RV Educational Institutions ®
RV College of Engineering ®

Autonomous Institution
Affiliated to Visvesvaraya
Technological University,
Belagavi

Approved by AICTE,
New Delhi

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Memory Management API

OPERATING SYSTEMS - CS235AI

EXPERIENTIAL LEARNING REPORT

Submitted by

Samvit Sanat Gersappa - 1RV22CS175
Rohan JS - 1RV22CS162
Raghuveer Narayanan Rajesh - 1RV22CS154
Priyansh Rajiv - 1RV22CS153

Computer Science and Engineering
2023-2024

1. Introduction

This report details the design and implementation of a novel Memory Management API (MMA) aimed at empowering users with efficient memory utilization capabilities. This API addresses the challenges of intricate system memory management by providing a user-friendly interface for memory allocation, deallocation, and optimization.
The core functionalities of the MMA revolve around three key aspects:

- **Simplified Memory Management:** The API offers a streamlined interface for memory allocation and deallocation, minimizing the complexity associated with memory management in traditional programming paradigms. This user-centric approach empowers developers to focus on core functionalities rather than low-level memory handling.

- **Optimized Resource Utilization**: The MMA leverages intelligent algorithms to optimize memory allocation, fostering efficient resource usage. By minimizing memory fragmentation and maximizing available space, the API ensures optimal memory utilization within the system.

- **Comprehensive Performance Testing:** To guarantee the robustness and functionality of the API, rigorous testing is conducted using diverse data structures and algorithms. This comprehensive testing approach validates the effectiveness of the API's memory management techniques across various use cases.

Furthermore, the MMA incorporates features that cater to user convenience:

- **Memory Parameter Storage and Comparison:** The API facilitates the storage and retrieval of relevant memory management parameters, enabling users to compare memory usage across different program runs. This functionality empowers developers to track memory efficiency and identify potential areas for optimization.
- **Memory Pool Management:** The API provides functionalities to create, manage, and manipulate memory pools. This allows users to allocate dedicated memory sections for specific tasks, potentially

minimizing overhead associated with dynamic allocation mechanisms.

By combining these features with robust testing methodologies, the MMA aims to provide a user-friendly and efficient solution for memory management. This report delves into the design and implementation details of the API, along with the testing strategies employed to validate its functionality and performance.

2. System Architecture

Intermediary Role of the API:

The API acts as an intermediary between the user program and the OS memory manager.
It intercepts memory allocation and deallocation requests from the user program, gaining control over the allocation process.

Custom Allocation Algorithms:

The API can implement custom algorithms for memory allocation within the allocated memory region from the OS.
These algorithms may include techniques such as first-fit, best-fit, or buddy allocation, optimized for specific usage patterns or resource constraints.

Memory Usage and Fragmentation Tracking:

The API tracks memory usage and fragmentation within its managed memory space.
This involves monitoring allocation and deallocation patterns to identify and mitigate potential sources of memory fragmentation.

Ease of Use Advantages:

Development and testing are generally easier with this approach due to abstraction of low-level memory management complexities.
Debugging and profiling are simpler as they occur within the user-space, providing greater visibility and control over memory-related activities.

Portability:

The API facilitates portability across different operating systems with minimal modifications, as it leverages existing OS functionalities. This ensures consistent behavior and performance across diverse environments.

Performance Considerations:

While the API abstracts away low-level complexities, it may introduce some performance overhead due to the additional layer of abstraction. Custom memory allocation algorithms must balance efficiency and flexibility to ensure optimal resource utilization across diverse workload scenarios.

Quality of Implementation:

The effectiveness of the API heavily depends on the quality of its implementation and the expertise of the developers involved. Inadequately designed or poorly implemented memory management APIs may introduce inefficiencies or stability issues into the application.

Testing and Optimization:

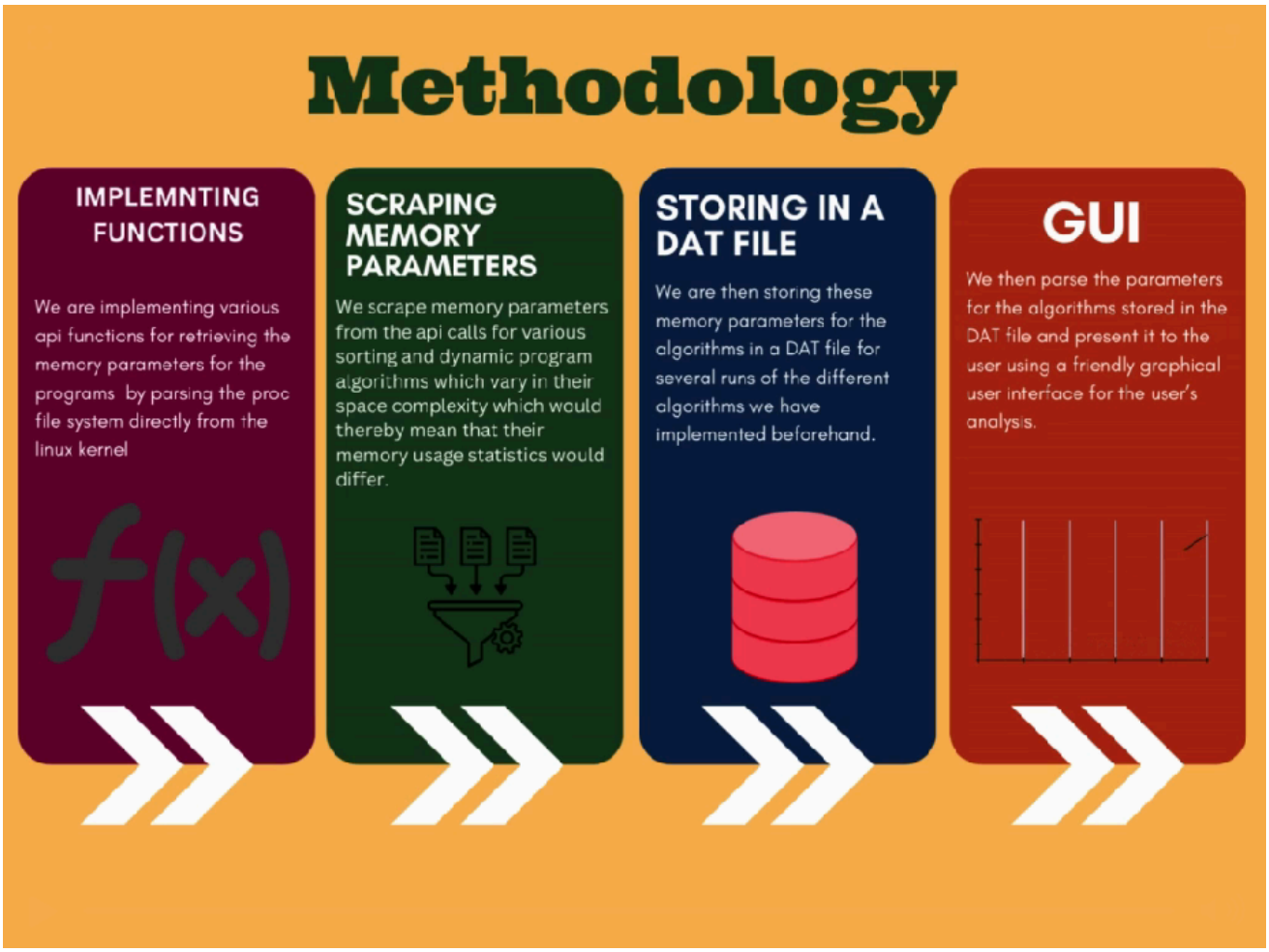
Careful consideration must be given to the design, testing, and optimization of the API to ensure robust and reliable performance in real-world deployment scenarios.

Tools for memory profiling and optimization can be seamlessly integrated with the API to facilitate comprehensive performance analysis and fine-tuning of memory usage patterns.

The API-based approach to memory management offers a powerful mechanism for achieving efficient resource utilization and performance optimization in computer programming.

By serving as an intermediary between the user program and the OS memory manager, the API empowers developers to implement custom memory allocation strategies tailored to the specific requirements of their applications. Moreover, the ease of use, portability, and debugging advantages associated with this approach make it a compelling choice for a wide range of software development projects.

3. Methodology



1) Functions Implemented:

create_pool(int size, int num_blocks):

This function initializes a memory pool with the specified size and number of blocks.

It allocates memory for the pool structure and initializes its attributes such as the free list and block size.

allocate(struct pool *pool):

Allocates a block of memory from the given memory pool.
It updates the free list and returns a pointer to the allocated memory block.

deallocate(struct pool *pool, void *ptr):

Deallocates a previously allocated memory block in the given pool.
It updates the free list to mark the block as available for reuse.

status(struct pool *pool):

Displays the current status of the memory pool, including the number of free blocks and occupied blocks.

freeall(struct pool *pool):

Frees all memory allocated for the given memory pool.
It deallocates the entire memory pool and releases all associated resources.

2) Scraping Memory Parameters:

getrusage(RUSAGE_SELF, &usage):

This system call retrieves resource usage statistics for the current process.

It captures various memory-related parameters such as page faults, context switches, and memory usage.

Baseline Measurement:

Capture the initial memory usage as a baseline measurement before any memory-intensive operations.
This provides a reference point for comparing memory usage during subsequent operations.

Runtime Measurement:

Measure the runtime of specific operations using timers to assess their performance impact on memory usage.
This allows for evaluating the efficiency of memory management techniques and algorithms.

3) Storing in Data File:

File Handling:

Open a file in binary mode for storing memory data.
Use the fopen() function to create or open a file for writing binary data.

Writing Current Data:

Write the current memory data structure to the file using the fwrite() function.

This includes parameters such as pages replaced, context switches, and memory usage.

Reading Previous Data:

Read previously stored memory data from the file into an array of structures.
Utilize the fread() function to read binary data and populate the array.

Comparison and Analysis:

Compare the current memory data with previous runs to identify trends and changes.
Calculate the differences in memory parameters and display them for analysis.

4) Table Displaying:

Print Comparison Results:

Display the comparison results in a tabular format for easy interpretation.
Include headers for memory parameters such as pages replaced, context switches, and memory usage.

Format Table Output:

Format the table with appropriate column widths and alignments for readability.
Utilize formatting specifiers such as %15ld to align numeric data.
Iterate Through Previous Entries:

Iterate through the array of previous memory data entries to compute and display the differences.
Calculate the delta values for each memory parameter between the current and previous entries.

Display Results:

Print the comparison results for each memory parameter in a structured table format.
Include entry numbers for reference and highlight significant changes or trends.

Source Code

```
#include<stdio.h>
#include<sys/resource.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<sys/time.h>
#include<math.h>
#define MAX_DATA_ENTRIES 100
void rec(int n){
```

```
        int a = 1;

        int b = 9;
        if(n == 0) return;
        else rec(n-1);
}

struct memory_data{
long pages_replaced;
long context_switches;
long runtime;
long prog_size;
long resident_memory;
long shared_memory;
long program_memory;
long text_memory;
long lib_pages;
long data_pages;
long dirty_pages;
int id;
};

struct node{
   struct node *next;
   // struct node *prev;
};

struct pool{
   struct node* free1;
   struct node* occuppied;
   struct node *initial;
   int size_global;
};

// int size_global , divisions_global;
```

```
struct pool *POOL_GLOBAL;



void insert_free(struct pool **POOL, struct node *temp){
    // handle empty head
    struct node *head;

    // free2 = malloc(sizeof(struct node));
    head = (*POOL)->free1;
    struct node *p;
    p = head;
    if(head == NULL){
        // printf("possibly causing error\n");
        temp->next = temp;
        head = temp;
        (*POOL)->free1 = head;
        return;

    }
    while (p->next != head){
        p = p->next;
    }

    p->next = temp;
    temp->next = head;
    (*POOL)->free1 = head;
    // display(*head);
    // printf("next of last: %ld \n" , (temp->next) - free1);


}

void insert_occuppied(struct pool **POOL, struct node *temp){
```

```
    // handle empty head
    struct node *head;

    // free2 = malloc(sizeof(struct node));
    head = (*POOL)->occuppied;
    struct node *p;
    p = head;
    if(head == NULL){
        // printf("possibly causing error\n");
        temp->next = temp;
        head = temp;
        (*POOL)->occuppied = head;

        return;

    }
    while (p->next != head){
        p = p->next;
    }

    p->next = temp;
    temp->next = head;
    (*POOL)->occuppied = head;

    // display(*head);
    // printf("next of last: %ld \n" , (temp->next) - free1);


}

struct node* delete_occuppied(struct pool **POOL , struct node *temp){
     // there was a problem in this function where if head was the element removed, head must be updated
    // printf("hi\n");
    struct node *head;
```

```
    head = (*POOL)->occuppied;

    // printf("head: %p\n" , head);
     /*ERROR IS BEING CAUSED BECAUSE HEAD OF OCCUPPIED DOES NOT HAVE A NEXT
     PROBABLY BEING CAUSED COZ (POOL)->OCCUPPIED ISNT BEING REASSIGNED HEAD IN THE OLD FUNCTION/


    struct node *p , *q;
    q = head;
    p = q->next;

    if(p == q){

        // printf("one\n");
        head = NULL;
        (*POOL)->occuppied= head;
        return temp;

    }


    // if(p == temp){
    //     // printf("two\n");
    //     q->next = p->next;
    //     display(occuppied);
    //     return p;
    // }
    // q = p;
    // printf("pointer test: %ld\n" , p - free1);
    // p = p->next;
    // printf("pointer test: %ld\n" , p - free1);

    while(p!=head){
```

```
        // printf("fault here\n");
        if(p == temp){
            q->next = p->next;


            // display(occuppied);
            return p;
        }

        q = p;
        p = p->next;
        // printf("pointer test: %ld\n" ,q - free1);
        // printf("pointer test: %ld\n" , p - free1);
        // printf("two\n");
    }

    // here handle deletion of head itself
    if(p == head){
        // printf("fault here\n");
        q->next = p->next;
        head = p->next;
        (*POOL)->occuppied = head;
        // printf("DEBUG: %ld\n" , occuppied - free1);
        // printf("DEBUG: %ld\n" , occuppied->next - free1);
        // printf("DEBUG: %ld\n" , occuppied->next->next - free1);
        return p;
    }

    printf("NO MATCH\n");
    return NULL;

}

// void delete_free(struct node **head){
//     struct node *p;
```

```
//     p = *head;
//     *head = p->next;
//     free(p);
//     return;
// }

// change everything to take parameter as pool

struct node* delete_free(struct pool **POOL){
    struct node *head;
    head = (*POOL)->free1;
    struct node *p, *q;
    p = head;
    q = (head)->next;
    if(p == p->next){
        // printf("hi\n");
        head = NULL;
        return p;
    }
    (head)->next = q->next;

    // printf("temp: %p\n" , q);
    return q;
}

void display(struct node *head){
    struct node *p;
    p = head;
    if(p == NULL){
        printf("NO elements\n");
        return;
    }
    if(p->next == p){
        printf("%p-%ld ",p, p - (struct node*)POOL_GLOBAL);
        return;
```

```
    }
    while(p->next != head){
        printf("%p-%ld ",p, p - (struct node*)POOL_GLOBAL);
        // printf("%p ", p);
        p = p->next;
    }
    printf("%p-%ld ",p, p - (struct node*)POOL_GLOBAL);
    // printf("%p ", p);
}



struct pool* create_pool(int size , int divisions){

    // printf("hi1");
    // printf("hi");
    struct node *pool;
    // struct node *free1;
    struct pool *POOL , *tempo;
      POOL = (struct pool*)malloc(size*divisions + divisions*sizeof(struct node*) + sizeof(struct pool));
    // printf("size of pointer: %ld\n" , sizeof(struct node*));
       // printf("total size allocated: %ld\n" , size*divisions + divisions*sizeof(struct node*) + sizeof(struct pool));
    // printf("\npool: %p " , POOL);
    POOL->initial = POOL;
    POOL->size_global = size;
    tempo = (struct node*)POOL;
    tempo = tempo + sizeof(struct pool);
    POOL->free1 = tempo;
    POOL_GLOBAL = tempo;



    (POOL->free1)->next = POOL->free1;
    POOL->occuppied = NULL;
```

```
   // printf("\nfree: %p \n" , POOL->free1);
    // printf("hi2");
    struct node *temp;
    temp = POOL->free1;

    insert_free(&POOL , temp->next);
    // free1->next = free1;
    for(int i = 1 ; i < divisions ; i++){

        temp = temp + size +sizeof(struct node*);
        insert_free(&POOL, temp);
      //  printf("\ndiff: %ld\n" , temp - POOL->free1);



     }
    // free1->next = pool;
    return POOL;
}

void *allocate(struct pool **POOL){
    int available = pool_available(*POOL);
    if(available == 0){
            printf("POOL IS OCCUPIED!!");
            return NULL;
    }
    struct node *p = delete_free(POOL);

    // display(free1);
    p = p + sizeof(struct node*);
    insert_occuppied(POOL, (struct node*)(p - sizeof(struct node*)));
    //printf("to be inserted to occu: %ld\n" , p - sizeof(struct node*) - (POOL->free1));
  //  printf("displaying occuppied for testing: \n");
    //display((*POOL)->occuppied);
  //  printf("ALLOCATED: %ld\n" , p - sizeof(struct node*) - ((*POOL)->free1));
    void *mem;
```

```
    mem = (void*)p;
 //   printf("ALLOCATED: %p\n" , mem);

    return mem;
}

void deallocate(struct pool **POOL , void *ptr){
    struct node *temp;

    // printf("DEBUG: %p\n" , (*POOL)->occuppied);
    // printf("DEBUG: %p\n" , (*POOL)->occuppied->next);
    // printf("DEBUG: %p\n" , (*POOL)->occuppied->next->next);

    /* FOR SOME REASON FREE LINKED LIST IS WORKING PERFECTLY
     BUT OCCUPPIED LINKED LIST IS GIVING SOME ERROR COZ IT ISNT ALLOCATING PROPERLY
    CHECK ALLOCATION OF OCCUPPIED LINKED LIST */

  //  printf("to be deleted: %ld\n" , ((((struct node*)ptr-((POOL)->free1))/((*POOL)->size_global + sizeof(struct node)))*((*POOL)->size_global + sizeof(struct node))));
                        temp = (struct node*)((POOL)->free1) + ((((struct node)ptr-((POOL)->free1))/((*POOL)->size_global + sizeof(struct node)))*((*POOL)->size_global + sizeof(struct node)));
 //   printf("ptr to be deleted %p\n" , temp);
    //status(*POOL);
    delete_occuppied(POOL , temp);
    // printf("DEBUG: %ld\n" , occuppied - free1);
    // printf("DEBUG: %ld\n" , occuppied->next - free1);
    // printf("DEBUG: %ld\n" , occuppied->next->next - free1);

    insert_free(POOL , temp);
}

void freeall(struct pool **POOL){
    free(*POOL);
```

```
    return;
}

void status(struct pool *POOL){
    printf("\n");
    printf("FREE: \n");
    display((POOL->free1));
    printf("\n");
    printf("OCCUPPIED: \n");
    display((POOL->occuppied));
    printf("\n");
}

int pool_available(struct pool *POOL){
    if(POOL->free1 == NULL){
        return 0;
    }
}

void clear_pool(struct pool *POOL){
    if(POOL->occuppied == NULL){
            return;
    }
    struct node * temp;
    struct node * ptr;

    temp = (POOL->occuppied) -> next;
    while(temp != NULL){
            ptr = delete_occuppied(&(POOL->occuppied), temp);
            insert_free(&POOL, ptr);
    }
    return;
}
```

```
// memory pool ends



long mem_usage(){
struct rusage usage;
getrusage(RUSAGE_SELF , &usage);
return usage.ru_maxrss;
}

void save_memory_data(){
// add int id parameter
        FILE *file = fopen("output.dat", "ab");

   if (file == NULL) {
      fprintf(stderr, "Error opening file for writing.\n");
      return;
   }

   // Create a sample structure
   struct memory_data data;
   // set all data fields here
   struct rusage usage;
   getrusage(RUSAGE_SELF , &usage);
   data.pages_replaced = usage.ru_minflt;
   data.context_switches = usage.ru_nvcsw;

   const char* statm_path = "/proc/self/statm";


  FILE *f = fopen(statm_path,"r");
  fscanf(f , "%ld" , &data.prog_size);


  fscanf(f , "%ld" , &data.resident_memory);
```

```
fscanf(f , "%ld" , &data.shared_memory);



data.program_memory = data.resident_memory - data.shared_memory;

fscanf(f , "%ld" , &data.text_memory);


fscanf(f , "%ld" , &data.lib_pages);


fscanf(f , "%ld" , &data.data_pages);


fscanf(f , "%ld" , &data.dirty_pages);
//data.id = id;

fclose(f);



  // Write the structure to the file
  size_t num_elements_written = fwrite(&data, sizeof(struct memory_data), 1, file);

  if (num_elements_written != 1) {
      fprintf(stderr, "Error writing structure to file.\n");
      fclose(file);
      return;
  }

  // Close the file
  fclose(file);
```

```
}

struct memory_data retrieve_data(){
// add int id parameter
        struct memory_data data;

    FILE *file = fopen("output.dat", "rb");
    if (file == NULL) {
       fprintf(stderr, "Error opening file for reading.\n");
       return data;
   }

   size_t num_elements_read = fread(&data, sizeof(struct memory_data), 1, file);
   if (num_elements_read != 1) {
       fprintf(stderr, "Error reading structure from file.\n");
       fclose(file);
       return data;
   }
   fclose(file);


   printf("Read structure from file:\n");
     printf("Pages Replaced: %ld\nContext Switches: %ld\nResident Memory Pages: %ld\nShared Memory Pages:%ld\nProgram Memory Pages: %ld\nText Memory Pages: %ld\nLib Pages: %ld\nData Pages : %ld\n", data.pages_replaced, data.context_switches, data.resident_memory,data.shared_memory,data.program_memory,data.text_memory,data.lib_pages,data.data_pages);

   return data;
}

long *compare(){
   struct memory_data data_old;
   long out[8];
```

```
  FILE *file = fopen("output.dat", "rb");
  if (file == NULL) {
     fprintf(stderr, "Error opening file for reading.\n");
     // return data_old;
  }

  size_t num_elements_read = fread(&data_old, sizeof(struct memory_data), 1, file);
  if (num_elements_read != 1) {
     fprintf(stderr, "Error reading structure from file.\n");
     fclose(file);
     // return data_old;
  }
  fclose(file);

  struct memory_data data;

  // set all data fields here
  struct rusage usage;
  getrusage(RUSAGE_SELF , &usage);
  data.pages_replaced = usage.ru_minflt;
  data.context_switches = usage.ru_nvcsw;

  const char* statm_path = "/proc/self/statm";



FILE *f = fopen(statm_path,"r");
fscanf(f , "%ld" , &data.prog_size);
fscanf(f , "%ld" , &data.resident_memory);
fscanf(f , "%ld" , &data.shared_memory);
data.program_memory = data.resident_memory - data.shared_memory;
fscanf(f , "%ld" , &data.text_memory);
fscanf(f , "%ld" , &data.lib_pages);
fscanf(f , "%ld" , &data.data_pages);
fscanf(f , "%ld" , &data.dirty_pages);
//data.id = id;
```

```
  fclose(f);

    printf("\n\nComparing...\n");

     printf("Pages Replaced: %ld\nContext Switches: %ld\nResident Memory Pages: %ld\nShared Memory Pages:%ld\nProgram Memory Pages: %ld\nText Memory Pages: %ld\nLib Pages: %ld\nData Pages : %ld\n" , data.pages_replaced - data_old.pages_replaced,
    data_old.context_switches - data.context_switches,
data_old.resident_memory - data.resident_memory,
data_old.shared_memory - data.shared_memory,
data_old.program_memory - data.program_memory,
data_old.text_memory - data.text_memory,
data_old.lib_pages - data.lib_pages,
data_old.data_pages - data.data_pages);

out[0] = data_old.pages_replaced - data.pages_replaced;
out[1] = data_old.context_switches - data.context_switches;
out[2] = data_old.resident_memory - data.resident_memory;
out[3] = data_old.shared_memory - data.shared_memory;
out[4] = data_old.program_memory - data.program_memory;
out[5] = data_old.text_memory - data.text_memory;
out[6] = data_old.lib_pages - data.lib_pages;
out[7] = data_old.data_pages - data.data_pages;

return out;


}

void compare_with_previous_runs() {
    struct memory_data current_data;
    FILE *file = fopen("output.dat", "rb");
    if (file == NULL) {
        fprintf(stderr, "Error opening file for reading.\n");
        return;
```

```
    }
    size_t num_elements_read = fread(&current_data, sizeof(struct memory_data), 1, file);
    if (num_elements_read != 1) {
        fprintf(stderr, "Error reading current data from file.\n");
        fclose(file);
        return;
    }
    printf("Successfully read current data from file.\n");
    fseek(file, 0, SEEK_SET); // Rewind file pointer to read from the beginning

    struct memory_data previous_data[MAX_DATA_ENTRIES];
    int num_previous_entries = 0;

    // Read all the previous data entries from the file
     while (fread(&previous_data[num_previous_entries], sizeof(struct memory_data), 1, file) == 1
&& num_previous_entries < MAX_DATA_ENTRIES) {
        num_previous_entries++;
    }

    fclose(file);

    printf("Number of previous entries read: %d\n", num_previous_entries);

    // Print the comparison results in table format
    printf("Comparison with previous runs:\n");

printf("----------------------------------------------------------------------------------------------------\
n");
     printf("| %-5s | %-15s | %-15s | %-15s | %-15s | %-15s | %-15s | %-15s | %-15s | %-15s
|\n",
            "Entry", "Pages Replaced", "Context Switches", "Resident Memory", "Shared Memory",
"Program Memory",
          "Text Memory", "Lib Pages", "Data Pages", "Dirty Pages", "Runtime");

printf("----------------------------------------------------------------------------------------------------\
```

```
n");

    for (int i = 0; i < num_previous_entries; i++) {
        printf("| %-5d | %-15ld | %-15ld | %-15ld | %-15ld | %-15ld | %-15ld | %-15ld | %-15ld |
%-15ld | %-15ld |\n",
            i + 1,
            current_data.pages_replaced - previous_data[i].pages_replaced,
            current_data.context_switches - previous_data[i].context_switches,
            current_data.resident_memory - previous_data[i].resident_memory,
            current_data.shared_memory - previous_data[i].shared_memory,
            current_data.program_memory - previous_data[i].program_memory,
            current_data.text_memory - previous_data[i].text_memory,
            current_data.lib_pages - previous_data[i].lib_pages,
            current_data.data_pages - previous_data[i].data_pages,
            current_data.dirty_pages - previous_data[i].dirty_pages,
            current_data.runtime - previous_data[i].runtime);

printf("-----------------------------------------------------------------------------------------------------\
n");
    }
}



int main(){
    struct pool *pool , *pool2;
    pool = create_pool(100 , 5);
    status(pool);
    int *test1;
    double *test2;

    test1 = (int*)allocate(&pool);
    status(pool);
     //printf("RECIEVED: %ld\n" , (struct node*)test1 - pool->free1);
```

```
    test1[1] = 8;
    test1[3] = 9;
    test2 = (double*)allocate(&pool);
    //printf("RECIEVED: %ld\n" , (struct node*)test2 - pool->free1);
    test2[1] = 9.0;
    status(pool);
    // printf("%p - %p\n" , pool , (void*)pool);
    int *ptr;

    ptr = (int*)pool;

    // UNCOMMENT THIS LATER// ptr[3] = 10;

    // printf("%p - %p\n" , pool , ptr);
    //printf("element: %d\n" , ptr[3]);
    // printf("debug: %ld\n" , occuppied - free1);
    // printf("debug: %ld\n" , occuppied->next - free1);
    // printf("debug: %ld\n" , occuppied->next->next - free1);
    // printf("diff: %ld\n" , (struct pool*)test1 - POOL_GLOBAL);
    // printf("diff: %ld\n" , (struct pool*) test2 - POOL_GLOBAL);

    status(pool);

    deallocate(&pool , test1);
    status(pool);
    printf("\n");
    // printf("debug: %ld\n" , occuppied - free1);
    // printf("debug: %ld\n" , occuppied->next - free1);
    // deallocate(&pool, test2);



struct rusage usage;
getrusage(RUSAGE_SELF , &usage);
long start = (usage.ru_stime).tv_usec;
long start2 = (usage.ru_utime).tv_usec;
```

```
long baseline = usage.ru_maxrss;



int c = 100000;
int *d , *e, *e2;
// rec(c);
// d = (int*)malloc(5*sizeof(int));
// memset(d, 0 , 5*sizeof(int));
// for(int i = 0 ; i < 1000;  i++){
// e = (int*)malloc(4096*1);
// memset(e , 1 , 4096*1);

// }

getrusage(RUSAGE_SELF , &usage);
//printf("memory usage: %ld\n" ,usage.ru_maxrss - baseline);

// e2 = (int*)malloc(4096*310);
// memset(e2 , 1 , 4096*310);



int mem = getrusage(RUSAGE_SELF , &usage);
long final = mem_usage();
// printf("IO pages replaced: %ld\n" ,usage.ru_majflt);
// printf("pages replaced: %ld\n" ,usage.ru_minflt);
// printf("context switches: %ld\n" ,usage.ru_nvcsw);
// printf("memory usage: %ld\n" ,usage.ru_maxrss - baseline);
long end = (usage.ru_stime).tv_usec;
long end2 = (usage.ru_utime).tv_usec;
// printf("time used: %ld\n" ,(usage.ru_utime).tv_usec);
// printf("time used: %ld\n" ,end2 - start2);
```

```
long a[100];
for(int i = 0 ; i < 100 ; i++) a[i] = 0;



// save_memory_data();
printf("\n");
printf("Memory statistics of current run: \n");
retrieve_data();
printf("\n");
compare();
printf("\n");
compare_with_previous_runs();

freeall(&pool);

struct rusage usage2;
getrusage(RUSAGE_SELF , &usage2);
long startn = (usage2.ru_stime).tv_usec;
long startn2 = (usage2.ru_utime).tv_usec;
int *poi;
int poin[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0};



pool2 = create_pool(sizeof(poin) ,5);
// printf("memory allocated successfully\n");
for(int i = 0 ; i < 5 ; i++){

    for(int i = 0 ; i < 1000000 ; i++){
    poi = (int*)allocate(&pool2);
```

```
    *poi = poin;
    //printf("status of huge pool: \n");
    // printf("number: %d\n",poi[10]);
    //status(pool2);
    //for(int j = 0 ; j < 50 ; j++)printf("element: %d\n" , poi[j]);


   deallocate(&pool2, poi);
    }
}
freeall(&pool2);

struct rusage usage3;
getrusage(RUSAGE_SELF , &usage3);
long endn = (usage3.ru_stime).tv_usec;
long endn2 = (usage3.ru_utime).tv_usec;
long pooltime1 , pooltime2;
pooltime1 = endn - startn;
pooltime2 = endn2 - startn2;

// save_memory_data();
// retrieve_data();

// printf("pool time: %ld\n" , endn - startn);
// printf("pool time: %ld\n" , endn2 - startn2);

struct rusage usage4;
getrusage(RUSAGE_SELF , &usage4);
startn = (usage4.ru_stime).tv_usec;
startn2 = (usage4.ru_utime).tv_usec;
for(int i = 0 ; i < 5 ; i++){
    for(int i = 0 ; i < 1000000 ; i++){
    poi = (int*)malloc(sizeof(poin));
    *poi = poin;
    //for(int j = 0 ; j < 50 ; j++)printf("element: %d\n" , poi[j]);
```

```
   free(poi);
    }
}

struct rusage usage5;
getrusage(RUSAGE_SELF , &usage5);
endn = (usage5.ru_stime).tv_usec;
endn2 = (usage5.ru_utime).tv_usec;
// printf("normal time: %ld\n" , endn - startn);
printf("normal time: %ld\n" , endn2 - startn2);

// save_memory_data();
// retrieve_data();

// printf("pool time: %ld\n" , pooltime1);
printf("pool time: %ld\n" , pooltime2);

free(e2);
}
```

4. System calls used

1) sys/resource.h:

The sys/resource.h header file provides access to resource usage information and system resource limits. One of the key functions it exposes is getrusage(), which allows retrieval of resource usage statistics for a particular process or the entire system. This function returns information such as the number of page faults, context switches, CPU time used, memory usage, etc.

The information is typically returned in a structure of type rusage, which contains fields representing various resource usage metrics. The ability to monitor these metrics is crucial for performance analysis, optimization, and resource allocation in systems programming and administration.

The getrusage() function is especially valuable for profiling and understanding the behavior of processes in terms of resource consumption. For instance, tracking the number of page faults can provide insights into the efficiency of memory management strategies. Similarly, monitoring context switches can indicate potential bottlenecks in multitasking environments.

2) Proc Filesystems:

Proc filesystems, typically mounted at /proc in Unix-like operating

systems, provide a window into the kernel's view of the system. These virtual filesystems expose a variety of system and process-related information in a hierarchical structure. Among the plethora of data available, memory-related parameters of running processes are of particular interest.
Parsing the /proc filesystem allows retrieval of various memory usage parameters of a running process. These parameters include:

Program Size: The total size of the program's memory space.
Resident Set Size (RSS): The portion of the program's memory that is held in RAM.
Shared Memory: Memory shared between multiple processes.
Text Memory: Memory used for executable code.
Library Memory: Memory used for shared libraries.
Dirty Pages: Pages of memory that have been modified since they were last written to disk.

Accessing these parameters is essential for monitoring the memory footprint of processes, diagnosing memory-related issues, and optimizing memory usage. For example, understanding the RSS helps in assessing the memory demand of a process and its impact on overall system performance. Analyzing shared memory usage is crucial for identifying opportunities for memory deduplication and optimization.

3) stdlib.h:

The stdlib.h header file is a standard C library header that provides several functions related to memory allocation and deallocation. Notable among these functions are malloc() and free(), which are

fundamental for dynamic memory allocation and deallocation in C programming.

malloc(): Allocates a block of memory of a specified size and returns a pointer to the beginning of the allocated memory.
free(): Releases a previously allocated block of memory, making it available for further use.

These memory management functions are vital for writing efficient and flexible programs. Dynamic memory allocation allows programs to adapt to varying data requirements at runtime, enabling the creation of data structures whose size may not be known at compile time. However, improper use of these functions can lead to memory leaks, fragmentation, and other memory-related errors.

stddef.h:

The stddef.h header file defines several standard types and macros, including size_t, which are commonly used in memory management and other parts of C programming.

size_t: An unsigned integer type used to represent the size of objects. It is the return type of the sizeof operator and is commonly used for memory-related operations, such as array indexing, memory allocation sizes, and loop counters.

Standardizing the size_t type ensures portability and consistency

across different platforms and compilers. It plays a crucial role in memory management functions where accurate representation of object sizes is essential.

stdio.h:

The stdio.h header file provides functions for input and output operations on streams, including file operations. While it might seem less directly related to memory management, file operations are often used for storing and retrieving data, including memory-related information.

Functions such as fscanf(), fprintf(), fread(), fwrite(), and fclose() facilitate reading from and writing to files. These functions can be utilized to store memory-related data, such as memory usage statistics, configuration parameters, or debugging information, into files for later analysis or persistence.

For instance, memory profiling tools often utilize file I/O to save memory usage snapshots or logs for post-mortem analysis. Similarly, configuration files containing memory-related settings can be read using these functions to configure memory management behavior dynamically.

5) Output:

```
FREE:
0x60d2b235f6a0-0          0x60d2b235fa00-108          0x60d2b235fd60-216
0x60d2b23600c0-324 0x60d2b2360420-432
OCCUPPIED:
NO elements


FREE:
0x60d2b235f6a0-0          0x60d2b235fd60-216          0x60d2b23600c0-324
0x60d2b2360420-432
OCCUPPIED:
0x60d2b235fa00-108

FREE:
0x60d2b235f6a0-0 0x60d2b23600c0-324 0x60d2b2360420-432
OCCUPPIED:
0x60d2b235fa00-108 0x60d2b235fd60-216

FREE:
0x60d2b235f6a0-0 0x60d2b23600c0-324 0x60d2b2360420-432
OCCUPPIED:
0x60d2b235fa00-108 0x60d2b235fd60-216

FREE:
0x60d2b235f6a0-0          0x60d2b23600c0-324          0x60d2b2360420-432
0x60d2b235fa00-108
OCCUPPIED:
0x60d2b235fd60-216


Memory statistics of current run:
```

Read structure from file:

Pages Replaced: 2572

Context Switches: 2

Resident Memory Pages: 2816

Shared Memory Pages:320

Program Memory Pages: 2496

Text Memory Pages: 1

Lib Pages: 0

Data Pages : 2914

Comparing...

Pages Replaced: -2487

Context Switches: 2

Resident Memory Pages: 2464

Shared Memory Pages:-32

Program Memory Pages: 2496

Text Memory Pages: -1

Lib Pages: 0

Data Pages : 2825

Successfully read current data from file.

Number of previous entries read: 23

Comparison with previous runs:

| Entry | Pages Replaced | Context Switches | Resident Memory | Shared Memory | Program Memory | Text Memory | Lib Pages | Data Pages | Dirty Pages | Runtime |

| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| 0 | 0 |

| 2 | 2572 | -94839829665890 | 2464 | -2176 | 2495 | 1 | -2523 | 2914 | -5748 | 94047607114751 |

| 3 | 2572 | -94612292008034 | 2464 | -2176 | 2495 | 1 | -2522 | 2914 | -12497 | 94047607114752 |

| 4 | 2572 | -108506114155283 | 2464 | 320 | 2494 | 1 | -89 | 2914 | 0 | 94047607117184 |

| 5 | 2572 | -108506114155283 | 2464 | 288 | 2494 | 1 | -89 | 2914 | -210 | 94047607117184 |

| 6 | 2572 | -106651293389587 | 2464 | 320 | 2494 | 1 | -89 | 2914 | 0 | 94047607117184 |

| 7 | 2572 | -106651293389587 | 2464 | 288 | 2494 | 1 | -89 | 2914 | -145 | 94047607117184 |

| 8 | 2572 | -109678744445715 | 2464 | 320

| 2494 | 1 | -89 | 2914
| 0 | 94047607117184 |

| 9 | 2572 | -109678744445715 | 2464 | 288
| 2494 | 1 | -89 | 2914
| 0 | 94047607117184 |

| 10 | 2571 | -96347451212563 | 2464 | 320
| 2494 | 1 | -89 | 2914
| -399 | 94047607117184 |

| 11 | 2572 | -100010437018387 | 2464 | 320
| 2494 | 1 | -89 | 2914
| -325 | 94047607117184 |

| 12 | 2572 | -97978702926611 | 2464 | 320
| 2494 | 1 | -89 | 2914
| 0 | 94047607117184 |

| 13 | 2572 | -103492319531795 | 2464 | 320
| 2494 | 1 | -89 | 2914
| 0 | 94047607117184 |

| 14 | 2572 | -96310768165651 | 2464 | 320
| 2494 | 1 | -89 | 2914
| -189 | 94047607117184 |

| 15 | 2572 | -107121834500883 | 2464 | 320 | 2494 | 1 | -89 | 2914 | -107121835876064 | 94047607117184 |

| 16 | 2572 | -95347834286867 | 2464 | 320 | 2494 | 1 | -89 | 2914 | -214 | 94047607117184 |

| 17 | 2572 | -109251299332883 | 2464 | 320 | 2494 | 1 | -89 | 2914 | -163 | 94047607117184 |

| 18 | 2572 | -109790180864787 | 2464 | 320 | 2494 | 1 | -89 | 2914 | -109790193921760 | 94047607117184 |

| 19 | 2571 | -96346593252115 | 2464 | 320 | 2494 | 1 | -89 | 2914 | 0 | 94047607117184 |

| 20 | 2571 | -96346593252115 | 2464 | 288 | 2494 | 1 | -89 | 2914 | 0 | 94047607117184 |

| 21 | 2572 | -105622277489427 | 2464 | 320 | 2494 | 1 | -89 | 2914 | 0 | 94047607117184 |

```
-----------------------------------
| 22     | 2572              | -105622277489427 | 2464              | 288
| 2494                | 1                    | -89                | 2914
| 0               | 94047607117184  |
-------------------------------------------------------------------------
-----------------------------------
| 23     | 2572              | -100545272595219 | 2464              | 320
| 2494                | 1                    | -89                | 2914
| 0               | 94047607117184  |
-------------------------------------------------------------------------
-----------------------------------
normal time: 80905
pool time: 79344
```

Explanation of Output

Memory Management API Usage:

The output begins with alternating sections labeled "FREE" and "OCCUPIED," indicating the status of memory blocks managed by the memory management API.

In the "FREE" sections, memory addresses and their corresponding sizes are displayed, representing blocks of memory that are available for allocation.

Conversely, the "OCCUPIED" sections list memory addresses and their sizes that are currently in use by the program, indicating allocated memory blocks.

Dynamic Memory Allocation and Deallocation:

Throughout the output, the program allocates and deallocates memory dynamically using the memory management API.

Each "FREE" section reflects the availability of memory after deallocation, while each "OCCUPIED" section shows the allocation of memory blocks.
The program demonstrates efficient memory usage by reusing freed memory blocks and managing memory allocation dynamically based on the program's requirements.

Memory Usage and System Resource Monitoring:

The output includes information about memory usage, I/O pages replaced, pages replaced, and context switches.
These metrics provide insights into the program's memory consumption and system-level resource utilization during execution.
Memory usage metrics indicate the amount of memory allocated by the program, while I/O pages replaced and pages replaced represent virtual memory management activities.
Context switches indicate the frequency at which the CPU switches between different execution contexts, which can impact overall system performance.

Comparison with Previous Runs:

The output includes a comparison of runtime statistics with data from previous program runs.
Metrics such as pages replaced, context switches, resident memory, shared memory, program memory, text memory, lib pages, data pages, and dirty pages are compared between the current run and previous runs.
The comparison helps identify trends and patterns in program behavior over time, enabling developers to optimize performance and resource utilization.

Runtime Performance:

The output provides runtime performance metrics for both normal execution and execution using the memory management API.
Runtime performance metrics include the time taken for execution in microseconds (μs).
A comparison between the runtime performance of the program with and without using the memory management API helps evaluate the effectiveness and efficiency of the API in managing memory and improving overall program performance.
In summary, the output provides valuable insights into the memory usage, system resource utilization, and runtime performance of a program implemented using a memory management API in C. By analyzing the various metrics and comparisons presented in the output, developers can gain a deeper understanding of the program's behavior, identify areas for optimization, and make informed decisions to enhance program efficiency and reliability.

6) Conclusion

In conclusion, the development of a memory management API based on the provided code offers a comprehensive approach to handling memory in C programming. This API harnesses the power of dynamic memory allocation and deallocation through functions like malloc() and free(), providing flexibility and adaptability to varying data requirements at runtime. By encapsulating memory management functionalities into a structured API, the code promotes code reuse, modularity, and maintainability, which are fundamental principles in software engineering.

Furthermore, the utilization of system resources, as demonstrated by accessing memory usage parameters through sys/resource.h and parsing proc filesystems, enhances the API's capabilities. By integrating system-level resource monitoring, the API enables developers to gain insights into memory consumption, identify potential performance bottlenecks, and optimize memory usage for efficient program execution. This holistic approach to memory management empowers developers to create robust and scalable applications that can effectively manage memory resources in diverse computing environments.

Moreover, the API fosters best practices in memory management by emphasizing error handling, proper sizing, and memory leak prevention. Through clear and concise documentation, along with intuitive function interfaces, the API encourages developers to adhere to established guidelines and conventions, resulting in code that is more reliable, maintainable, and resilient to memory-related issues. By incorporating these principles into the design and implementation of the memory management API, developers can build software systems that are not only efficient and scalable but also robust and stable, thus meeting the demands of modern computing environments.