RV College of Engineering®

Go, change the world

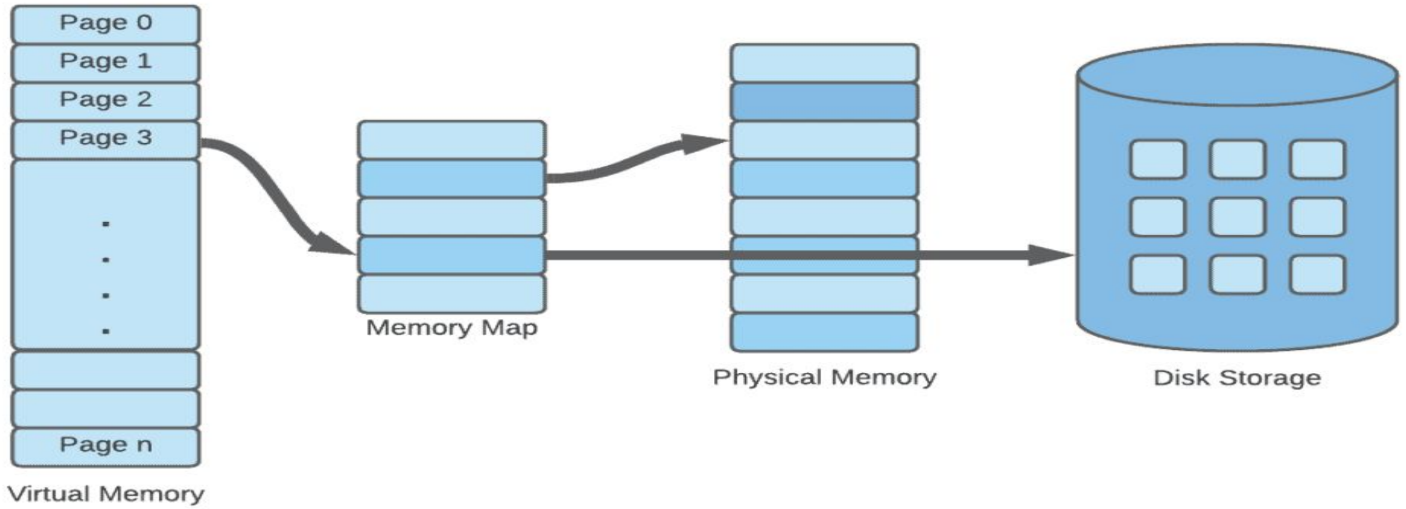
Department of Computer Science and Engineering

Course : Operating Systems
Course Code : CS235AI
Topic : Virtual Memory Analytics Suite

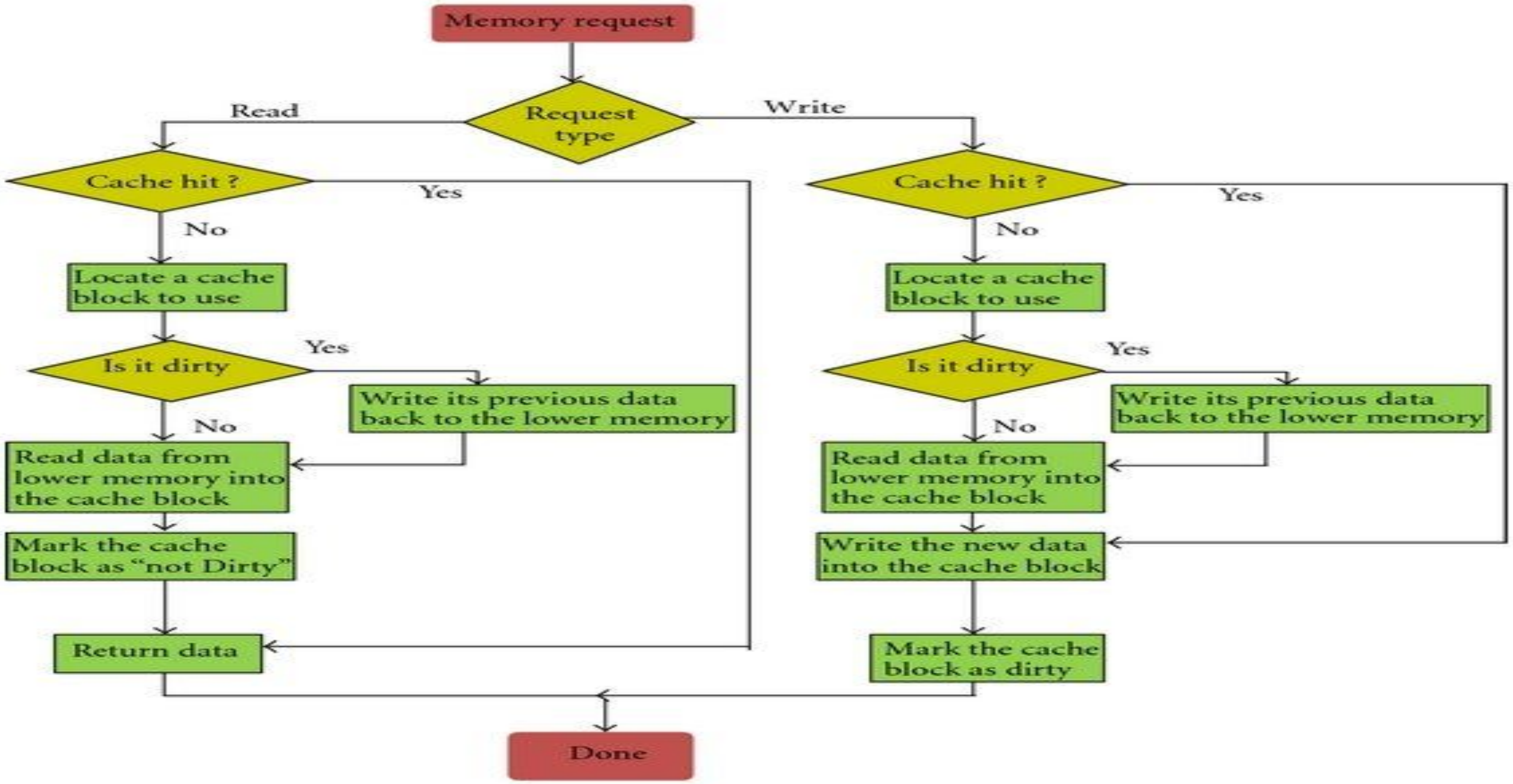
SEMESTER	III	
BRANCH	Computer Science and Engineering	
USN	Name	Submitted to
1RV22CS222	Vaibhav U Navalagi	**Dr. Jyoti Shetty** Assistant Professor
1RV22CS219	Tejas Ganesh Hegde	

Problem Statement

Develop a comprehensive Virtual Memory Analytics Suite that employs advanced algorithms to analyze, optimize, and monitor virtual memory usage in computing systems, enhancing overall performance and resource efficiency.



Flowchart


Memory request
Request type
Read
Write
Cache hit ?
Yes
No
Locate a cache block to use
Is it dirty
Yes
Write its previous data back to the lower memory
No
Read data from lower memory into the cache block
Mark the cache block as "not Dirty"
Return data
Cache hit ?
Yes
No
Locate a cache block to use
Is it dirty
Yes
Write its previous data back to the lower memory
No
Read data from lower memory into the cache block
Write the new data into the cache block
Mark the cache block as dirty
Done

Relevance

Relevance of the project to the course:

This project is highly relevant to courses in operating systems, computer architecture, and system programming. It allows students to gain a deeper understanding of virtual memory management, system performance analysis, and low-level programming concepts.

Methodology

Methodology (Flow chart/block diagram/system architecture):

The methodology involves designing and implementing a suite of tools to monitor and analyze virtual memory usage. This could include components such as:

Data collection modules to gather information about virtual memory usage and page faults. Data processing and analysis modules to interpret collected data and generate insights. User interface components to visualize the analyzed data. A simplified flowchart might illustrate the process of data collection, analysis, and presentation in the Virtual Memory Analytics Suite.

Tools/API Used

Research papers/Blogs/Books referred:

- "Understanding the Linux Virtual Memory Manager" by Mel Gorman
- "Operating System Concepts" by Abraham Silberschatz, Peter B. Galvin, Greg Gagne
- Various research papers on virtual memory management algorithms and techniques.

Tools/APIs used:

- C programming language for system-level programming.
- System calls such as getrusage for gathering memory usage statistics.
- Visualization libraries such as matplotlib or gnuplot for generating graphical representations of data.

Program code - 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>
// Function to get virtual memory usage
void getVirtualMemoryUsage() {
    struct rusage usage;
    if (getrusage(RUSAGE_SELF, &usage) == 0)
   {
        printf("Virtual Memory Usage: %ld KB\n", usage.ru_maxrss);
    }
 else {
        printf("Failed to get virtual memory usage.\n");
    }
 }
```

Program code - 1:

```
// Function to analyze page faults
void analyzePageFaults() {
    struct rusage usage;
    if (getrusage(RUSAGE_SELF, &usage) == 0) {
        printf("Page Faults (soft page faults): %ld\n", usage.ru_minflt);
        printf("Page Faults (hard page faults): %ld\n", usage.ru_majflt);
    }
    else {
        printf("Failed to analyze page faults.\n");
    }  }
// Function to track memory allocations
void* customMalloc(size_t size) {
    void* ptr = malloc(size);
    if (ptr != NULL) {
        printf("Allocated %zu bytes at address %p\n", size, ptr);
    }
```

Program code - 1:

```
else {
        printf("Memory allocation failed\n");
    }
    return ptr;
}
// Function to release memory allocations
void customFree(void* ptr) {
    free(ptr);
    printf("Freed memory at address %p\n", ptr);
}
// Function to plot memory usage over time using gnuplot
void plotMemoryUsage() {
    FILE *gnuplotPipe = popen("gnuplot -persistent", "w");
    if (gnuplotPipe != NULL) {
        fprintf(gnuplotPipe, "set title 'Memory Usage Over Time'\n");
        fprintf(gnuplotPipe, "set xlabel 'Time (s)'\n");
```

Program code - 1:

```
  fprintf(gnuplotPipe, "set ylabel 'Memory Usage (KB)'\n");
  fprintf(gnuplotPipe, "plot 'memory_usage.dat' with linespoints\n");
  fclose(gnuplotPipe);
  }
 else {
   printf("Failed to open gnuplot pipe.\n");
    }
}
int main() {
    // Example usage of the virtual memory analytics suite
    getVirtualMemoryUsage();
    analyzePageFaults();
    // Example memory allocation and deallocation
    int* arr = (int*)customMalloc(10 * sizeof(int));
    if (arr != NULL) {
        // Use allocated memory
```

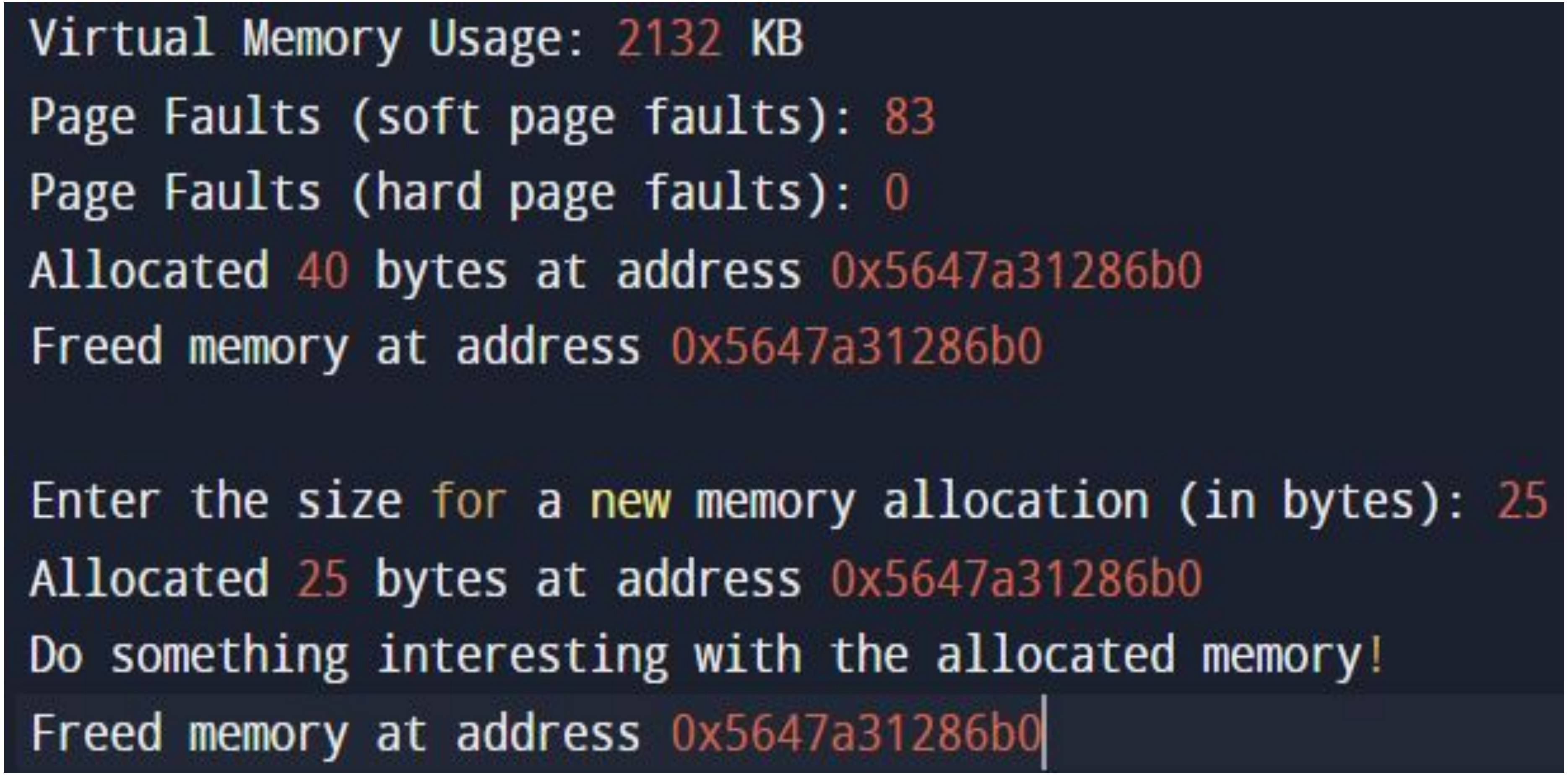
Program code - 1:

```
for (int i = 0; i < 10; ++i) {
            arr[i] = i;
        }
        // Free allocated memory
        customFree(arr);
    }
    // Plot memory usage over time
    plotMemoryUsage();
    // Additional cases with user inputs
    printf("\nEnter the size for a new memory allocation (in bytes): ");
    size_t newSize;
    scanf("%zu", &newSize);
    // Example of user-defined memory allocation
    void* userAllocatedMemory = customMalloc(newSize);
    if (userAllocatedMemory != NULL) {
        // Use allocated memory
```

Program code - 1:

```
printf("Do something interesting with the allocated memory!\n");
        // Free user-defined memory allocation
        customFree(userAllocatedMemory);
    }
    return 0;
}
```

Output



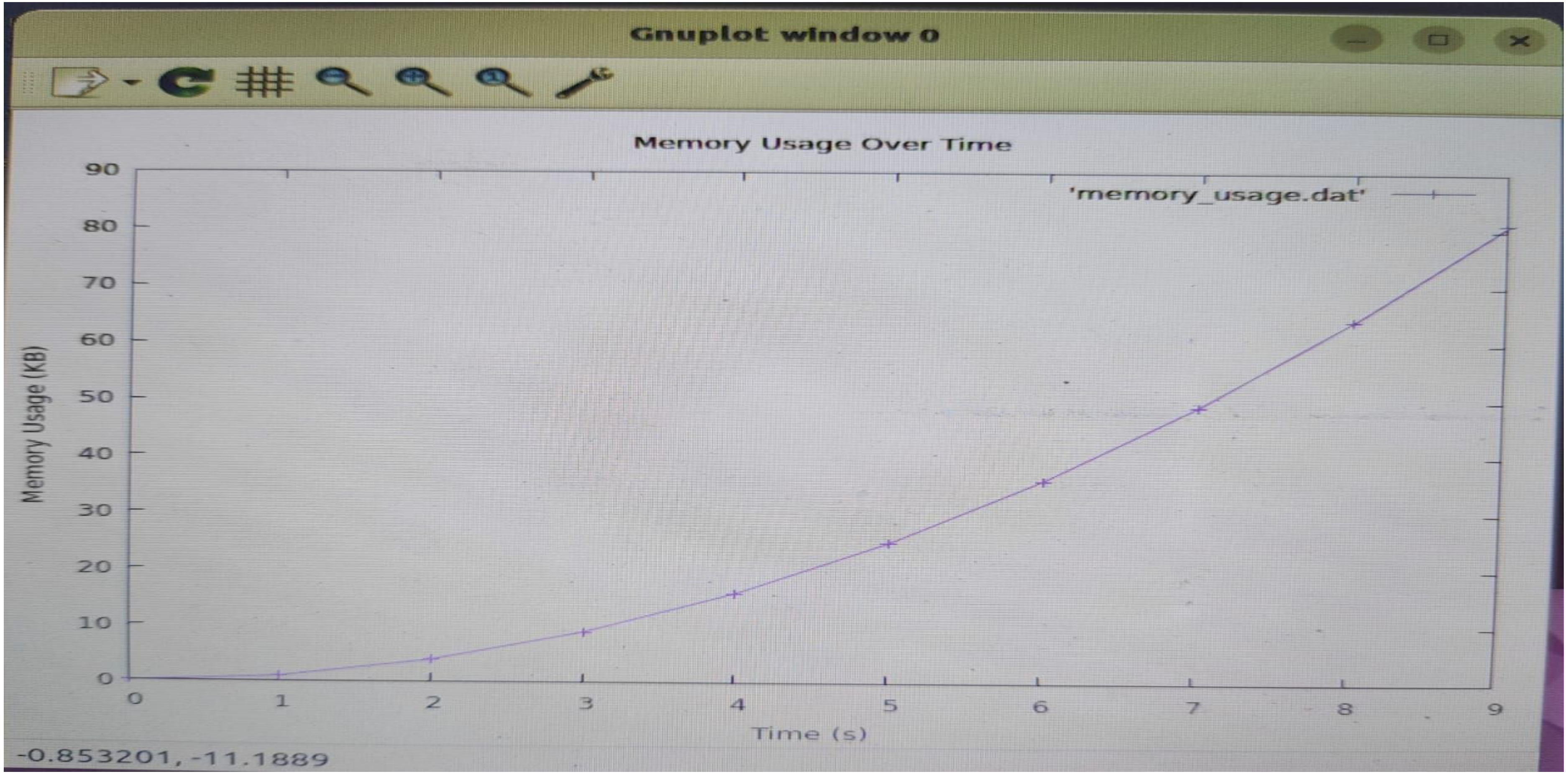
```
Virtual Memory Usage: 2132 KB
Page Faults (soft page faults): 83
Page Faults (hard page faults): 0
Allocated 40 bytes at address 0x5647a31286b0
Freed memory at address 0x5647a31286b0

Enter the size for a new memory allocation (in bytes): 25
Allocated 25 bytes at address 0x5647a31286b0
Do something interesting with the allocated memory!
Freed memory at address 0x5647a31286b0
```

RV College of Engineering®

GNUPLOT

Go, change the world



Analysis of Program-1:

1)getVirtualMemoryUsage(): Retrieves virtual memory usage statistics using **getrusage** and prints the maximum resident set size in kilobytes (**usage.ru_maxrss**).

2)analyzePageFaults(): Retrieves and analyzes page fault statistics, printing both soft and hard page fault counts (**usage.ru_minflt** and **usage.ru_majflt** respectively).

3)customMalloc(size_t size): A wrapper function for **malloc** that allocates memory of specified size, prints the allocated memory size and address, and returns the allocated pointer.

4)customFree(void* ptr): A wrapper function for **free** that deallocates memory pointed to by the given pointer and prints the address of the freed memory.

5)plotMemoryUsage(): Uses **gnuplot** to plot memory usage over time. It opens a pipe to gnuplot, sends commands to plot data from a file named **memory_usage.dat**, and closes the pipe.

Analysis of Program-1:

In the main function:

1)It first demonstrates the usage of virtual memory analytics suite by calling **getVirtualMemoryUsage()** and **analyzePageFaults().**

2)It allocates an integer array of size 10 using **customMalloc**, populates it with values, and frees it using **customFree**.

3)It plots memory usage over time using **plotMemoryUsage**.

4)It prompts the user to enter the size for a new memory allocation, allocates memory of the specified size using **customMalloc**, performs some operation (as indicated by the comment), and frees the allocated memory.

5)Finally, it **returns 0** to indicate successful program execution.

Program code - 2:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <dirent.h>
#include <string.h>
// Function to get virtual memory usage
void getVirtualMemoryUsage() {
    struct rusage usage;
    if (getrusage(RUSAGE_SELF, &usage) == 0) {
        printf("Virtual Memory Usage: %ld KB\n", usage.ru_maxrss);
    } else {
        printf("Failed to get virtual memory usage.\n");
    }
}
```

Program code - 2:

```
// Function to analyze page faults
void analyzePageFaults() {
    struct rusage usage;
    if (getrusage(RUSAGE_SELF, &usage) == 0) {
        printf("Page Faults (soft page faults): %ld\n", usage.ru_minflt);
        printf("Page Faults (hard page faults): %ld\n", usage.ru_majflt);
    } else {
        printf("Failed to analyze page faults.\n");
    }
}
// Function to track memory allocations
void* customMalloc(size_t size) {
    void* ptr = malloc(size);
    if (ptr != NULL) {
        printf("Allocated %zu bytes at address %p\n", size, ptr);
    }
```

Program code - 2:

```
else {
        printf("Memory allocation failed\n");
    }
    return ptr;
}
// Function to release memory allocations
void customFree(void* ptr) {
    free(ptr);
    printf("Freed memory at address %p\n", ptr);
}
// Function to plot memory usage over time using gnuplot
void plotMemoryUsage() {
    FILE *gnuplotPipe = popen("gnuplot -persistent", "w");
    if (gnuplotPipe != NULL) {
        fprintf(gnuplotPipe, "set title 'Memory Usage Over Time'\n");
        fprintf(gnuplotPipe, "set xlabel 'Time (s)'\n");
```

Program code - 2:

```
     fprintf(gnuplotPipe, "set ylabel 'Memory Usage (KB)'\n");
        fprintf(gnuplotPipe, "plot 'memory_usage.dat' with linespoints\n");
        fclose(gnuplotPipe);
    }
else {
        printf("Failed to open gnuplot pipe.\n");
    }
}
// Function to get memory usage for a process
void getProcessMemoryUsage(const char *pid) {
    char filename[256];
    snprintf(filename, sizeof(filename), "/proc/%s/statm", pid);
    FILE *fp = fopen(filename, "r");
    if (fp != NULL) {
        unsigned long size, resident, share, text, lib, data, dt;
```

Program code - 2:

```
if (fscanf(fp, "%lu %lu %lu %lu %lu %lu %lu", &size, &resident, &share, &text, &lib, &data, &dt) == 7) {
            printf("Process ID: %s\n", pid);
            printf("Size: %lu pages\n", size);
            printf("Resident: %lu pages\n", resident);
            printf("Share: %lu pages\n", share);
            printf("Text: %lu pages\n", text);
            printf("Library: %lu pages\n", lib);
            printf("Data+Stack: %lu pages\n", data);
            printf("Dirty: %lu pages\n", dt);
            printf("\n");
        }
        fclose(fp);
    } else {
        printf("Failed to open file for process: %s\n", pid);
    }
}
```

Program code - 2:

```
// Function to get process information
void getProcessInfo() {
    printf("Getting information about processes...\n");
    DIR *dir = opendir("/proc");
    if (dir == NULL) {
        perror("Failed to open /proc directory");
        exit(EXIT_FAILURE);
    }
    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
        if (entry->d_type == DT_DIR && atoi(entry->d_name) != 0) {
            getProcessMemoryUsage(entry->d_name);
        }
    }
    closedir(dir);
}
```

Program code - 2:

```
int main() {
    // Example usage of the virtual memory analytics suite
    getVirtualMemoryUsage();
    analyzePageFaults();
    // Example memory allocation and deallocation
    int* arr = (int*)customMalloc(10 * sizeof(int));
    if (arr != NULL) {
        // Use allocated memory
        for (int i = 0; i < 10; ++i) {
            arr[i] = i;
        }
        // Free allocated memory
        customFree(arr);
    }
    // Plot memory usage over time
    plotMemoryUsage();
```

Program code - 2:

```
// Get process information
    getProcessInfo();
    // Additional cases with user inputs
    printf("\nEnter the size for a new memory allocation (in bytes): ");
    size_t newSize;
    scanf("%zu", &newSize);
    // Example of user-defined memory allocation
    void* userAllocatedMemory = customMalloc(newSize);
    if (userAllocatedMemory != NULL) {
        // Use allocated memory
        printf("Do something interesting with the allocated memory!\n");
        // Free user-defined memory allocation
        customFree(userAllocatedMemory);
    }
    return 0;
}
```

Additional Output:

```
Process ID: 4306
Size: 74866 pages
Resident: 2208 pages
Share: 1984 pages
Text: 14 pages
Library: 0 pages
Data+Stack: 20483 pages
Dirty: 0 pages

Process ID: 4321
Size: 701794 pages
Resident: 15987 pages
Share: 12026 pages
Text: 2 pages
Library: 0 pages
Data+Stack: 21680 pages
Dirty 0 pages
```

Additional Output:

```
Process ID: 4369
Size: 99728 pages
Resident: 7809 pages
Share: 6346 pages
Text: 14 pages
Library: 0 pages
Data+Stack: 9197 pages
Dirty: 0 pages

Process ID: 4375
Size: 693 pages
Resident: 384 pages
Share: 384 pages
Text: 1 pages
Library: 0 pages
Data+Stack: 89 pages
Dirty: 0 pages
```

Analysis of Program - 2

This output is produced by the function **getProcessMemoryUsage (const char *pid)** when it encounters a process directory in the **/proc** directory. In this case, the process directory has the name "60". **There are many process directories in the /proc directory**. Consider one process directory having:

Process ID: This is the process ID of the process whose memory usage information is being displayed.

Size: This represents the total size of memory used by the process in pages. If it's 0, indicating that the process is not using any memory.

Resident: This represents the number of resident pages in memory. **A resident page is a page that is currently in physical memory**. If it's 0, indicating no memory usage.

Analysis of Program - 2

Share: This represents the number of shared pages in memory. **Shared pages are memory pages that are shared among multiple processes**.

Text: This represents the number of pages dedicated to executable code (such as instructions) of the process.

Library: 0 pages - This represents the number of pages used by **shared libraries**. No memory is used for libraries in this case.

Data+Stack: This represents the combined size of data and stack memory used by the process.

Dirty: 0 pages - This represents the number of dirty pages. Dirty pages are those that have been modified since they were last saved to disk or read from disk. It's 0 here, indicating no modified pages.

Applications

The Virtual Memory Analytics Suite can be used in various scenarios:

- System performance analysis and optimization: Identify memory-intensive processes and optimize memory usage to improve overall system performance.
- Software development: Debug memory-related issues such as memory leaks and excessive page faults in software applications.
- Capacity planning: Estimate future memory requirements based on historical data and trends.