){
    if(!curr_gthread->right){
        curr_gthread->right = new_gthread;
        new_gthread->left = curr_gthread;
        return;
    }
    gthread_t *temp = curr_gthread->right;
    curr_gthread->right = new_gthread;
    new_gthread->left = curr_gthread;
    new_gthread->right = temp;
    temp->left = new_gthread;
}

void
gthread_add_before(gthread_t *curr_gthread, gthread_t *new_gthread){
    if(!curr_gthread->left){
        new_gthread->left = NULL;
        new_gthread->right = curr_gthread;
        curr_gthread->left = new_gthread;
        return;
    }
    gthread_t *temp = curr_gthread->left;
    temp->right = new_gthread;
    new_gthread->left = temp;
    new_gthread->right = curr_gthread;
    curr_gthread->left = new_gthread;
}

void
remove_gthread(gthread_t *curr_gthread){
    if(!curr_gthread->left){
```

Fig. 4.6 Block Management

4.7 Final Push-Implement Xmalloc

4.7.1 Xmalloc Algorithm Discussion

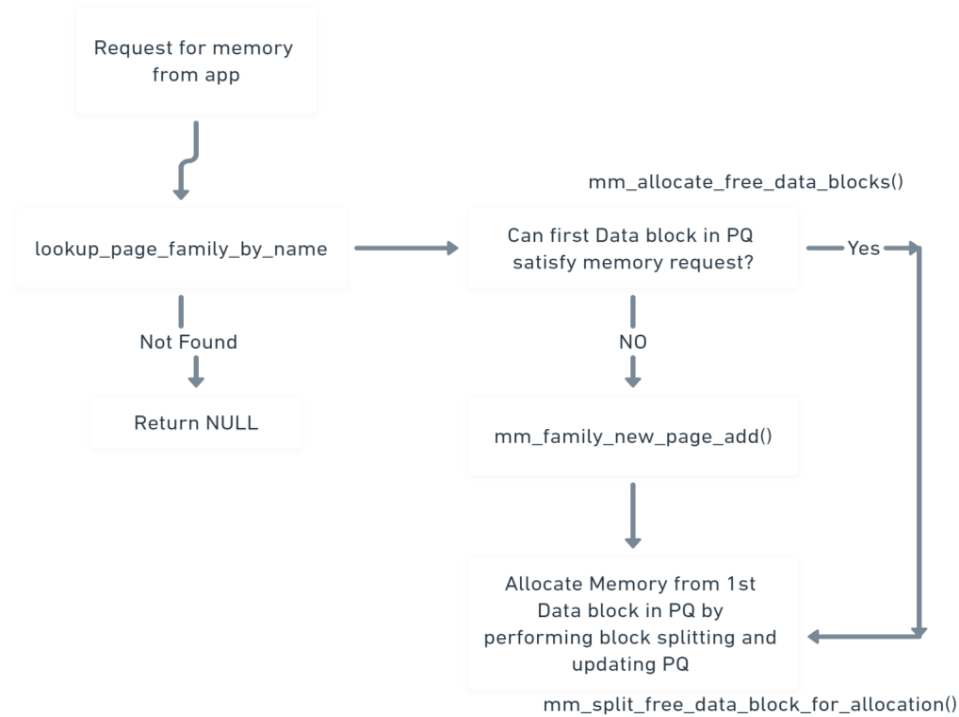


Fig. 4.7 Flow chart

4.7.2 Xmalloc Implementation

The seventh stage of our project is the implementation of X malloc api. The return value of the xmalloc api is a void * pointer because it is up to the application to decide that how it is going to use that chunk of memory which is allocated by our Linux memory manager. So it is for that reason that we return void*. The very first thing that Linux memory manager

must do is to search for Page family corresponding to the structure which is provided by the application.

And if such a page family do not exist then in that case our Linux memory manager should simply print an error and return null.

returning a null will be an indication to the application that memory allocation has failed and let us go forward and assume that such a Page family exist. Now the second thing that all Linux memory managers should check that weather application is demanding from our Linux memory manager a chunk of memory that exceeds the size of vm page.

So if you multiply the units with the structure size that will give you the total amount of memory which application is demanding from the Linux memory manager And if this much memory exceeds the size of one complete virtual memory data page.

Then we will presume that in such scenario our Linux memory manager cannot certify application request from memory. So, in that case simply print an error or null. So, it simply means that Linux memory manager does not have the capability to allocate memory to the application whose size exceeds the size of virtual memory data page. Now let us assume that our Linux memory manager passes put these restrictions and now it is in a position to perform memory allocation. So, what we need to do is to invoke an API M.M. allocate free data block.

```

static vm_bool_t
mm_allocate_free_block(
    block_meta_data_t *block_meta_data,
    uint32_t size){
    assert(block_meta_data->is_free == MM_TRUE);
    assert(block_meta_data->block_size >= size);

    uint32_t remaining_size =
        block_meta_data->block_size - size;

    block_meta_data->is_free = MM_FALSE;
    block_meta_data->block_size = size;

    /*Unchanged*/
    //block_meta_data->offset = ??

    /* Since this block of memory is not allocated, remove it from
     * priority list of free blocks*/
    remove_glthread(&block_meta_data->priority_thread_glue);

    vm_page_family->total_memory_in_use_by_app +=
        sizeof(block_meta_data_t) + size;
    /* No need to do anything else if this block is completely used
     * to satisfy memory request*/
    if(!remaining_size)
        return MM_TRUE;

    /* If the remaining memory chunk in this free block is unusable
     * because of fragmentation - however this should not be possible
     * except the boundary condition*/
    if(remaining_size <
        (sizeof(block_meta_data_t) + vm_page_family->struct_size)){
        /*printf("Warning : %uB Memory Unusable at page bottom\n",
            remaining_size);*/
        return MM_TRUE;
    }
}

```

Fig. 4.7.1 api code

So, remember this is the api which represents the circle in our flowchart.

The logic to search of free data block which is big enough to satisfy a memory request of the application and then perform a splitting of that particular free data block in order to perform memory allocation within this function. So, you can see that this function accepts two argument page family and the total amount of memory to be allocated to the application. The return value of this function is a pointer to the meta block which is a guardian of free data block which will be allocated to the application for use. So once this api returns, we will have a pointer to the betablockade now what we need to do we simply have to initialize the free data block which will be assigned to the application for use. And then simply return the starting address of the free data block to the memory. Meta block is always there, but what we return to the application is the starting address of data block which is to be allocated to the application for use. Right now, at this point of time, this metal block represents a guardian of the data block which has been allocated to the application. So this Meta block is a guardian of occupied data block at this point of time and not a guardian of free data block it means that is free member will be set to false in this meta block. And if for some reason this particular api returns null it means that we have no choice other than to return now to the application which represents that for some x you this is the complete implementation of this Malloc function and you can see it is very simple and short and all the logic of this Xmalloc function lies in this api and therefore we will implement this api next.

4.7.2 Hard Internal Fragmentation

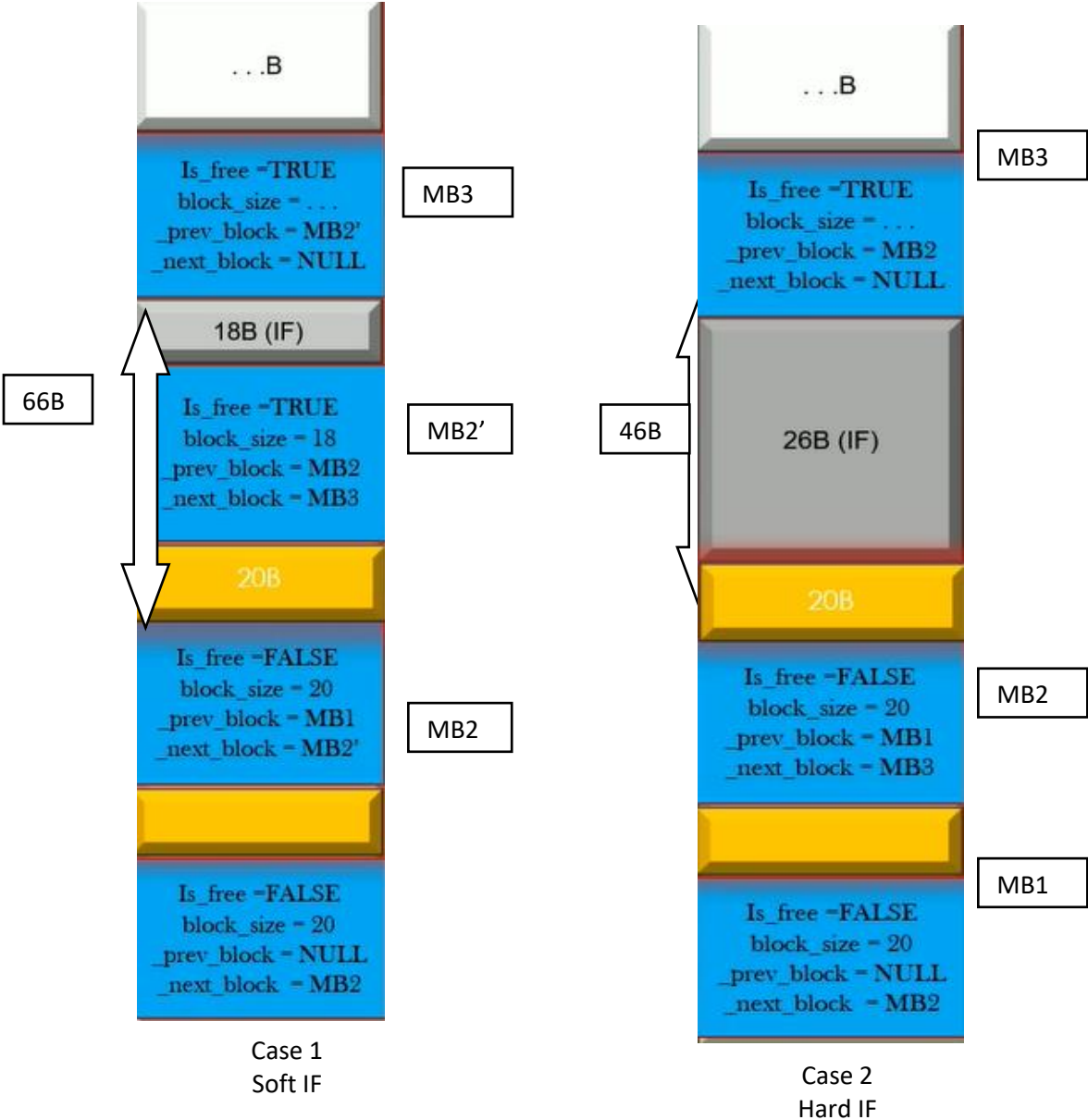


Fig. 4.7.2 Hard and soft If

4.8 Testing Our Project

A lot of functionality has been implemented in our project and here we verify and test the project. Some printing functions are written which shall dump the state of linux memory manager.

4.8.1 Explanation of output API

So, in order to test the functionality of a project remember we wrote a file desktop dot c which represent the application right. So let us open this file and from this file we will try to claim some memory from our Linux memory manager

Remember we defined two structure supply and student. And we have registered these two structures with Linux memory manager and we have already implemented this api to print the registered families. Now going further since we have completed the implementation of our exe malloc therefore, we will clear memory from our Linux memory manager for the object of type employee and student. Now here if you check the implementation of xmalloc it does nothing but it is just a wrapper or what xlog function which we had implemented right. We have done this so that the call to the X function from our application would resemble as closely as possible to the standard malloc function right. So, we have just wrote a macro xmalloc which does nothing but just a wrapper over the malloc api which we wrote so you can see in this example that the application is claiming the memory for free objects of type employee and two objects of type student.

Creating an executable after compiling the program and it can be seen that first that the very first thing the program prints is the base families which are registered with the Linux memory manager. So, this output is coming

from functions which we have implemented long before. And now at this point of time the application is halted at this line. It means that the application has already requested these five objects from our Linux memory manager. And as soon as I press any key then our application will invoke these api which will print the internal state of our Linux memory manager.

Now we will implement to explain why this output is now let me explain the output which is coming from this fast function that is an imprint memory usage. The functionality of the first api that is an imprint memory use such as that it is played or all the its families which are registered with our Linux memory manager.

And for each page family it prints the information about each watchful memory database which is present in the doubly linked list for that its family. Right now, what is that information that is contained in each watch on memory data page for each watch all memory data page.

We also print each matter a block present in a watch on memory data page and we print this information for all the virtual memory data pages present in this doubly linked list. So basically, this function is used to dump the entire content of a page family. So, you can see here that first of all we print the page size that is supported on your system.

And then I tripped over each page families which is registered with our Linux memory manager and for each family then be outraged over all the virtual memory data pages which is present in a doubly linked list of that page family. Now in this case there is only one virtual memory data page which is present in a doubly linked list because only one virtual memory data page is suffice to meet the requirement of an application for memory. So, we are printing the Page family which we optimized from the watch

on memory data page. Remember each watch will memory data page has a bad pointer to the page family. So, from virtual memory database we are printing this information right. And in the application, you can see that for the structure of type imply our application requester three objects from our Linux memory manager. So it is for this reason that we I treat or all the metal blocks in this virtual memory data page and you can see that the block 0 1 and 2 are allocated so we tripped over all the meta blocks which is present in the watch of memory data page.

Starting from the lower most meta block and looping over metal blocks towards the metal block which is present in the higher memory address of a virtual memory data page. This value is the starting address of a metal block and if the metal block is allocated to the application, then we print allocated. Otherwise, we print freed and this is the size of a data block which is being guarded by this metal block. So, you can see that this block size must match with the structure size of a piece family and this represents the offset of a metal block. And since it is a first metal block of a virtual memory data page that is the lower most metal block therefore the previous pointer of this metal block is no. And the next pointer points to the next metal block present in a watch remember memory data page so you can see that this should be the address of block number one. So, in your outward you should take care there D when news matches so block number 0 1 and 2 are allocated blocks and block number three is a metal block which is in the free to state and its block size is 3 7 7 2 bytes. So, if your application request more memory from the Linux memory manager for the page family of type imply your Linux memory manager will going to perform a split of the block number three to meet the memory requirement of the application and because the block number three is the uppermost matter block in a virtual memory data page therefore its next

pointer is no and the similar output is printed for the page family of type student right. So, you must notice all these statistics in your output and see that all these statistics are should be asked for that expectation if you've seen any anomalies in these statistics it means you're The next memory manager implementation is buggy and any unexpected behaviour could we observe and you can also see that we are also printing the total number of virtual memory pages which is being used by our Linux memory manager. So basically, this is the total number of virtual memory data pages which are assigned to the ps families one virtual memory data page has been assigned to the page family imply and one virtual memory database has been assigned to the page family student therefore total number of data virtual memory pages that limits memory manager has requested from the kernel is to. So, this is the output that is coming from the first api.

```

Page Size = 4096 Bytes
vm_page_family : emp_t, struct size = 36
  App Used Memory 408B, #Sys Calls 1
  Page Index : 0 , address = 0x55d00c9d2000
  next = (nil), prev = (nil)
  page family = emp_t
    0x55d00c9d2020 Block 0 ALLOCATED block_size = 36 offset = 32 prev = (nil) next = 0x55d00c9d2074
    0x55d00c9d2074 Block 1 ALLOCATED block_size = 36 offset = 116 prev = 0x55d00c9d2020 next = 0x55d00c9d20c8
    0x55d00c9d20c8 Block 2 ALLOCATED block_size = 36 offset = 200 prev = 0x55d00c9d2074 next = 0x55d00c9d211c
    0x55d00c9d211c Block 3 ALLOCATED block_size = 108 offset = 284 prev = 0x55d00c9d20c8 next = 0x55d00c9d21b8
    0x55d00c9d21b8 Block 4 F R E E D block_size = 3608 offset = 440 prev = 0x55d00c9d211c next = (nil)

vm_page_family : student_t, struct size = 56
  App Used Memory 368B, #Sys Calls 1
  Page Index : 0 , address = 0x55d00c9d3000
  next = (nil), prev = (nil)
  page family = student_t
    0x55d00c9d3020 Block 0 ALLOCATED block_size = 56 offset = 32 prev = (nil) next = 0x55d00c9d3088
    0x55d00c9d3088 Block 1 ALLOCATED block_size = 112 offset = 136 prev = 0x55d00c9d3020 next = 0x55d00c9d3128
    0x55d00c9d3128 Block 2 ALLOCATED block_size = 56 offset = 296 prev = 0x55d00c9d3088 next = 0x55d00c9d3190
    0x55d00c9d3190 Block 3 F R E E D block_size = 3648 offset = 400 prev = 0x55d00c9d3128 next = (nil)

Total Application Memory Usage : 776 Bytes
# Of VM Pages in Use : 2 (8192 Bytes).
Heap Segment Start ptr = 0x55d00c9d1000 , sbrk(0) = 0x55d00c9f5000 , gb_hbsa = 0x55d00c9d2000, diff = 143360
Memory In Use by Application = 9.472656%
Total Memory being used by Memory Manager = 8336 Bytes
emp_t          TBC : 5      FBC : 1      OBC : 4      AppMemUsage : 408
student_t     TBC : 4      FBC : 1      OBC : 3      AppMemUsage : 368
ria@ria-virtual-machine: $

```

Fig 4.8.1 Output of xalloc implementation

4.9 Implementing xfree

4.9.1 xfree algorithm flowchart

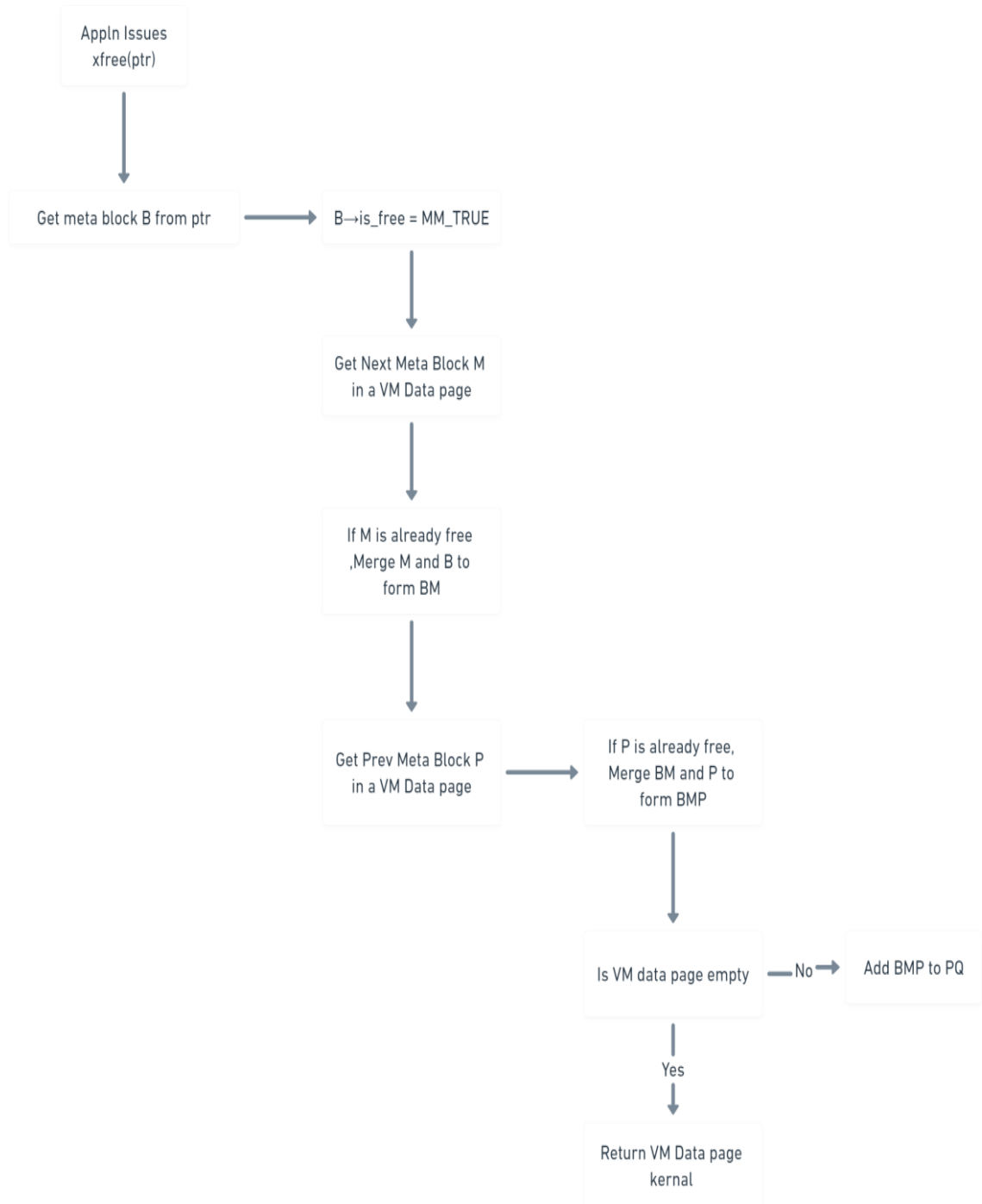


Fig. 4.9 Flow chart

4.9.2 Xfree implementation

Xfree in the file mm.c accepts only one argument. And this argument is a pointer to the data block which was assigned to the application for use. This data block must be data block, which is present somewhere in some virtual memory page. So, the very first thing that we need to do is to obtain a pointer or access to the meta block, which protects this particular data block. So, you can simply opt in a pointer to the protecting metal block by simply decreasing the address, which is pointed by this application data, which is nothing but a data block, and reduce the size of this pointer by the size of the meta block.

You will get the pointer to the matter block, which is protecting this particular data block. And now, henceforth, we will perform all the operations on this meta block. since this particular meta block is a block which is requested by the application to free, therefore is free member of this particular matter, block must be set to. At this point of time, this meta block or data block has been assigned to the application for use. So, it's just a sanity check to ensure that we are doing the right thing. And the next thing is that simply the free api invokes the api M-m free blocks and this api will do rest of the thing. That is, we will implement that entire logic of freeing a particular data block from a virtual memory data page. And doing all the merging algorithm. The scenario one and scenario two that we discussed for handling hard internal fragment and memory will be implemented within this function itself. So now, max, let us discuss the implementation of this function. So, guys, now we will discuss the implementation of this api. This api accepts one argument, which is nothing but a pointer to the matter block, which is to be freed. So, guys, the very first thing that we will going to do is to obtain a pointer to the virtual memory data page in which this particular matter block lies. So,

for that, you can simply use the macro and look at page from the metal block. We have already discussed how we can implement this macro in our previous sections. So, hosting page is a pointer to the virtual memory data page in which this particular free metal block lies. And once we get the pointer to this hosting page, it is not difficult to get a pointer to the page family every which will memory did a page has appointed to the page family.

And therefore, you can easily opt in the pointer to the virtual memory page family. Basically, you can guard the access to the what type of a structure this particular matter block is being used for memory allocation or the allocation. And now roadblock is a local variable, which will hold the address of the matter block, which is to be returned by this api. This api will return the matter block, which will be formed after performing all the marching right. Now, since this is the metal block, which is to be freed, therefore are allotted to must set that is free member to true meaning that this metal block has been marked as free by our Linux memory manager. Now we will be going to covered the snapdeal one and scenario two. And therefore, it's a time for us to test whether this particular matter block is the last metal block towards higher address of the virtual memory page boundary or not. So for that, we will going to obtain the pointer off immediate next metal block, which is present in a virtual memory data page towards higher address in a virtual memory data page. You can simply opt in this pointer using the macro next metal block on our to be free to block.

If this next block is not known, it means that a block which is to be freed, is not the last metal block in a virtual memory data page. In the upper space boundary. And therefore, we enter into the scenario number one. So, if this is a scenario number one, then as discussed, we will going to

add just the block size of this matter block. We have already discussed the scenario.

We just have to invoke this api to add just the block size of the metal block, which is being freed. The hard internal, fragmented memory will be taken into account by using this api. So here you can see that I'm doing the same thing. I am simply in walking this api and any hard internal fragmented memory with the sandwiched between Automator block, which is to be freed and the next batter block will be accommodated in this block size. So, this covers the scenario number one.

And going further now, it's a time to cover the scenario number two. So, if next block pointer is null, then we will enter into the 1st case, meaning that our meta block is the last metal block in the upper space boundary of a virtual memory data page. So, it simply means that we need to now take care of the scenario number two. The block, which is to be freed in this example, was Amber three. So I have written a fairly good comment to explain the scenario number two.

And remember, in order to cover the scenario number two, the very first thing that we need to do is to find the size of hard internal fragment and memory, which is present in the appropriate boundary of a virtual memory data page. And that can be opt in if you just subtract the end address of this watch memory data page from the end address of the data block, which is being freed. So now let's see how we can compute them. So first of all, let us obtain the very end address of a virtual memory data page. We can simply opt ended by adding the system page size to the starting address of the virtual memory data page.

So, hosting page is actually the address of the start of the virtual memory data page. And we are adding the system page size to this value to obtain

the end address of the words of memory data page. So, this is what we are doing here. And the second thing that we need to do is to obtain the end address of the free data block, meaning that we are trying to get the end address of the data block, which is being freed. So you can simply opt ended by using this equation, the metal block, which is to be freed, simply incremented by one and add the block size to this equation to get the end address of the free data block.

So now that we have opt in board the values now, we can simply subtract these two addresses to get the size of hard, internal, fragmented mammary, which is present, which is present at the uppermost region of habitual mammary decoupage. Remember, you should always typecast the address to unsigned long while performing the automatic addition or subtraction. Now that we have now opt in the size of internal, fragmented memory, we can now accommodate this internal fragmented memory to the block size which is being freed. And discovers the scenario number two, right?

So you can see that the algorithm of X free walks in three simple steps where most of the logic is implemented in the step number two and three, the step number two and three will be implemented inside this function. And the argument to this function is a pointer to the matter block, which is to be freed, though, in our project. We do not use the return value of this API, but still we choose to return the pointer to the matter block, which has been marked free by Linux memory manager, and all the necessary merging has already been performed. So the goal of this section of the course is to implement this particular API. That is M m three blocks. The overall goal of this API is to free the memory and if required, return the free or empty virtual memory data page.


```

SCENARIO 2 *****

Page Size = 4096 Bytes
vm_page_family : emp_t, struct size = 36
  App Used Memory 240B, #Sys Calls 1
  Page Index : 0 , address = 0x55ac0bdce000
  next = (nil), prev = (nil)
  page family = emp_t
    0x55ac0bdce020 Block 0 F R E E D block_size = 36   offset = 32   prev = (nil)   next = 0x55ac0bdce074
    0x55ac0bdce074 Block 1 ALLOCATED block_size = 36   offset = 116  prev = 0x55ac0bdce020 next = 0x55ac0bdce0c8
    0x55ac0bdce0c8 Block 2 F R E E D block_size = 36   offset = 200  prev = 0x55ac0bdce074 next = 0x55ac0bdce11c
    0x55ac0bdce11c Block 3 ALLOCATED block_size = 108  offset = 284  prev = 0x55ac0bdce0c8 next = 0x55ac0bdce1b8
    0x55ac0bdce1b8 Block 4 F R E E D block_size = 3608  offset = 440  prev = 0x55ac0bdce11c next = (nil)

vm_page_family : student_t, struct size = 56
  App Used Memory 208B, #Sys Calls 1
  Page Index : 0 , address = 0x55ac0bdcf000
  next = (nil), prev = (nil)
  page family = student_t
    0x55ac0bdcf020 Block 0 ALLOCATED block_size = 56   offset = 32   prev = (nil)   next = 0x55ac0bdcf088
    0x55ac0bdcf088 Block 1 F R E E D block_size = 112  offset = 136  prev = 0x55ac0bdcf020 next = 0x55ac0bdcf128
    0x55ac0bdcf128 Block 2 ALLOCATED block_size = 56   offset = 296  prev = 0x55ac0bdcf088 next = 0x55ac0bdcf190
    0x55ac0bdcf190 Block 3 F R E E D block_size = 3648  offset = 400  prev = 0x55ac0bdcf128 next = (nil)

Total Application Memory Usage : 448 Bytes
# Of VM Pages in Use : 2 (8192 Bytes).
Heap Segment Start ptr = 0x55ac0bdcd000, sbrk(0) = 0x55ac0bdf1000 , gb_hsba = 0x55ac0bdce000, diff = 143360
Memory In Use by Application = 5.468750%
Total Memory being used by Memory Manager = 8336 Bytes
emp_t      TBC : 5   FBC : 3   OBC : 2   AppMemUsage : 240
student_t  TBC : 4   FBC : 2   OBC : 2   AppMemUsage : 208

```

Fig 4.9.2 a Code xfree

So, at this point of time, we have covered the scenario number one as well as the scenario number two, in order to accommodate any hard, internally fragmented.

```

printf("\nSCENARIO 1 *****\n");
mm_print_memory_usage(0);
mm_print_block_usage();

scanf("%d",&wait);

xfree(emp1);
xfree(emp3);
xfree(stu2);

printf("\nSCENARIO 2 *****\n");
mm_print_memory_usage(0);
mm_print_block_usage();

xfree(emp2);
xfree(stu1);

printf("\nSCENARIO 3 *****\n");
mm_print_memory_usage(0);
mm_print_block_usage();

```

Fig 4.9.2 b Output xfree

```

SCENARIO 3 *****

Page Size = 4096 Bytes
vm_page_family : emp_t, struct size = 36
  App Used Memory 156B, #Sys Calls 1
  Page Index : 0 , address = 0x55ac0bdce000
  next = (nil), prev = (nil)
  page family = emp_t
    0x55ac0bdce020 Block 0  F R E E D  block_size = 204   offset = 32   prev = (nil)       next = 0x55ac0bdce11c
    0x55ac0bdce11c Block 1  ALLOCATED  block_size = 108  offset = 284  prev = 0x55ac0bdce020  next = 0x55ac0bdce1b8
    0x55ac0bdce1b8 Block 2  F R E E D  block_size = 3608  offset = 440  prev = 0x55ac0bdce11c  next = (nil)

vm_page_family : student_t, struct size = 56
  App Used Memory 104B, #Sys Calls 1
  Page Index : 0 , address = 0x55ac0bdcf000
  next = (nil), prev = (nil)
  page family = student_t
    0x55ac0bdcf020 Block 0  F R E E D  block_size = 216   offset = 32   prev = (nil)       next = 0x55ac0bdcf128
    0x55ac0bdcf128 Block 1  ALLOCATED  block_size = 56    offset = 296  prev = 0x55ac0bdcf020  next = 0x55ac0bdcf190
    0x55ac0bdcf190 Block 2  F R E E D  block_size = 3648  offset = 400  prev = 0x55ac0bdcf128  next = (nil)

Total Application Memory Usage : 260 Bytes
# Of VM Pages in Use : 2 (8192 Bytes).
Heap Segment Start ptr = 0x55ac0bdc000, sbrk(0) = 0x55ac0bdf1000 , gb_hrsa = 0x55ac0bdce000, diff = 143360
Memory In Use by Application = 3.173828%
Total Memory being used by Memory Manager = 8336 Bytes
emp_t      TBC : 3      FBC : 2      OBC : 1      AppMemUsage : 156
student_t  TBC : 3      FBC : 2      OBC : 1      AppMemUsage : 104

```

Fig 4.9.2 c Output xfree

Chapter-5 CONCLUSIONS

5.1 Conclusions

In virtual memory pages, linux memory manager purchases and releases memory from the kernel for the benefit of the application. For this, the system calls `mmap()` and `munmap()` were employed. Cache vm pages and utilize them as a storage space for application requests for additional memory up until all of the additional memory pages have been used. when the programmer releases enough memory that the vm side has not been used by the application or allocated for use by the application, the vm side kernel.

As its name suggests, the Linux memory management subsystem is in charge of overseeing the system's memory. This covers the implementation of demand paging and virtual memory, memory allocation for both kernel internal structures and user space programs, mapping of files into processes address space, and many other awesome things.

The aim of this project is to assist you create your own heap memory manager to help you manage the memory needs of your processes. The memory manager improves process speed by minimizing or eliminating internal and external fragmentation problems and preventing needless system calls from allocating/deallocating memory. Users may also view statistics on how much memory each process structure uses via the linux memory manager. These statistics can be used to analyses the memory requirements of your application and provide recommendations for reducing those requirements. Her linux memory manager can detect memory leaks as well.

The allocation and deallocation of virtual memory pages with fixed sizes is handled by the Linux memory manager in continual communication with the system's kernel. Instead of dealing with little data blocks or independent data blocks, the linux memory manager asks the kernel for entire virtual memory pages, which are subsequently divided into memory chunks based on the needs of the application using our linux memory manager.

As long as the `xalloc()` method is called, the programmer calls the linux memory manager's `xalloc()` function, which causes the linux memory manager to register the size and name of the application's structures as page families. Because of this, the virtual memory pages continue registering the page families in a contiguous way from bottom to top in order to give the linux memory manager information it needs to allocate memory to the application. With the use of linked lists and a pointer to them, these page families are accessible within virtual memory pages.

This concludes that we are successful able to buil a linux memory manager which is able to allocate and the free the memory using the functions `xmalloc()` and `xfree()` will be used for allocation and freeing, respectively, in this. The application will call our linux memory manager with an `xmalloc()` request, and the linux memory manager will allocate the needed memory in the form of blocks (meta and data blocks).Catch memory leaks ,if there are any memory leaks in the system, can be found using our linux memory manager. Memory leaks typically happen when a user creates virtual memory pages using the linux memory manger and then forgets to pass those pages back to the kernel. The similar thing can happen if the memory that our linux memory manager allotted to the application is not released or given back to it. And address memory fragmentation problems both hard internal fragmentation (not enough space for meta block) and soft

internal fragmentation will also be addressed by linux memory manger (not enough space for data block).

5.2 FUTURE SCOPE

Our heap memory manager can handle a variety of tasks, such malloc and free memory, find memory leaks, and deal with the issue of memory fragmentation. But there are some future enhancements that can be done to make this memory manager efficient.

There is no such method of displaying the stats obtained by the Linux memory manager in a way such that it is easy for the user to understand it and work upon it. The best method of representing data or stats in such a way that it is very useful for the user and he can understand it and work upon it is pictorial representation i.e., to represent the data in a pictorial way most commonly used are graphs. Therefore, a GUI (Graphical User Interface) can be made which can get the stats of the usage of memory from the linux memory manager and display it in the form of graphs so that it is easy for the user to understand it and work upon it.

Most of the memory manager have a very big computational and execution time so there is a very big delay in the processes carried out by the memory manager. Therefore, there can be made a very efficient memory manager which has less computational time and less execution time. This will be very beneficial for the user and the ram.

The content monitoring can be added to the memory manager so that its performance can be easily studied and managed by the user as by monitoring the content on regular intervals can help the user to understand the working of the memory manager.

REFERENCES

- [1] S.Poornima, Chakunta Venkata Guru Rao, Nazia Thabassum, Dr.S.P. Anandaraj, “Memory Management by Using the Heap and the Stack in Java,” December 2014
- [2] Xutong Ma, Jiwei Yan, Wei Wang, Jun Yan, Jian Zhang, Zongyan Qiu, “Detecting Memory-Related Bugs by Tracking Heap Memory Management of C++ Smart Pointers,” November 2021
- [3] Yu Ding, Tao Wei, TaiLei Wang, Zehnkai, Wei Zou, “Heap Taichi : exploiting memory allocating granularity in heap spraying attacks,"June 2010
- [4] C.Cantalupo, Vishwanath Venkatesan, Jeff R. Hammond, Krzysztof Czuryło, Simon Hammond, “Mmemkind: An Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies," September 2015
- [5] Anthony Savidis, “Development Recipe for a Heap Manager Embedding Advanced Bug Defence," January 2004
- [6] T Printezis, R Jones, “GCspy : an adaptive heap visualization framework,"March 2004