The Better Malloc

Project submitted to the

SRM University – AP, Andhra Pradesh

Submitted in partial fulfillment of the requirement for the award of the degree of

**Bachelor of Technology**

**in**

**Computer Science and Engineering**

**School of Engineering and Sciences**

STUDENT NAME

DEDIPYA GOSWAMI- (AP21110010650)

Under the guidance of

**M. Krishna Siva Prasad**



**Department of Computer Science and Engineering SRM University, AP**

**Neerukonda, Mangalagiri, Guntur**

**Andhra Pradesh – 522 240**

**[May, 2023]**

**Department of Computer Science and Engineering** SRM University, AP



**CERTIFICATE**

This is to certify that the Project report entitled **THE BETTER MALLOC** is being submitted by **DEDIPYA GOSWAMI (AP21110010650),** a student of Department of Computer Science and Engineering, SRM University, AP, in partial fulfillment of the requirement for the degree of **“B.Tech(CSE)”** carried out by her/his during the academic year 2022-2023.

Signature of the Supervisor Signature of Head of the Dept.

**M KRISHNA SIVA PRASAD JATINDRA KUMAR** **DASH**

**Acknowledgement**

The satisfaction that accompanies the successful completion of any task would be incomplete without introducing the people who made it possible and whose constant guidance and encouragement crowns all efforts with success.

I am extremely grateful and express my profound gratitude and indebtedness to my project guide, **M. Krishna Siva Prasad,** Lecturer, Department of Computer Science & Engineering, SRM University, Andhra Pradesh, for his kind help and for giving me the necessary guidance and valuable suggestions in completing this project work.

**Abstract**

This project aims to implement a "better" memory allocator similar to malloc() and free() functions, focusing on understanding the intricacies of building a memory allocator and performance tuning for different workloads. The memory allocator operates within the address space of a user-level process, managing memory allocation and deallocation tasks. The allocator expands the process's heap by requesting memory from the operating system using mmap. The implementation includes managing a free list data structure to keep track of available memory blocks and selecting appropriate chunks of memory to fulfill user requests. The project follows guidelines to use mmap only once for initial initialization. The memory allocator can be fine-tuned by experimenting with different allocation strategies, measuring performance metrics, and optimizing efficiency. Additionally, the implementation can be packaged as a shared library with a public API to allow users to replace default malloc/free calls with the custom allocator. Overall, this project provides a hands-on opportunity to understand the nuances of memory allocation, performance optimization, and creating a shared library.

**Table of contents**

1. Aim
2. Problem statement
3. What is heap ds and why did we include it
4. What are malloc and free.
5. Problems faced
6. Code
7. Outputs
8. Code explanation
9. Conclusion

**AIM**

Implementation of a Memory Allocator in an Operating System: Understanding, Performance Tuning, and Shared Library Creation

**PROBLEM STATEMENT**

"A "Better" Malloc::: There are three objectives to this project:

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.

Overview

In this project, you will be implementing a memory allocator for the heap of a user-level process. Your functions will be similar to those provided by malloc() and free(), but a little more interesting.

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 your project, we have a few guidelines. First, when requesting memory from the OS, you must use mmap (which we think is easier to use than sbrk). Second, although a real memory allocator requests more memory from the OS whenever it can't satisfy a request from the user, your memory allocator must call mmap only one time (when it is first initialized). Third, you are free to use any data structures you want to manage the free list as well as any policy for choosing a chunk of memory."

**Project contribution:**

Coding implementation

**What is heap ds and why did we include it**

Using a heap data structure in this project has several advantages:

Dynamic Memory Allocation: The heap allows for dynamic memory allocation, meaning memory can be allocated and deallocated as needed during runtime. This provides flexibility in managing memory resources and allows for efficient memory utilization.

Variable Memory Size: The heap allows for allocating memory blocks of varying sizes. Unlike static memory allocation, where memory size is fixed at compile time, the heap allows for allocating memory based on runtime requirements. This enables the program to handle different data structures and sizes dynamically.

Memory Reusability: The heap allows for reusing memory blocks that have been deallocated. This helps in optimizing memory usage by reallocating freed memory to fulfill subsequent allocation requests. This feature is particularly useful when memory demands fluctuate during program execution.

Memory Management Control: By implementing a custom memory management system, as done in this code, you have more control over how memory is allocated and deallocated. You can define your own allocation policies, such as first-fit or best-fit, to optimize memory usage based on the specific requirements of your program.

Simplified Memory Management Functions: The use of a heap data structure simplifies the implementation of memory management functions like myMalloc and myFree. These functions provide a higher level of abstraction to the programmer, making it easier to allocate and deallocate memory without having to directly manage low-level memory operations.

Heap Status Monitoring: The code includes a viewHeapStatus function that allows you to visualize the status of the heap, including the used and free memory sections. This provides valuable insights into the memory usage of the program and can help identify potential memory-related issues, such as memory leaks or excessive memory consumption.

It's important to note that while this custom heap implementation provides some advantages, it is a simplified version and may not have the same level of robustness and features as professional memory management systems. In real-world scenarios, it is generally recommended to use the built-in memory management functions provided by the programming language or platform such as malloc and free in C, unless there are specific requirements or constraints that necessitate a custom memory management approach.

**What are malloc and free.**

malloc and free are functions commonly used in programming languages, particularly in C and C++, for dynamic memory allocation and deallocation.

malloc (Memory Allocation):

The malloc function is used to dynamically allocate a specified amount of memory from the heap.

It takes the number of bytes to allocate as its argument and returns a pointer to the allocated memory block.

For example, int\* ptr = (int\*)malloc(sizeof(int)); allocates memory for a single integer and assigns the pointer to the allocated memory to the variable ptr.

free (Memory Deallocation):

The free function is used to deallocate memory that was previously allocated with malloc (or related functions like calloc and realloc).

It takes a pointer to the memory block to be deallocated as its argument.

For example, free(ptr); frees the memory block pointed to by the ptr pointer.

The malloc and free functions are essential for managing dynamic memory in programs, especially when the required memory size is unknown at compile-time or needs to be dynamically adjusted during runtime. They allow programs to allocate memory as needed and release it when it is no longer required, preventing memory leaks and improving memory utilization.

It's important to note that malloc and free are part of the C standard library, and many programming languages provide similar memory allocation and deallocation functions with different names and syntax.

**Problems faced**

Problems faced during our project coding:

We need to use the mmap function which is present in the #include<sys/mman.h> library. This library is not available on windows. Thus we tried Linux.

In Linux we tried installing the gcc and g++ compiler for compiling the code that was in C++. We read about how to do that and we got to know that we needed to use the sudo apt command in the terminal. But when we tried , it said that we donot have the authorization to do so. So we needed to exclude the part where we needed to use either mmap or sbrk.

Thus leaving that point we were only left with designing a malloc and free function of our own that mimics how the real malloc and free request memory and free the requested memory thst was taken respectively.

So we had to modify our project due to the complexities and thus our project is as follows.

**Code**

#include <stdio.h>

#include <stdlib.h>

char\* heap = NULL;

int heapIndex = 0;

int maxHeapSize = 0;

void\* myMalloc(int size) {

if (heapIndex + size <= maxHeapSize) {

void\* ptr = &heap[heapIndex];

heapIndex += size;

return ptr;

} else {

return NULL;

}

}

void myFree(void\* ptr) {

int deallocatedSize = (int)((char\*)ptr - heap);

for (int i = deallocatedSize; i < heapIndex; i++) {

heap[i - deallocatedSize] = heap[i];

}

heapIndex -= deallocatedSize;

}

void viewHeapStatus() {

printf("Heap status:\n");

// Calculate the free size

int freeSize = maxHeapSize - heapIndex;

// Print the header line

printf("+");

for (int i = 0; i < maxHeapSize; i++) {

printf("-");

}

printf("+\n");

// Print the used memory section

printf("|");

for (int i = 0; i < heapIndex; i++) {

printf("X");

}

printf("|\n");

// Print the free memory section

printf("|");

for (int i = 0; i < freeSize; i++) {

printf(" ");

}

printf("|\n");

// Print the footer line

printf("+");

for (int i = 0; i < maxHeapSize; i++) {

printf("-");

}

printf("+\n");

// Print the size information

printf("Total size: %d\n", maxHeapSize);

printf("Used size : %d\n", heapIndex);

printf("Free size : %d\n", freeSize);

}

int main() {

int choice, size;

void\* ptr;

printf("Enter the maximum heap size: ");

scanf("%d", &maxHeapSize);

heap = (char\*)malloc(maxHeapSize);

while (1) {

printf("\nMenu:\n");

printf("1. Allocate memory\n");

printf("2. Deallocate memory\n");

printf("3. View heap status\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the size to allocate: ");

scanf("%d", &size);

ptr = myMalloc(size);

if (ptr == NULL) {

printf("Memory allocation failed. Not enough space in the heap.\n");

} else {

printf("Memory allocated at address: %p\n", ptr);

}

break;

case 2:

printf("Enter the address to deallocate: ");

scanf("%p", &ptr);

myFree(ptr);

printf("Memory deallocated.\n");

break;

case 3:

viewHeapStatus();

break;

case 4:

printf("Exiting program.\n");

free(heap);

exit(0);

default:

printf("Invalid choice. Please try again.\n");

break;

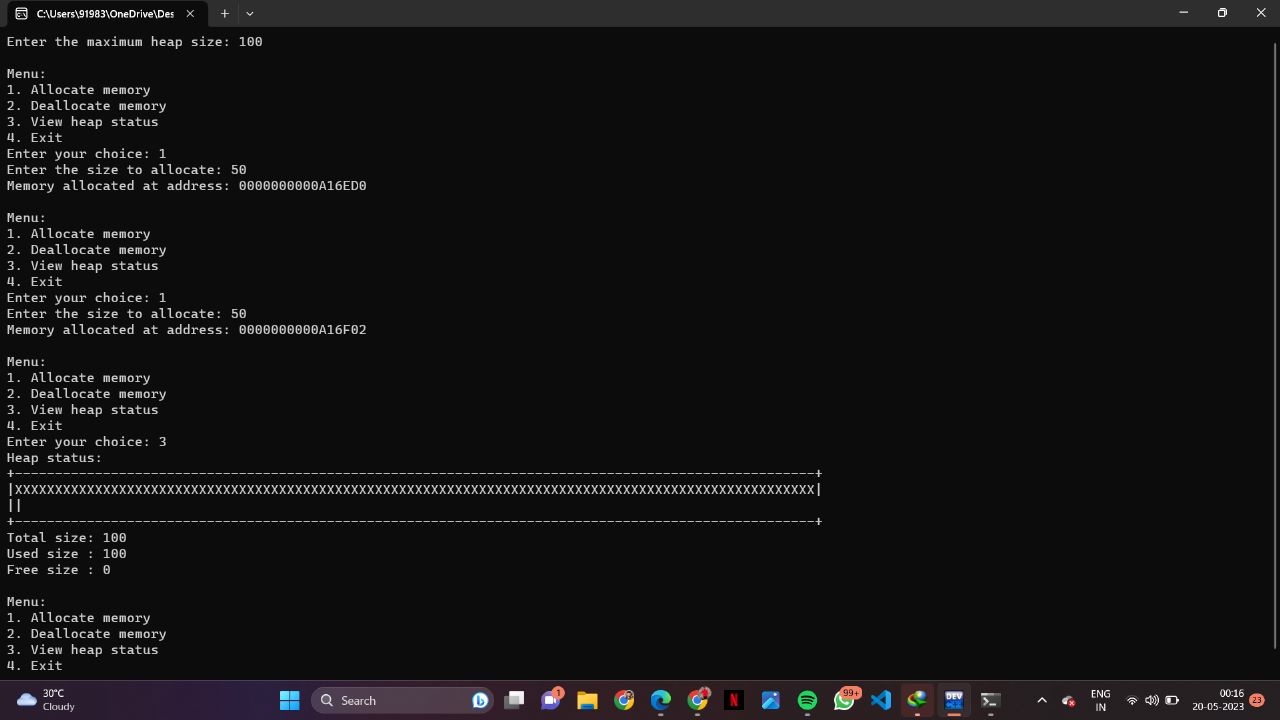
}

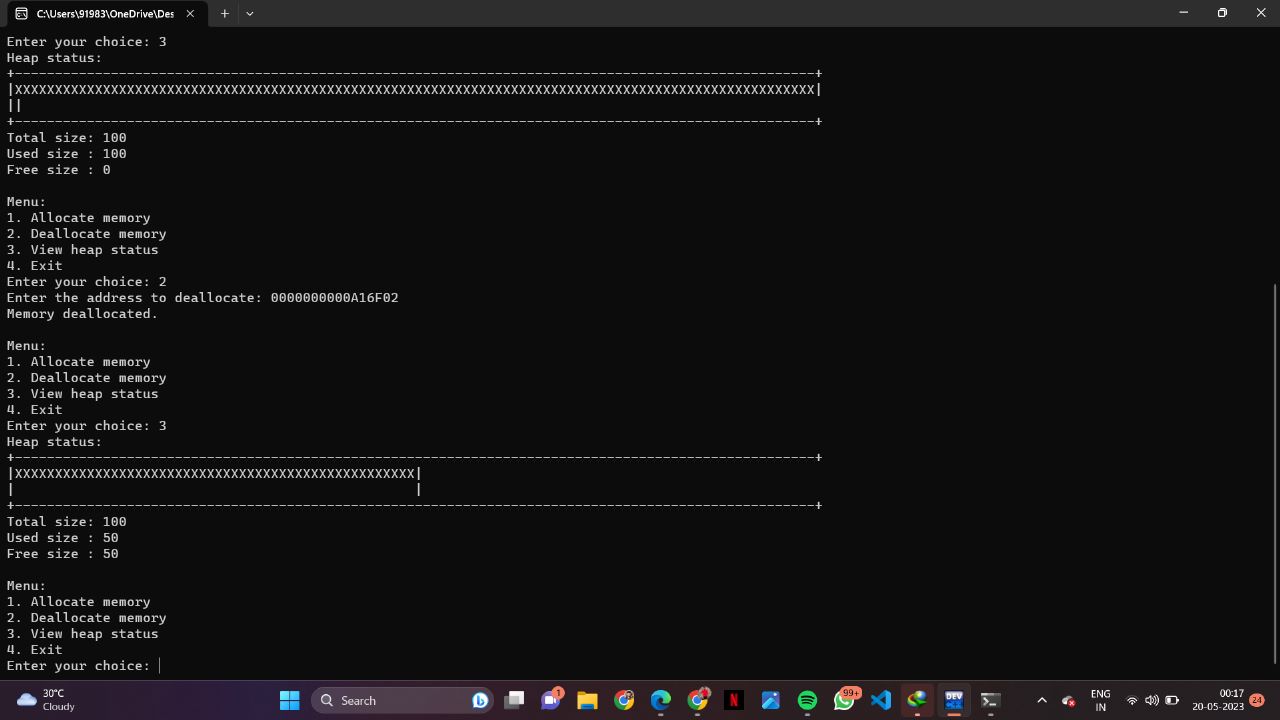
}

return 0;

}

**Outputs**





**Code explanation**

This code implements a basic memory management system using a heap in C. Here's an explanation of each function:

myMalloc(int size):

This function is similar to the standard malloc() function. It takes a size as input and attempts to allocate a block of memory of that size from the heap. If there is enough space in the heap, it returns a pointer to the allocated memory. Otherwise, it returns NULL to indicate failure.

This function takes an integer size as input and attempts to allocate a block of memory of that size from the heap.

It returns a void\* pointer to the allocated memory. Here's how pointers are used within this function:

The variable ptr is declared as a void\* pointer. It will hold the address of the allocated memory.

If there is enough space in the heap (heapIndex + size <= maxHeapSize), the pointer ptr is assigned the address of the current heapIndex location within the heap array.

The heapIndex is then incremented by size to point to the next available memory location.

Finally, the function returns the ptr pointer, which now points to the allocated memory block.

myFree(void\* ptr): This function is similar to the standard free() function. It takes a pointer to a previously allocated block of memory and deallocates it from the heap. It moves the remaining blocks of memory to fill the gap left by the deallocated block.

This function takes a void\* pointer ptr as input, representing a previously allocated memory block. It deallocates the memory block from the heap. –

> > Here's how pointers are used within this function:

The variable deallocatedSize is declared as an integer and represents the size of the memory block to be deallocated.

It is calculated by subtracting the address of ptr from the base address of the heap array ((int)((char\*)ptr - heap)).

This gives the number of bytes between ptr and the start of the heap.

A loop is used to shift the remaining memory blocks in the heap array to fill the gap left by the deallocated block.

The loop iterates from deallocatedSize to heapIndex, moving each byte in the heap array deallocatedSize positions back.

After the loop, the heapIndex is decremented by deallocatedSize, effectively reducing the used memory size.

viewHeapStatus(): This function displays the current status of the heap. It shows the total size of the heap, the amount of used memory (heapIndex), and the amount of free memory.

This function displays the current status of the heap. It doesn't directly use pointers, but it relies on the values stored in the variables maxHeapSize and heapIndex to calculate the free memory size.

>>This is followed by the main function:

The main function utilizes pointers in a few places:

. The ptr variable in the main function is declared as a void\* pointer. It is used to store the return value of myMalloc(), representing the address of the allocated memory block, or to store the user-provided address for deallocation.

. In the case 2 of the switch statement, the user is prompted to enter the address to deallocate, and the ptr pointer is used to store this address.

. In the case 4, when the program is exiting, the heap array is deallocated by calling free(heap). Here, the heap pointer is used to release the memory allocated for the heap.

Throughout the code, pointers are used to keep track of memory addresses, allocate memory blocks, and release memory from the heap. They allow the program to manage the heap and provide dynamic memory allocation functionality.

**Conclusion**

In conclusion, the provided code implements a basic memory management system using a heap in C. It demonstrates how pointers can be utilized to allocate and deallocate memory dynamically. The code allows users to allocate memory blocks of variable sizes from the heap and later deallocate them when they are no longer needed.

The `myMalloc()` function uses a pointer to keep track of the next available memory location in the heap, allowing it to allocate memory blocks of the desired size. On the other hand, the `myFree()` function leverages pointers to identify the memory block to deallocate and shifts the remaining blocks to fill the gap left by the deallocated block.

The `viewHeapStatus()` function provides a snapshot of the current heap status, indicating the total size of the heap, the used memory size, and the free memory size.

By utilizing pointers effectively, the code provides a simplified illustration of memory management and showcases the principles behind dynamic memory allocation and deallocation. However, it's important to note that this code is simplistic and lacks error handling and advanced memory management techniques typically found in professional memory management systems.

***------------ THANK YOU ------------***