A BETTER MALLOC

A report on package submitted by

ANANYA

Roll no. 18PT04

SHRUTHI ABIRAMI P

Roll no. 18PT34

18XT44 - OPERATING SYSTEMS

April 2020

M.Sc. THEORETICAL COMPUTER SCIENCE

DEPARTMENT OF APPLIED MATHEMATICS AND COMPUTATIONAL SCIENCES

PSG COLLEGE OF TECHNOLOGY

COIMBATORE – 641 004.

TABLE OF CONTENTS

ABSTRACT

This report aims to cover the basics of memory allocation in a computer system and deals in detail the points that need to be considered while building a memory allocator. It also provides a detailed description about the problems that would arise if memory is not utilized efficiently and the solutions to those problems.

1.1 INTRODUCTION

Computer memory is any physical device capable of storing information temporarily, like RAM (random access memory), or permanently, like ROM (read-only memory). Memory is central to the operation of a modern computer system. Memory consists of a large array of bytes,each with its own address. Usually,a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process.The processes on the disk that are waiting to be brought into memory for execution form the **input queue**. The normal single-tasking procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.

The main memory must accommodate both the operating system and the various user processes. Each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To realize the increase in performance,in terms of utilization of the CPU and the speed of the computer's response to its users, we must keep several processes in memory—that is, we must **share memory**.

We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory and also ensure **minimal wastage of memory**.

This project aims at building a memory allocator and deallocator along with a few other functionalities that strive to keep the memory wastage minimal.

1.1.1 OBJECTIVES:

The objectives of this project are

1. To understand the nuances of building a memory allocator.
2. To understand the art of performance tuning for different workloads.
3. To create a shared library.

1. 2 DESCRIPTION

In this project we will be implementing a **memory allocator for the heap of a user - level process**.
Memory allocators have two distinct tasks.

- First, the memory allocator asks the operating system to expand the heap portion of the process's address space by calling either ***sbrk*** or ***mmap***.
- Second, the memory allocator doles out this memory to the calling process. This involves managing a free list of memory and finding a contiguous chunk of memory that is large enough for the user's request; when the user later frees memory, it is added back to this list.

This memory allocator is usually provided as part of a standard library and is not part of the OS. To be clear, the memory allocator operates entirely within the address space of a single process and knows nothing about which physical pages have been allocated to this process or the mapping from logical addresses to physical addresses.

When implementing this basic functionality in the project, we have a few guidelines. First, when requesting memory from the OS, we must use **mmap**. Second, although a real memory allocator requests more memory from the OS whenever it can't satisfy a request from the user, here memory allocator must call mmap only one time (when it is first initialized).

Thus, the functions are similar to those provided by malloc() and free(), but a little more interesting as the user can specify out of any valid memory, what memory should be freed. For comparison, the traditional malloc() and free() are defined as follows.

- **void *malloc(size_t size)**: malloc() allocates **size** bytes and returns a pointer to the allocated memory. The memory is not cleared.
- **void free(void *ptr)**: free() frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc() or realloc(). Otherwise, or if

free(ptr) has already been called before, undefined behaviour occurs. If ptr is NULL, no operation is performed.

The implementations of **Mem_Alloc(int size)** and **Mem_Free(void *ptr)** are identical, except the **ptr** passed to Mem_Free does not have to have been previously returned by Mem_Alloc; instead, ptr can point to any valid range of memory returned by Mem_Alloc. We will refer to this range as an "object". For example, the following code sequence is valid with the allocator, but not with the traditional malloc and free:

```
int *ptr;

// The returned memory object is between ptr and ptr+49
if ((ptr = (int *)Mem_Alloc(50 * sizeof(int))) == NULL) exit(1);

// Could replace 30 with any value from 0 to 49..
Mem_Free(ptr+30);
```

For this project, we will be implementing several different routines as part of a shared library. The routines are as follows:

- **int Mem_Init(int sizeOfRegion):** Mem_Init is called one time by a process using routines. **sizeOfRegion** is the number of bytes that is requested from the OS using **mmap**. This amount is rounded up so that the requested memory is in units of the page size (using **getpagesize()**).
- **void *Mem_Alloc(int size):** Mem_Alloc() is similar to the library function malloc(). Mem_Alloc takes as input the size in bytes of the object to be allocated and returns a pointer to the start of that object. The function returns NULL if there is not enough free space within **sizeOfRegion** allocated by Mem_Init to satisfy this request.

- **int Mem_Free(void *ptr):** Mem_Free frees the memory object that ptr falls within, according to the rules described above. Just like with the standard free(), if ptr is NULL, then no operation is performed. The function returns 0 on success and -1 if ptr does not fall within the currently allocated object (note that this includes the case where the object was already freed with Mem_Free).
- **int Mem_IsValid(void *ptr):** This function returns 1 if **ptr** falls within a currently allocated object and 0 if it does not..
- **int Mem_GetSize(void *ptr):** If **ptr** falls within the range of a currently allocated object, then this function returns the size in bytes of that object; otherwise, the function returns -1.

All these routines are provided in a **shared library**. Placing the routines in a shared library instead of a simple object file makes it easier for other programmers to link with the code. There are further advantages to shared (dynamic) libraries over static libraries. When we link with a static library, the code for the entire library is merged with the object code to create the executable; if we link to many static libraries, the executable will be enormous. However, when we link to a shared library, the library's code is not merged with the program's object code; instead, a small amount of stub code is inserted into the object code and the stub code finds and invokes the library code when we execute the program. Therefore, shared libraries have two advantages: they lead to **smaller executables** and they enable users to use the most recent version of the library at run-time.

1.3 SYSTEM CALLS USED

In order to expand the heap portion of the process's address space, the memory allocator requests the operating system by calling either sbrk or mmap system call.

brk() and **sbrk()** change the location of the program break(Figure 1), which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.

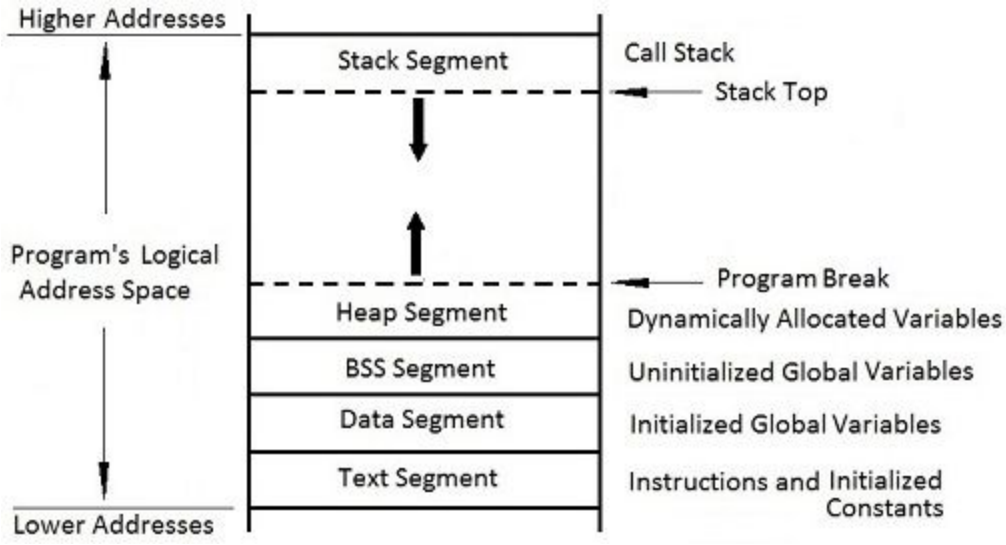

Figure 1 : Illustration of a program break in the heap segment

brk() sets the end of the data segment to the value specified by addr, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size.

sbrk() increments the program's data space by increment bytes. Calling sbrk() with an increment of 0 can be used to find the current location of the program break.

mmap() creates a new mapping in the virtual address space of the calling process.

Since **brk() or sbrk()** only increases or decreases program break, this operation is quite **fast** most of the time. On the other hand, the mmap system call picks up a completely different portion of memory and maps it into the address space of the process, so that the process can see it. Additionally, the mmap call also has to put zeroes in the entire memory area it is about to allocate so that it does not end up leaking information of some old process to this one. This makes mmap quite slow.

Since brk() grows one way and is always contiguous. Take a case where I allocate 10 objects using brk and then free the one that I had allocated first. Despite the fact that the location is now free, it cannot be given back to the OS since it is locked by the other 9 objects. In order to overcome this problem we use mmap().

The basic syntax of mmap is :

#include <sys/mman.h>

void *mmap(**void *** addr, **size_t** length, **int** prot, **int** flags, **int** fd, **off_t** offset);

The starting address for the new mapping is specified in *addr*. If *addr* is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping. If *addr* is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the kernel will pick a nearby page boundary (but always above or equal to the value specified by */proc/sys/vm/mmap_min_addr*) and attempt to create the mapping there. If another mapping already exists there, the kernel picks a new address that may or may not depend on the hint. The address of the new mapping is returned as the result of the call.

The *length* argument specifies the length of the mapping (which must be greater than 0).

The *prot* argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file). It is either **PROT_NONE** or the bitwise OR of one or more of the following flags:

PROT_EXEC Pages may be executed.
PROT_READ Pages may be read.
PROT_WRITE Pages may be written.
PROT_NONE Pages may not be accessed.

The *flags* argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file. This behavior is determined by including exactly one of the following values in *flags*:

MAP_SHARED

Share this mapping. Updates to the mapping are visible to other processes mapping the same region, and (in the case of file-backed mappings) are carried through to the underlying file.

MAP_SHARED_VALIDATE

This flag provides the same behavior as **MAP_SHARED** except that **MAP_SHARED** mappings ignore unknown flags in *flags*. By contrast, when creating a mapping using **MAP_SHARED_VALIDATE**, the kernel verifies all passed flags are known and fails the mapping with the error **EOPNOTSUPP** for unknown flags. This mapping type is also required to be able to use some mapping flags (e.g., **MAP_SYNC**).

MAP_PRIVATE

Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the **mmap**() call are visible in the mapped region.

The contents of a file mapping are initialized using *length* bytes starting at offset *offset* in the file (or other object) referred to by the file descriptor *fd*. *offset* must be a multiple of the page size as returned by getpagesize().

1.4 TOOLS AND TECHNOLOGY USED

The entire program is coded in C language. It is because C provides direct memory control without sacrificing the benefits of high-level languages. We can directly access memory using pointers and perform various operations using them. When memory requirement is not known, we can allocate it dynamically and when memory requirement is known, we can allocate the memory statically. Besides this, C code gets compiled into a raw binary executable which can be directly loaded into memory and executed.

The program uses the following pre - defined header files :

1. **stdio.h** - "stdio.h " is the header file for **St**andar**d I**nput **O**utput. It contains header information for 'File related Input/Output' functions.

   Functions : printf() & scanf()
2. **stdlib.h** - "stdlib.h" is the header file for **St**andar**d Lib**rary. It contains header information for 'Memory Allocation/Freeing' functions.

   Functions : malloc()
3. **unistd.h** - "unistd.h" is the header file that is your code's entry point to various constant, type and function declarations that comprise the POSIX operating system API.

   Functions : getpagesize()
4. **sys/mmap.h** - "mmap.h" header is marked as dependent on support for either the Memory Mapped Files, Process Memory Locking, or Shared Memory Objects options.

1.5 WORKFLOW

1.5.1 MEMORY PROTECTION

Each process has a separate memory space. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a **base** and a **limit**. The base register holds the smallest legal physical memory address; the limit register specifies the size of the range. The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode,and since only the operating system executes in kernel mode,**only the operating system can load the base and limit registers**. This scheme allows the operating system to change the value of the registers but prevent user programs from changing the registers' contents. Since our user program cannot access the base and limit registers, we have introduced two variables namely, base and limit. These variables will carry out similar functionalities as the base and limit register.

1.5.2 MEMORY ALLOCATION

We need to maintain a data structure that tracks the allocated and free portions of the memory. This has been carried out using a linked list whose structure is as follows:

```
struct Node
{
    void* start;    //to point to memory allocated
    int size;
    int status;     //to indicate whether that block is free(0) or allocated(1)
    struct Node* link;
};
```

The start variable is a pointer that points to the starting address to an allocated chunk of memory. The size variable stores the size of that chunk of memory. The status variable takes a

value of 1 if that chunk of memory is allocated and 0 if free. The link stores a pointer to the next Node in the linked list. The file "**linkedList.h**" has the Node definition and also the necessary functions for a linked list.

Functions in LinkedList.h :

`struct Node* New(int sz)` – to create a new Node for the linked list. Sz is the size of that portion of memory. The status variable will be set to 0 initially.

```
struct Node* insert(void *p,int e)
```

The above function is to insert a new node into the list. The variable p points to the starting address of that memory and e is the size of the memory.

```
void traverse(struct Node *head)
```

The above function displays the contents of the linked list.

Functions included in main.c file :

```
int allotted = 0;
int allocated = 0 ;
```

The global variables 'allotted' represents the number of bytes that have been allocated to that process and 'allocated' represents the total number of bytes of memory that have been allocated.

The number of bytes a process requests should be rounded off to a multiple of page size. The reason is a page is allotted exclusively to a single process. Hence, the number of pages that need to be allotted for a process is given by the function `int no_pages(int bytes)`

One of the simplest methods for allocating memory is to divide memory into several **fixed-sized partitions**. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple partition method,when

a partition is free,a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.
In the **variable-partition** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole. Eventually, memory contains a set of holes of various sizes.

A new process that arrives requests memory using **Mem_Init(int sizeOfRegion) .** This function can be called only once by a process. The memory requested , if available, is allocated using mmap() function that returns the starting address of the allotted memory region. Incase of insufficient memory, Mem_Init() cannot be called again unlike malloc(). Instead, the process has to wait until it gets free memory.

int Mem_IsValid(void *ptr) returns 1 if the pointer passed to it lies in the memory space returned by MemInit() and 0 otherwise. This function has been used to check whether a pointer that is passed lies in the address space allotted for that process.

int Mem_GetSize(void *ptr) returns the size of the memory chunk in which the pointer ptr lies, provided ptr is a valid pointer.

void *Mem_Alloc(int size):

The first-fit,best-fit, and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes.
• **First fit** : Allocate the first hole that is big enough.Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended.We can stop searching as soon as we find a freehole that is large enough.
• **Best fit :** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftoverhole.

• **Worst fit :** Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftoverhole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.

Hence, in our implementation of Mem_Alloc(), we have used the **best fit approach.**

If the memory requirement of a request cannot be satisfied at the moment, then the request is enqueued in a sorted linked list.

- **int Mem_Free(void *ptr)**

When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes,these adjacent holes are merged for a larger hole. This is implemented by merging two nodes. If the memory allocated is freed partially, then the node is split into two nodes, an allocated node and a free node.

At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes. If it could be satisfied, that request is processed and removed from the linked list for pending requests.

1.6 RESULTS AND DISCUSSION

This project maintains memory and handles requests using a linked list data structure. We opted linked lists due to the following reasons :

1. Mem_Alloc() and Mem_Free() are frequently occurring operations in this project.
2. Handling memory as an array would provide better time complexity in search operations.
3. But Mem_Alloc() involves inserting a new element and Mem_Free() involves splitting up the element(partitioning the memory).
4. Inserting a new element in an array would increase the time complexity by $O(n^2)$ due to shifting whereas linked lists are capable of doing the same in O(n).

1.6.1 FRAGMENTATION

First-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces.External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes. One solution to the problem of external fragmentation is compaction i.e. to shuffle the memory contents so as to place all free memory together in one large block. The other solutions can be paging and segmentation. Compaction has not been implemented in our project as it is an expensive operation to bring the allocated parts to one side and thereby moving the holes to the other end. Paging and segmentation cannot be employed because only the operating system has the knowledge about all processes and the pages associated with each process.

Hence, a request that cannot be satisfied is made to wait until enough memory is available.

1.6.2 SHARED LIBRARY

A shared library is a file containing object code that several *a.out* files may use simultaneously while executing. When a program is linked with a shared library, the library code that defines the program's external references is not copied into the program's object file. Instead, a special section called **.lib** that identifies the library code is created in the object file. When the UNIX system executes the resulting *a.out* file, it uses the information in this section to bring the required shared library code into the address space of the process.

A shared library offers several benefits by not copying code into *a.out* files. It can:

- Save disk storage space. Shared library code is not copied into all the *a.out* files that use that code. *a.out* files are smaller and use less disk space.
- Save memory. By sharing library code at run time the dynamic memory needs of the processes are reduced.
- Make executable files using library code easier to maintain.

At run time shared library code is brought into the processes' address space. Therefore, updating a shared library effectively updates all executable files that use the library. If an error in shared library code is fixed, all processes automatically use the corrected code.

Hence, the functions specified above can be called by including the shared library.

1.7 CONCLUSION

The result of the program is an efficient memory allocator which is better than the traditional malloc() anf free().

The program uses **mmap()** system call to request memory from the OS which is allocated to the calling process. This memory is accessed by the "objects" within the program by calling the **Mem_Alloc** function. The function is designed in such that it takes care of all the possible memory allocations. It makes sure that properties of continuous memory are maintained as well as memory is used efficiently. For the purpose of allocation the "Best-Fit" method is used as this method provides better performance as compared to Worst-Fit and First-Fit.

In order to free any valid memory, that is memory out of the total memory allocated to the process, the process calls for **Mem_Free** function. This function provides flexibility to the user to pass any valid pointer other than returned by the Mem_Alloc. This makes it better and more flexible as compared to the free() function.

The program consists of the two more functions that are Mem_IsValid and Mem_GetSize for debugging purposes so that the user can check the validity of it's memory allocation. The program is converted in form of a shared library instead of a simple object file makes it easier for other programmers to link with the code.

On the whole the program satisfies all the grounds to behave as a memory allocator.

1.8 BIBLIOGRAPHY

Books

- Silberschatz A, Peter Baer Galvin ,Greg Gagne,"Operating Systems Concepts Essentials", John Wiley, 2016.
- William Stallings, "Operating Systems", Pearson Education, 2015.
- Andrew S Tanenbaum, "Modern Operating Systems", Prentice Hall, 2015.
- McHoes, A M and Flynn, I.M. , "Understanding Operating Systems", Cengage Learning, 2013.

Websites

- http://pages.cs.wisc.edu/~dusseau/Classes/CS537-F07/Projects/P3/index.html