MEMORY MANAGEMENT SIMULATOR FOR MODULAR PROGRAMS

25ES601 - Embedded Software Development Essentials

Mini Project

Submitted by

BL.EN.P2EBS25010	LOGESH L
BL.EN.P2EBS25020	JAY PRAKASH MANDAL

Guided by

Ms. Nithya M.

ELECTRICAL AND ELECTRONICS ENGINEERING

AMRITA SCHOOL OF ENGINEERING,

BENGALURU AMRITA VISHWA VIDYAPEETHAM

BENGALURU

560035

November 2025

Problem Statement

Effective memory management is crucial to computer systems in order to guarantee peak performance and resource usage. However, fragmentation—the division of available memory into tiny, useless blocks—occurs frequently as a result of inadequate memory space and poor allocation techniques. Even when there is enough free memory overall, this lowers system efficiency and could result in allocation failures.

In order to solve the issue, this project uses the C programming language to simulate a dynamic memory allocation system using a fixed-size heap. It uses a linked list of memory segments to track allocated and free regions, simulating how an operating system handles processes in memory. The simulation reorganizes memory into continuous free blocks by performing compaction, identifying and reporting internal and external fragmentation, and offering mechanisms to decrease fragmentation.

Objectives

1. **To simulate dynamic memory allocation** within a fixed-size heap using the C programming language.
2. **To understand and demonstrate memory management concepts** such as allocation, deallocation, and fragmentation.
3. **To analyse and differentiate** between **internal** and **external fragmentation** in memory systems.
4. **To implement algorithms** for reducing internal fragmentation and merging free memory blocks.
5. **To develop a compaction technique** that reorganizes memory and minimizes wasted space.
6. **To visualize memory allocation and fragmentation** through a clear console-based representation.
7. **To enhance understanding** of operating system-level memory management techniques in embedded software environments.

Algorithm:

1. Start
2. Initialize Heap:
    - Define a fixed-size simulated heap (e.g., 64 KB).
    - Create a linked list where each node (segment) represents a block of memory.
    - Mark the entire heap as a single free segment initially.
3. Load modules:
    - Read file sizes for each moule(e.g., ADD.h, MUL.h, DIV.h)
    - Align each size to the defined memory boundary (e.g., 8 bytes).

4. Allocate Memory
    - Search the linked list for a free segment large enough to satisfy the request.
    - If found :
      a. Split the segment into two parts — allocated and remaining free memory.
      b. Mark the allocated block with process ID (PID) and requested size.
    - If not found:
      a. Merge adjacent free blocks and retry allocation.
      b. If still not possible, report "Allocation Failed."
5. Free memory
    - When a process completes, search for its PID in the linked list.
    - Mark the corresponding segment as free.
    - Merge adjacent free blocks to reduce external fragmentation.
6. Fragmentation calculation
    - Internal Fragmentation: Difference between allocated block size and actual requested size.
    - External Fragmentation: Total free memory available in non-contiguous blocks.
7. Reduce fragmentation
    - Adjust block sizes to match requested sizes.
    - Convert leftover space into new free segments.
8. Memory compaction
    - Shift allocated blocks to the beginning of the heap.
    - Combine remaining free space at the end as one continuous block.
9. Display result
    - Print current memory map, fragmentation report, and process details.
10. End

Data Structure Implementation:

The project uses a linked list-based data structure to represent the simulated heap. Each node in the linked list corresponds to a memory segment, containing information about its starting address, size, allocation status, and process ownership.

```
typedef struct Segment {
       void *start;             // Starting address of the memory
   segment
       size_t size;             // Total size of the segment
       size_t requested_size;   // Actual size requested by the
   process
       int free;                // Allocation status: 1 = Free, 0
   = Allocated
       int pid;                 // Process ID (PID) using the
   segment
       struct Segment *next;    // Pointer to the next segment in
   the list
} Segment;
```

Explanation

1. **start –** Holds the starting address of the segment within the simulated heap.
2. **size –** The total size allocated for this memory block (may include padding for alignment).
3. **requested_size –** The actual size requested by the process; helps calculate internal fragmentation.
4. **free –** A flag that indicates whether the segment is free (1) or allocated (0).
5. **pid –** A unique process identifier used to track which process owns the segment.
6. **next –** Points to the next segment in the linked list, forming a dynamic memory map.

Purpose of Using a Linked List

The linked list provides flexibility in dynamically adding, splitting, merging, and deleting memory blocks.
This structure enables:

- Easy splitting of segments during allocation.
- Efficient merging of adjacent free blocks.
- Simple traversal to compute fragmentation and display the memory map.

Thus, the Segment structure acts as the core component of the memory management simulation, effectively modeling how an operating system tracks memory usage and fragmentation.

Functions :

1. Initialization Module

Function: `init_heap(size_t size)`

- Allocates the simulated heap of a fixed size (64 KB).
- Creates the first segment that represents the entire heap as a single free block.
- Initializes the linked list for memory management.

2. Memory Allocation Module

Function: `allocate(size_t size, int pid)`

- Searches the linked list for a free segment large enough for the requested memory size.
- Allocates memory by marking the segment as used and assigning a Process ID (PID).

- Calls split_segment() if the segment is larger than required, splitting it into allocated and free parts.

Helper Function: `split_segment(Segment *seg, size_t size, int pid)`

- Splits a free segment into two parts: allocated and remaining free space.
- Updates the linked list accordingly.

3. Memory Deallocation Module

Function: `free_pid(int pid)`

- Frees all memory blocks associated with a given process ID (PID).
- Marks the segment as free and merges adjacent free segments to minimize external fragmentation.

Helper Function: `merge_free()`

- Scans the linked list and merges consecutive free segments into one larger block.
- Reduces external fragmentation and simplifies the memory map.

4. Fragmentation Analysis Module

Function: `fragmentation()`

- Calculates and displays internal and external fragmentation.
    - *Internal Fragmentation:* Difference between allocated block size and requested size.
    - *External Fragmentation:* Total memory in non-contiguous free segments.

Function: `reduce_internal_fragmentation()`

- Reduces internal fragmentation by resizing allocated segments to their exact requested size.
- Converts leftover space into new free blocks and merges them with adjacent free memory.

5. Memory Compaction Module

Function: `compact()`

- Rearranges allocated blocks to eliminate gaps caused by freed memory.
- Moves all allocated segments to the start of the heap and combines remaining free space at the end into a single block.
- Helps in reducing external fragmentation completely.

6. Display and Monitoring Module

Function: `display_memory()`

- Prints the current state of memory allocation to the console.
- Shows each segment's start and end address, size, status (FREE or ALLOC), and associated PID.

7. Supporting Modules (Header Files)

- filesize.h → Contains function to get file size for simulating process memory requests.
- ADD.h, MUL.h, DIV.h → Represent different modules (processes) with unique memory requirements.
- These help test memory allocation behavior for multiple processes.

Code:

Main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "filesize.h"
#include "ADD.h"
#include "MUL.h"
#include "DIV.h"

#define SIM_HEAP_SIZE (64*1024)  // 64 KB simulated heap
#define ALIGNMENT 8

typedef struct Segment {
    void *start;
    size_t size;
    size_t requested_size;
    int free;
    int pid;
    struct Segment *next;
} Segment;
static char *sim_heap = NULL;
```

```
static Segment *head = NULL;
static int next_pid = 1;
size_t align_up(size_t size) {
    return (size + ALIGNMENT - 1) & ~(ALIGNMENT - 1);
}
void init_heap(size_t size) {
    sim_heap = (char*) malloc(size);
    if (!sim_heap) {
        fprintf(stderr, "Failed to allocate simulated heap\n");
        exit(1);
    }
    head = (Segment*) malloc(sizeof(Segment));
    head->start = sim_heap;
    head->size = size;
    head->requested_size = 0;
    head->free = 1;
    head->pid = -1;
    head->next = NULL;
}
void merge_free() {
    Segment *cur = head;
    while (cur && cur->next) {
        if (cur->free && cur->next->free) {
            Segment *tmp = cur->next;
            cur->size += tmp->size;
            cur->next = tmp->next;
            free(tmp);
        } else cur = cur->next;
    }
}
Segment* split_segment(Segment *seg, size_t size, int pid) {
    if (!seg || !seg->free || seg->size < size) return NULL;
```

```
    size_t leftover = seg->size - size;
    seg->free = 0;
    seg->pid = pid;
    seg->requested_size = size;
    seg->size = size;
    if (leftover >= ALIGNMENT) {
        Segment *free_seg = (Segment*) malloc(sizeof(Segment));
        free_seg->start = (char*)seg->start + size;
        free_seg->size = leftover;
        free_seg->requested_size = 0;
        free_seg->free = 1;
        free_seg->pid = -1;
        free_seg->next = seg->next;
        seg->next = free_seg;
    }
    return seg;
}
Segment* allocate(size_t size, int pid) {
    Segment *cur = head, *target = NULL;
    while (cur) {
        if (cur->free && cur->size >= size) { target = cur; break; }
        cur = cur->next;
    }
    if (!target) {
        merge_free();
        cur = head;
        while (cur) {
            if (cur->free && cur->size >= size) { target = cur; break; }
            cur = cur->next;
        }
    }
    if (target) return split_segment(target, size, pid);
    return NULL;
```

```
}
int free_pid(int pid) { //remove the process
    Segment *cur = head; int found = 0;
    while (cur) {
        if (!cur->free && cur->pid == pid) {
            cur->free = 1;
            cur->pid = -1;
            cur->requested_size = 0;
            found = 1;
        }
        cur = cur->next;
    }
    if (found) merge_free();
    return found;
}
void display_memory() {
    printf("\nSegment# | Start -> End | Size | Status\n");
    printf("-----------------------------------------------\n");
    Segment *cur = head; int idx = 0;
    while (cur) {
        printf("%3d | %p -> %p | %5zu | %s", idx,
               cur->start, (char*)cur->start + cur->size - 1,
               cur->size, cur->free ? "FREE" : "ALLOC");
        if (!cur->free) printf(" (P%d)", cur->pid);
        printf("\n");
        idx++; cur = cur->next;
    }
}
void fragmentation() {
    size_t internal = 0, external = 0;
    Segment *cur = head;
    while (cur) {
        if (cur->free) external += cur->size;//add all avaiable memeory as
exernal memeory
```

```
        else if (cur->size > cur->requested_size)//for internal size -
required size
            internal += cur->size - cur->requested_size;
        cur = cur->next;
    }
    printf("\n--- Fragmentation Report ---\nInternal: %zu bytes\nExternal:
%zu bytes\n", internal, external);
}
void reduce_internal_fragmentation() {
    Segment *cur = head;
    size_t reduced_bytes = 0;
    while (cur) {
        if (!cur->free && cur->size > cur->requested_size) {// allocated
and is there any free space in the memory
            size_t diff = cur->size - cur->requested_size;//remaining
space
            reduced_bytes += diff;
            cur->size = cur->requested_size;
            Segment *free_seg = (Segment*) malloc(sizeof(Segment));
//create a new block for the free space
            free_seg->start = (char*)cur->start + cur->size;
            free_seg->size = diff;
            free_seg->requested_size = 0;
            free_seg->free = 1;
            free_seg->pid = -1;
            free_seg->next = cur->next;
            cur->next = free_seg;
        }
        cur = cur->next;
    }
    merge_free();
    printf("\nReduced internal fragmentation by %zu bytes.\n",
reduced_bytes);
}
void compact() {
    Segment *cur = head, *prev = NULL;
```

```
    char *next_addr = sim_heap;
    while (cur) {
        if (!cur->free) { // segment is in used don't do anything just
increment
            cur->start = next_addr;
            next_addr = (char*)cur->start + cur->size;
            prev = cur;
            cur = cur->next;
        } else {// free means remove the gap
            Segment *tmp = cur;
            cur = cur->next;
            if (prev) prev->next = cur;
            else head = cur;
            free(tmp);
        }
    }
    size_t rem = (sim_heap + SIM_HEAP_SIZE) - next_addr;//free memory
    if (rem > 0) {//to say that all other are empty
        Segment *free_seg = (Segment*) malloc(sizeof(Segment));
        free_seg->start = next_addr;
        free_seg->size = rem;
        free_seg->free = 1;
        free_seg->pid = -1;
        free_seg->requested_size = 0;
        free_seg->next = NULL;
        if (!head) head = free_seg;
        else { //till the last step because the free_seg is stored in last
block
            cur = head;
            while (cur->next) cur = cur->next;
            cur->next = free_seg;
        }
    }
}
```

```
int main() {
    printf("Initializing %d bytes simulated heap...\n", SIM_HEAP_SIZE);
    init_heap(SIM_HEAP_SIZE);



    struct { const char *name; const char *filename; } modules[] = {
        {"ADD", "ADD.h"},
        {"MUL", "MUL.h"},
        {"DIV", "DIV.h"}
    };
    for (int i = 0; i < 3; i++) {
        long fsize = getFileSize(modules[i].filename);
        if (fsize <= 0) {
            printf("File not found or invalid size for %s, skipping.\n",
modules[i].name);
            continue;
        }
        int pid = next_pid;
        size_t aligned = align_up(fsize);
        Segment *seg = allocate(aligned, pid);
        if (seg) {
            seg->requested_size = fsize;
            printf("Allocated P%d: %ld bytes (aligned to %zu) for %s
module\n",
                   pid, fsize, aligned, modules[i].name);
            next_pid++;
        } else printf("Allocation failed for %s module\n",
modules[i].name);
    }


    int running = 1;
    while (running) {
        int opt;
```

```
        printf("\n1) Show memory map\n2) Fragmentation report\n3) Reduce
internal fragmentation\n");
        printf("4) Compact memory\n5) Free PID\n6) Exit\nChoose: ");
        if (scanf("%d", &opt) != 1) { while (getchar() != '\n'); continue;
}


        switch (opt) {
            case 1: display_memory(); break;
            case 2: fragmentation(); break;
            case 3: reduce_internal_fragmentation(); break;
            case 4: compact(); printf("Compaction done.\n"); break;
            case 5: {
                int pid;
                printf("Enter PID to free: ");
                scanf("%d", &pid);
                if (free_pid(pid)) printf("Freed P%d\n", pid);
                else printf("PID not found\n");
            } break;
            case 6: running = 0; break;
            default: printf("Invalid option.\n");
        }
    }
    //reset the sim_heap
    Segment *cur = head;
    while (cur) { Segment *t = cur->next; free(cur); cur = t; }
    free(sim_heap);
    printf("Exiting simulator.\n");
    return 0;
}
```

Filesize.c

```
#include <stdio.h>
#include "filesize.h"

long getFileSize(const char *filename) {
```

```
    FILE *fp = fopen(filename, "rb"); // open file in binary mode
    if (!fp) {
        printf("Cannot open file '%s'\n", filename);
        return -1;
    }

    fseek(fp, 0, SEEK_END);          // move to end
    long size = ftell(fp);           // get file size
    fclose(fp);
    return size;
}
```

Filesize.h

```
#ifndef FILESIZE_H_INCLUDED
#define FILESIZE_H_INCLUDED
long getFileSize(const char *filename);
#endif // FILESIZE_H_INCLUDED
```

ADD.h

```
#ifndef ADD_H_INCLUDED
#define ADD_H_INCLUDED
int add(int a, int b);
int doubl=34;
#endif // ADD_H_INCLUDED
```

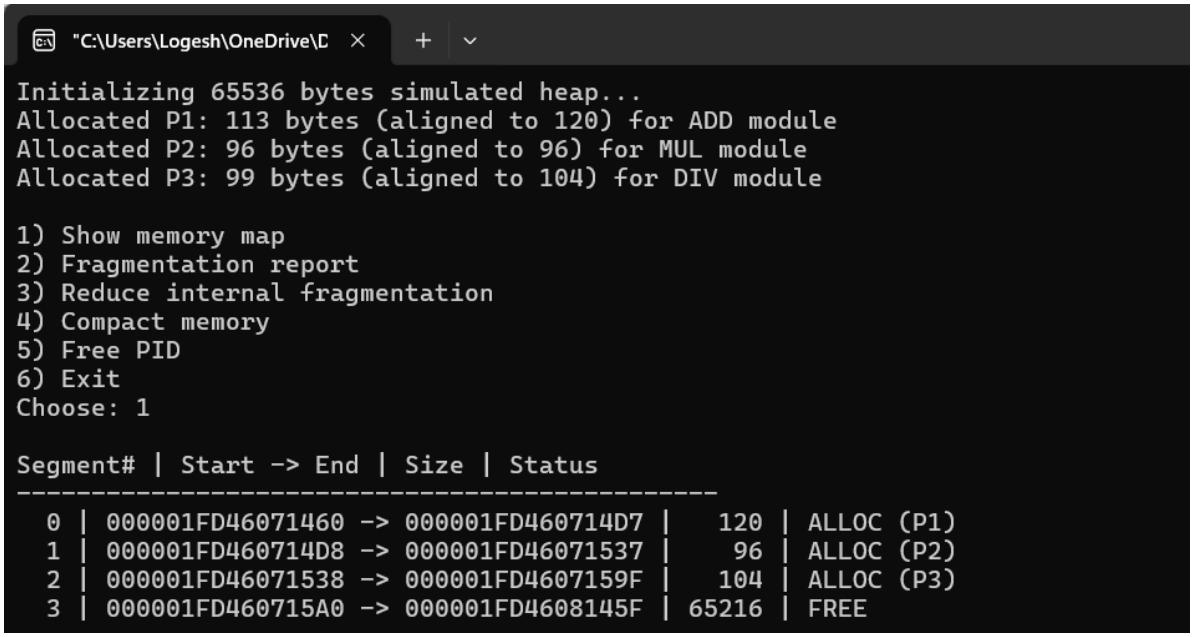
MUL.h

```
#ifndef MUL_H_INCLUDED
#define MUL_H_INCLUDED
int mul(int a, int b);
#endif // MUL_H_INCLUDED
```

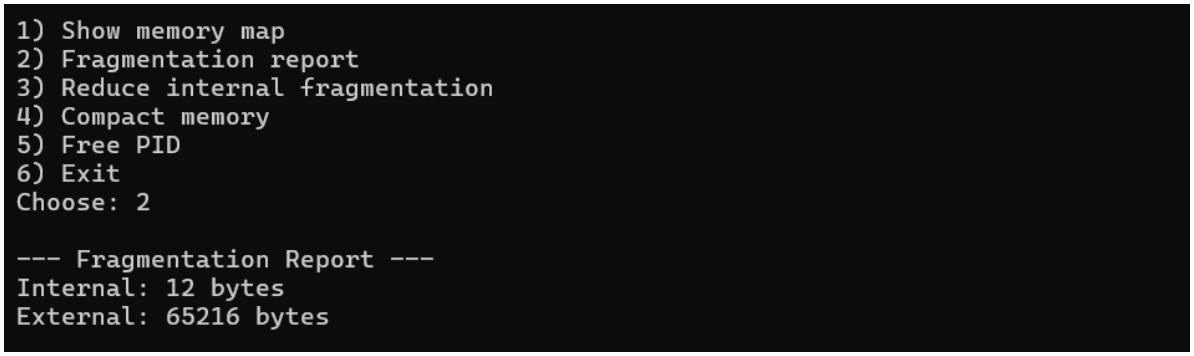
DIV.h

```
#ifndef DIV_H_INCLUDED
#define DIV_H_INCLUDED
int divide(int a, int b);
#endif // DIV_H_INCLUDED
```

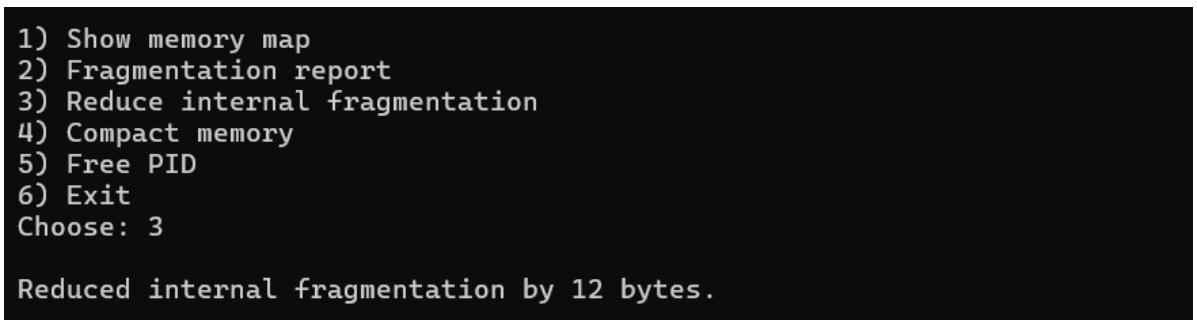
Inputs and Testing (for Different Cases)



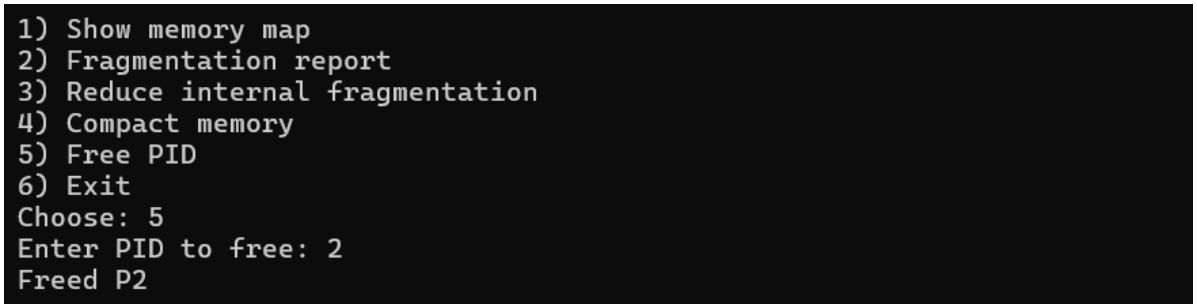
```
"C:\Users\Logesh\OneDrive\D
Initializing 65536 bytes simulated heap...
Allocated P1: 113 bytes (aligned to 120) for ADD module
Allocated P2: 96 bytes (aligned to 96) for MUL module
Allocated P3: 99 bytes (aligned to 104) for DIV module

1) Show memory map
2) Fragmentation report
3) Reduce internal fragmentation
4) Compact memory
5) Free PID
6) Exit
Choose: 1

Segment# | Start -> End | Size | Status
------------------------------------------------
  0 | 000001FD46071460 -> 000001FD460714D7 |   120 | ALLOC (P1)
  1 | 000001FD460714D8 -> 000001FD46071537 |    96 | ALLOC (P2)
  2 | 000001FD46071538 -> 000001FD4607159F |   104 | ALLOC (P3)
  3 | 000001FD460715A0 -> 000001FD4608145F | 65216 | FREE
```



```
1) Show memory map
2) Fragmentation report
3) Reduce internal fragmentation
4) Compact memory
5) Free PID
6) Exit
Choose: 2

--- Fragmentation Report ---
Internal: 12 bytes
External: 65216 bytes
```



```
1) Show memory map
2) Fragmentation report
3) Reduce internal fragmentation
4) Compact memory
5) Free PID
6) Exit
Choose: 3

Reduced internal fragmentation by 12 bytes.
```



```
1) Show memory map
2) Fragmentation report
3) Reduce internal fragmentation
4) Compact memory
5) Free PID
6) Exit
Choose: 5
Enter PID to free: 2
Freed P2
```

```
1) Show memory map
2) Fragmentation report
3) Reduce internal fragmentation
4) Compact memory
5) Free PID
6) Exit
Choose: 1

Segment# | Start -> End | Size | Status
-----------------------------------------------
  0 | 000001FD46071460 -> 000001FD460714D0 |   113 | ALLOC (P1)
  1 | 000001FD460714D1 -> 000001FD46071537 |   103 | FREE
  2 | 000001FD46071538 -> 000001FD4607159A |    99 | ALLOC (P3)
  3 | 000001FD4607159B -> 000001FD4608145F | 65221 | FREE
```

```
1) Show memory map
2) Fragmentation report
3) Reduce internal fragmentation
4) Compact memory
5) Free PID
6) Exit
Choose: 4
Compaction done.

1) Show memory map
2) Fragmentation report
3) Reduce internal fragmentation
4) Compact memory
5) Free PID
6) Exit
Choose: 1

Segment# | Start -> End | Size | Status
-----------------------------------------------
  0 | 000001FD46071460 -> 000001FD460714D0 |   113 | ALLOC (P1)
  1 | 000001FD460714D1 -> 000001FD46071533 |    99 | ALLOC (P3)
  2 | 000001FD46071534 -> 000001FD4608145F | 65324 | FREE
```

Results and Discussion

The simulator successfully allocates and manages memory dynamically. Internal and external fragmentation are calculated accurately, and functions for reducing fragmentation and compacting memory work as intended. Visualization of memory status through the console provides a clear understanding of heap behavior and memory management principles.

Conclusion

This project demonstrates the internal working of dynamic memory management using a simulated heap. By implementing allocation, deallocation, fragmentation handling, and compaction manually, students gain insight into how operating systems handle memory. The simulator serves as an educational tool for learning heap management, process memory allocation, and fragmentation control techniques in embedded software systems.