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

OPERATING SYSTEMS - CS235AI

REPORT

Submitted by

Shreyashwini R	**1RV22CS192**
Siri H	**1RV22CS198**
Vijayshree M	**1RV22CS232**

Computer Science and Engineering
2023-2024

ABSTRACT

Virtual Memory Profiler (VMP) is a software tool designed to analyze and optimize memory usage in virtualized environments. With the increasing prevalence of virtualization technologies in modern computing infrastructures, efficient memory management becomes crucial for maintaining system performance and scalability.

VMP offers comprehensive insights into the virtual memory utilization patterns of applications running within virtualized environments. By monitoring memory allocation, page swapping, and resource contention, VMP identifies potential bottlenecks and inefficiencies in memory usage. Additionally, VMP provides visualization tools to help users understand memory consumption trends over time and across different virtual machines.

Key features of VMP include real-time monitoring, historical data analysis, and customizable alerting mechanisms. Users can set thresholds for various memory metrics and receive notifications when memory usage exceeds predefined limits or exhibits unusual behavior. Moreover, VMP integrates with existing virtualization platforms, allowing seamless deployment and integration into virtualized environments.

In summary, Virtual Memory Profiler enables organizations to optimize memory utilization, improve system performance, and reduce resource overhead in virtualized environments. By providing actionable insights and monitoring capabilities, VMP empowers users to effectively manage virtual memory resources and ensure the optimal operation of their virtualized infrastructure.

INTRODUCTION

- A Virtual Memory Profiler is a vital tool for analyzing and optimizing the utilization of virtual memory in software. It provides insights into memory usage patterns, potential leaks, and optimization opportunities.
- By offering developers a detailed view of virtual memory management, the profiler enhances program performance, minimizes system crashes, and promotes efficient resource use. Additionally, it serves as an educational resource, helping developers understand the complexities of virtual memory management for creating more robust software systems.
- Creating a memory profiler in C holds significant relevance for software development, offering essential benefits in debugging, optimization, and resource efficiency. Such a tool aids in identifying and rectifying memory-related issues early in the development process, leading to improved program performance and reduced system downtime.
- The project's cross-platform compatibility, educational value, customization options, and potential for open source contribution further enhance its significance.
- Ultimately, this memory profiler contributes to the overall quality and reliability of software, addressing real-world challenges faced by developers and organizations.

PROBLEM STATEMENT

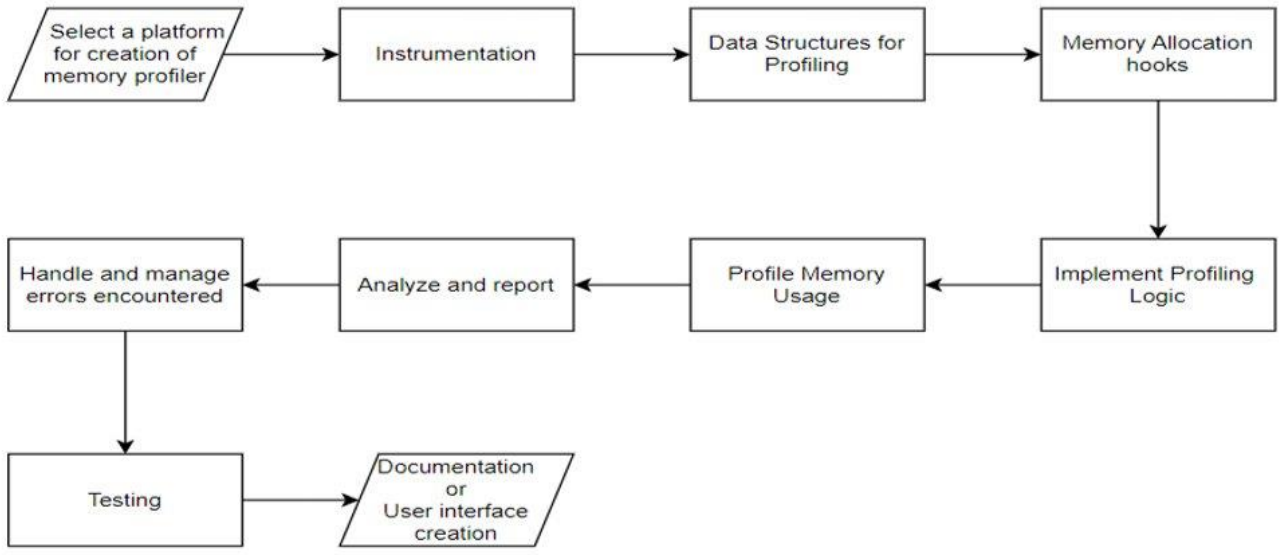
- Develop a Virtual Memory Profiler that can monitor, analyze, and optimize the virtual memory usage of software applications.
- The profiler should be capable of identifying memory leaks, inefficient memory allocation, and excessive page swapping, providing valuable insights for developers and system administrators to enhance the performance of their applications.
- The goal of this project is to develop a Virtual Memory Profiler that can analyze and optimize the utilization of virtual memory in software applications.
- The main aim of this project is to create the virtual memory profiler that generates detailed reports on memory usage patterns.
- Profiling the virtual memory behavior during different stages of application execution.

System Architecture

- **Profiler Interface:** This is the user-facing component where users interact with the profiler. It could be a command-line interface (CLI), a graphical user interface (GUI), or a web-based interface.
- **Profiler Engine:** This is the core of the profiler system responsible for coordinating the profiling process. It manages the interaction with the virtualized environment, collects memory usage data, and controls the profiling workflow.
- **Virtual Machine Monitor (VMM):** In a virtualized environment, the VMM sits between the operating systems and the underlying hardware. It manages virtual machine (VM) resources, including memory allocation and management. The profiler needs to interface with the VMM to gather memory usage information for each VM.
- **Profiling Agent:** This component resides within the virtual machines being profiled. It collects detailed memory usage statistics from within each VM, including information about memory allocation, deallocation, usage patterns, etc. The profiling agent communicates this information back to the profiler engine.
- **Data Storage:** The profiler needs a mechanism to store the collected memory usage data for analysis and visualization. This could be a database, a file system, or any other suitable storage solution.
- **Analysis and Visualization Module:** This module processes the collected memory usage data to generate insights and visualizations. It may include algorithms for identifying memory leaks, inefficient memory usage patterns, trends over time, etc. Visualization tools can be used to present this information to users in a meaningful way, such as charts, graphs, and reports.
- **Reporting and Alerting:** The profiler may include functionality to generate reports summarizing the profiling results. It could also provide alerting mechanisms to notify users of critical issues detected during profiling, such as excessive memory usage or potential memory leaks.

- **Configuration and Management:** This component allows users to configure profiling parameters, manage profiling sessions, and adjust settings as needed. It provides an interface for specifying which VMs to profile, what metrics to collect, how often to sample data, etc. Security: Security measures should be implemented throughout the system architecture to ensure that sensitive data is protected, access controls are enforced, and the profiler does not introduce vulnerabilities into the virtualized environment. Integration Interfaces: Depending on the specific requirements of the project, the profiler may need to integrate with other systems or tools, such as virtualization management platforms, continuous integration/continuous deployment (CI/CD) pipelines, or issue tracking systems.

Methodology



Advantages of Virtual memory Profiler

- **Identification of Memory Leaks:** Virtual Memory Profilers can detect memory leaks in applications running within virtualized environments. By monitoring memory allocation and deallocation patterns, they can identify areas where memory is allocated but not properly released, helping developers pinpoint and fix memory leak issues.

- **Optimization of Memory Usage:** These profilers can provide insights into memory consumption patterns, allowing developers to optimize memory usage and improve application performance. By identifying inefficient memory allocation practices or excessive memory usage, developers can refactor code to reduce memory overhead and enhance system responsiveness.

- **Detection of Resource Contention**: Virtual Memory Profilers can identify instances of resource contention, where multiple processes or virtual machines compete for limited memory resources. By monitoring memory access patterns and page swapping activity, they can highlight areas of contention and help administrators allocate resources more effectively to prevent performance degradation.

- **Real-time Monitoring and Alerting:** Many Virtual Memory Profilers offer real-time monitoring capabilities, allowing administrators to track memory usage metrics and receive alerts when memory usage exceeds predefined thresholds or exhibits abnormal behavior. This proactive approach enables administrators to address memory-related issues promptly and prevent system downtime or performance degradation.

- **Historical Analysis and Trend Identification:** Virtual Memory Profilers often include features for historical data analysis, allowing users to track memory usage trends over time and identify long-term patterns or anomalies. By analyzing historical data, administrators can gain insights into application behavior and make informed decisions regarding resource allocation and capacity planning.

TOOLS/API’s used

1. System Calls fopen: Opens a file descriptor for the /proc/{pid}/status file.
2. fclose: Closes the file descriptor.
3. getrusage: Retrieves resource usage information for the current process.
4. getrlimit: Gets the current or maximum value for a specific resource limit.
5. The getrusage system call retrieves resource usage information for the current process (itself). This includes peak heap memory usage.
6. The getrlimit system call retrieves the current limit for the stack size of the process.

Functions:

The sscanf function scans specific lines for desired information:

- VmSize: %ld kB: Extracts the virtual memory size of the process.
- VmRSS: %ld kB: Extracts the resident set size (physical memory usage) of the process.
- majflt: %d: Extracts the number of major page faults incurred by the process.

Struct rusage is a structure used to represent resource usage statistics.
It contains various fields, including ru_minflt and ru_majflt, which represent the number of minor and major page faults.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include <sys/resource.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <pid>\n", argv[0]);
        return 1;
    }

    int pid = atoi(argv[1]);

    // Construct `/proc/{pid}/status` path
    char proc_status_file[32];
    sprintf(proc_status_file, "/proc/%d/status", pid);

    // Open the `status` file
    FILE *fp = fopen(proc_status_file, "r");
    if (!fp) {
        perror("fopen");
        return 1;
    }

    // Read lines and extract memory information
    char line[128];
    long vmsize = 0, rss = 0;
    while (fgets(line, sizeof(line), fp)) {
        if (sscanf(line, "VmSize: %ld kB", &vmsize) == 1) {
            break;
        } else if (sscanf(line, "VmRSS: %ld kB", &rss) == 1) {
            break;
        }
    }
    int page_faults = 0;
    char line1[128]; // Buffer for reading file lines
    while (fgets(line, sizeof(line1), fp) != NULL) {
        if (strstr(line, "majflt:") != NULL) {
            // Extract major page faults
            sscanf(line, "majflt: %d", &page_faults);
            break;
        }
```

```
    fclose(fp);

    // Print results
    printf("Virtual memory size: %ld kB\n", vmsize);
    struct rusage usage;
    int err = getrusage(RUSAGE_SELF, &usage);
    if (err == -1) {
        perror("getrusage");
        return 1;
    }

    long heap_max_kb = usage.ru_maxrss * 1024 / 1024;
    struct rlimit stack_limit;
    if (getrlimit(RLIMIT_STACK, &stack_limit) == -1) {
    perror("getrlimit");
    return 1;
}

long stack_max_kb = stack_limit.rlim_cur * 1024 / 1024;

    printf("PID: %d\n", pid);
    printf("Peak Heap Usage: %ld kB\n", heap_max_kb);
    printf("Peak Stack Usage: %ld kB\n", stack_max_kb);
    if (page_faults == 0) {
        fprintf(stderr, "Could not find page fault information for PID %d\n", pid);
        return 1;
    }

    printf("PID: %d\n", pid);
    printf("Estimated Page Faults: %d\n", page_faults);


    return 0;
}
```

Page fault:

Minor Page Fault

It occurs when a process doesn't have a logical mapping to a page, yet the page is present in a frame in RAM.

Major Page Fault

This fault occurs when a page is referenced and it is not present in the RAM, and this is worse because now to get the page to the logical address space will cause heavy penalty in performance of system.

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/resource.h>
#include <sys/types.h>
#include <sys/wait.h>

void print_page_faults(pid_t pid) {
    struct rusage usage;
    if (getrusage(RUSAGE_CHILDREN, &usage) != 0) {
        perror("getrusage");
        return;
    }

    printf("Page faults (minor): %ld\n", usage.ru_minflt);
    printf("Page faults (major): %ld\n", usage.ru_majflt);
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <command> [args...]\n", argv[0]);
        return EXIT_FAILURE;
    }

    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        return EXIT_FAILURE;
    } else if (pid == 0) {
        // Child process: execute the command
        execvp(argv[1], &argv[1]);
        // If execvp returns, an error occurred
        perror("execvp");
        return EXIT_FAILURE;
    } else {
        // Parent process: wait for the child to finish
        int status;
        waitpid(pid, &status, 0);

        if (WIFEXITED(status)) {
            printf("Child process exited with status: %d\n", WEXITSTATUS(status));
        } else {
            printf("Child process terminated abnormally\n");
        }
```

```
        // Print page faults
        print_page_faults(pid);

        return EXIT_SUCCESS;
    }
}
```

OUTPUT

```
siri@Ubuntu:~$ gcc -o memwatch memwatch.c -lrt
siri@Ubuntu:~$ ./memwatch 1403
Virtual memory size: 3760576 kB
PID: 1403
Peak Heap Usage: 2112 kB
Peak Stack Usage: 8388608 kB
Error: Could not find page fault information for PID 1403
```

```
siri@Ubuntu:~$ gcc -o pagefault pagefault.c
siri@Ubuntu:~$ ./pagefault ls
3child.c    fork.c              memory_tracker.c   Pictures
a.out       ls.c                memory_tracker.so  prg1.c
app         mem2.c              memwatch           Public
app.c       mem.c               memwatch.c         snap
Desktop     memory_profiler     Music              target_program
Documents   memory_profiler.c   pagefault          target_program.c
Downloads   memory_profiler.so  pagefault.c        targetprogram.c
Child process exited with status: 0
Page faults (minor): 114
Page faults (major): 3
```

Memory Mapping Information

Provides information about memory-mapped files and shared memory regions used by the process. This can provide insights into how the process interacts with external data sources and other processes.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>


void check_memory_maps(pid_t pid) {
    char command[128];
    FILE *fp;

    // Construct the command to read the process's memory maps
    sprintf(command, "cat /proc/%d/maps", pid);
    fp = popen(command, "r");
    if (!fp) {
        perror("popen");
        exit(1);
    }

    char line[256];
    while (fgets(line, sizeof(line), fp)) {
        // Print the memory mapping information
        printf("Memory Mapping Information (PID %d):\n%s", pid, line);
    }
    pclose(fp);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <pid>\n", argv[0]);
        return 1;
    }
    pid_t pid = atoi(argv[1]);
    check_memory_maps(pid);

    return 0;
}
```

OUTPUT:

```
siri@Ubuntu:~$ gcc -o mm mem_map.c
siri@Ubuntu:~$ ./mm 2369
Memory Mapping Information (PID 2369):
279e0d00000-279e0e00000 rw-p 00000000 00:00 0
Memory Mapping Information (PID 2369):
29a16e00000-29a16f00000 rw-p 00000000 00:00 0
Memory Mapping Information (PID 2369):
3aa45e00000-3aa45f00000 rw-p 00000000 00:00 0
Memory Mapping Information (PID 2369):
57f6fd00000-57f6fe00000 rw-p 00000000 00:00 0
Memory Mapping Information (PID 2369):
63d87500000-63d87600000 rw-p 00000000 00:00 0
```

This output specifies memory mapping information for the process with PID 1429.

- 7fd21772f000-7fd21773a000: This is the memory address range of the memory mapping. It specifies the start and end addresses of the mapped memory region.
- r--p, r-xp: These are the permissions of the memory mapping. The first field indicates read-only permissions (r--p), while the second field indicates readable and executable permissions (r-xp). The permissions are represented using the format {readable}{writable}{executable}{private}.
- 00000000, 0000b000, 00039000: These fields typically contain information about the offset of the memory mapping within the file or device, indicating where the mapping begins relative to the start of the file or device.
- 08:03: These are the major and minor device numbers. They indicate the device that backs the memory mapping.
- /usr/lib/x86_64-linux-gnu/libdbus-1.so.3.19.13: This is the file or device associated with the memory mapping. In this case, it specifies the path to the shared object file libdbus-1.so.3.19.13, which is being memory-mapped into the process's address space.
- This information helps understand how the process interacts with external data sources, such as shared libraries or memory-mapped files.

Different system calls and functions used:

sprintf: This function formats and stores a series of characters and values in the command array, constructing the command to read the process's memory maps. It's used to generate the command string.

popen: This function opens a pipe to execute a shell command (cat /proc/{pid}/maps) and returns a stream that can be used to read the output of the command. It's used to execute the command and read the memory mapping information.

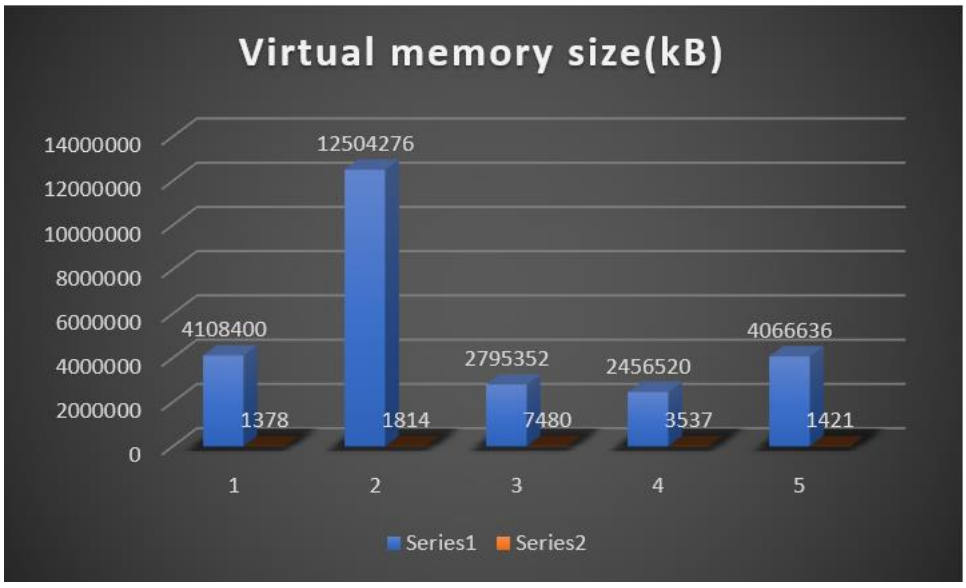
fgets: This function reads a line from the stream (fp) and stores it into the line buffer. It's used to read the output of the command line by line, representing memory mapping information.

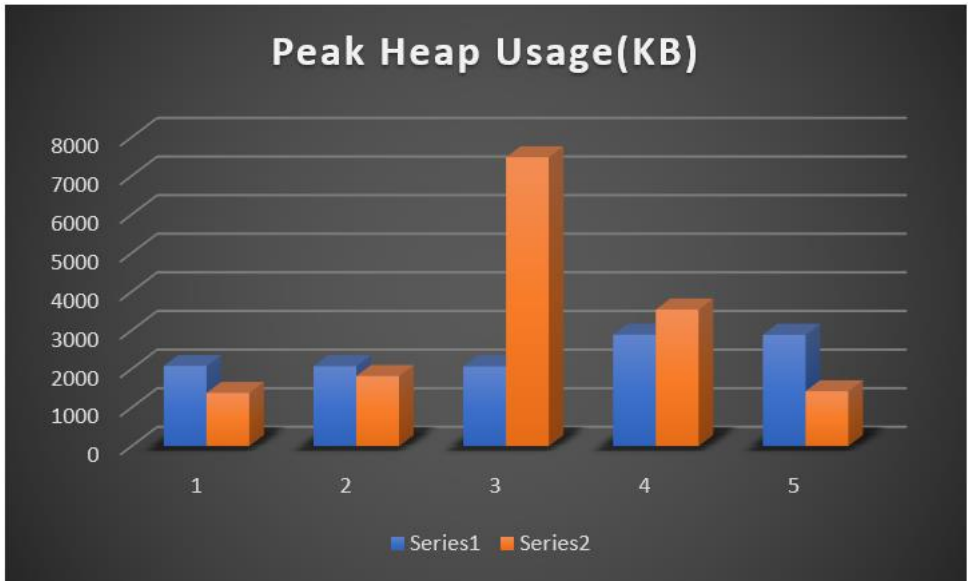
printf: This function prints formatted output to the standard output. It's used to print memory mapping information to the console.

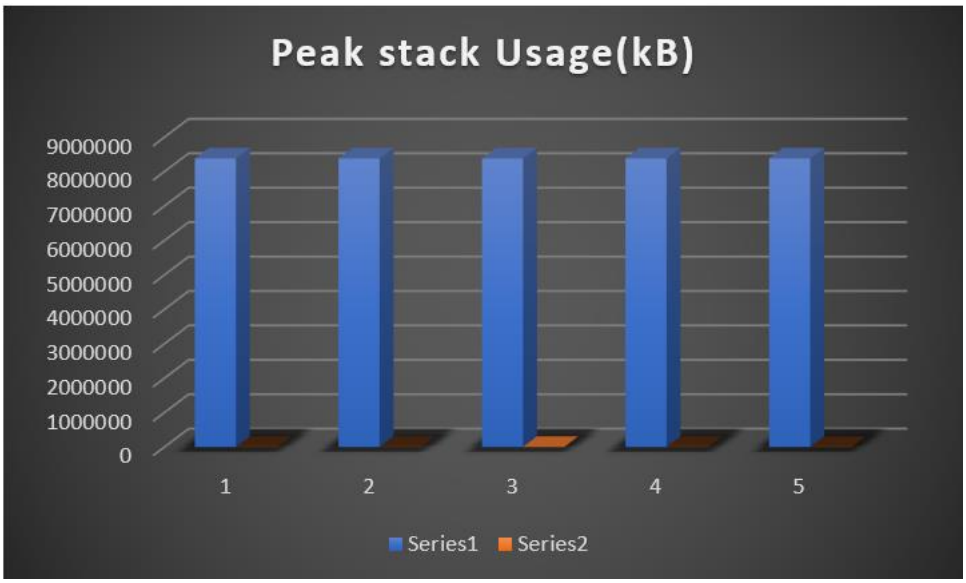
pclose: This function closes the pipe (fp) associated with the command, ensuring that the command execution is complete and releasing any resources associated with it.

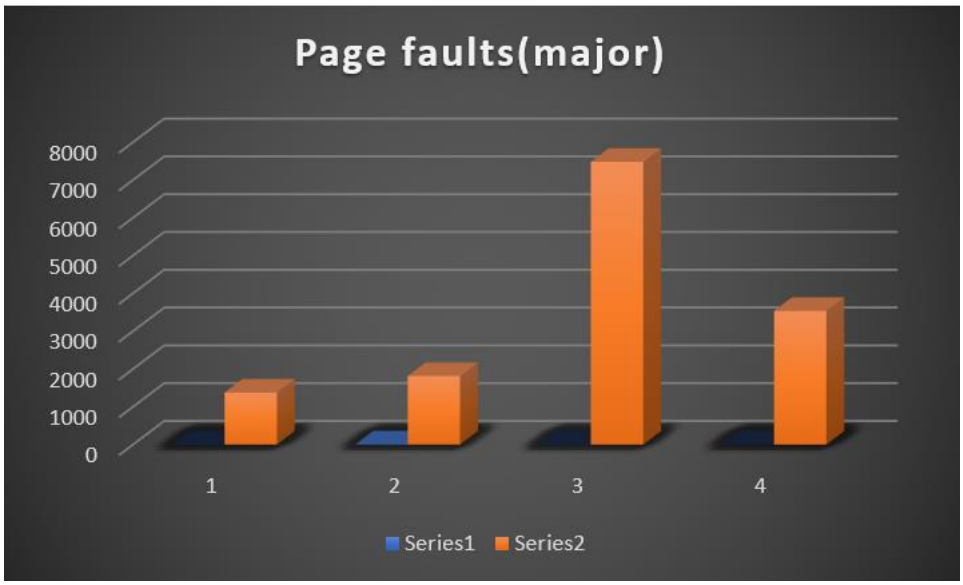
atoi: This function converts a string argument (argv[1]) to an integer (pid_t pid). It's used to extract the process ID from the command-line argument

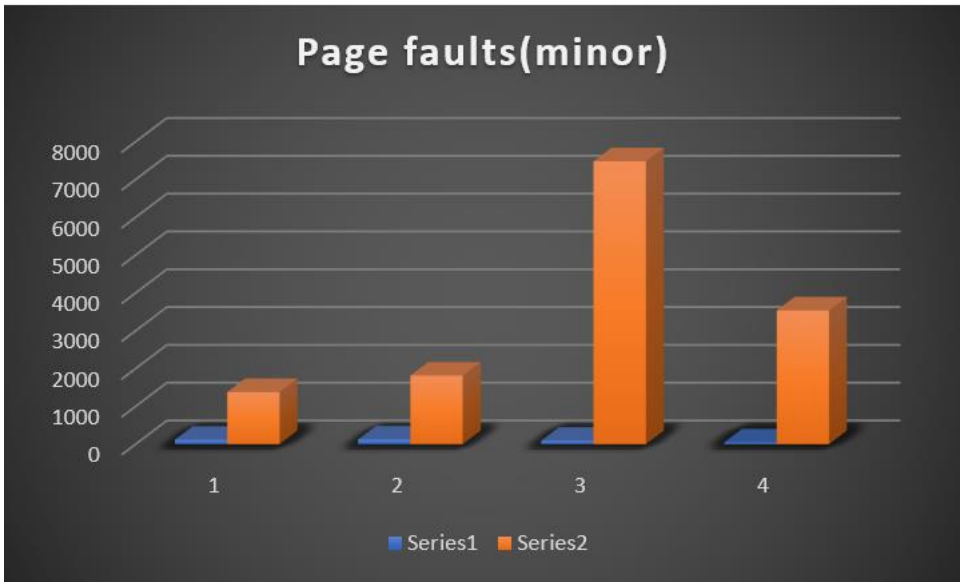
User Dashboard




Peak Heap Usage(KB)
8000
7000
6000
5000
4000
3000
2000
1000
0
1
2
3
4
5
Series1
Series2


Peak stack Usage(kB)
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
1
2
3
4
5
Series1
Series2


Page faults(major)
8000
7000
6000
5000
4000
3000
2000
1000
0
1
2
3
4
Series1
Series2



CONCLUSION

In conclusion, a Virtual Memory Profiler (VMP) is a valuable tool for analyzing and optimizing memory usage within virtualized environments. By providing insights into memory allocation, page swapping, resource contention, and memory access patterns, VMP empowers users to identify inefficiencies, detect memory-related issues such as leaks or contention, and improve overall system performance. Through real-time monitoring, historical analysis, and customizable alerting mechanisms, VMP enables proactive management of memory resources, helping to prevent downtime, enhance application responsiveness, and optimize resource utilization. With its ability to integrate with virtualization platforms and provide customizable reporting and visualization features, VMP offers a comprehensive solution for effectively managing memory in virtualized environments, ultimately contributing to the stability, scalability, and efficiency of modern computing infrastructures.