Siddaganga Institute of Technology, Tumakuru-03

(An Autonomous Institute, Affiliated to Visvesvaraya Technological University Belagavi, Approved by AICTE, New Delhi, Accredited by NAAC and ISO 9001:2015 certified)

B.H. road, Tumkuru 572103, Karnataka, India



“Tools To Capture Dynamic Memory Consumption of Application”

-Memory Management Dashboard

*Submitted in the partial fulfillment of the requirements for II year Bachelor of Engineering in Artificial Intelligence & Data Science*

*By;*

## Suprith T S – 1SI21AD050

## Akash L – 1SI21AD007

## Atchutha Rao T A – 1SI21AD064

## *Submiting to:*

**Dr.N.R.Sunitha**

HOD, department of CSE

**Department of Artificial Intelligence & Data Science**

**Academic Year: 2022-23**

**INTRODUCTION**

Tool to capture Dynamic Memory Consumption:

Effective memory management is a cornerstone of software development, directly impacting application performance and system stability. As software becomes increasingly sophisticated and resource-intensive, the ability to capture and analyze dynamic memory consumption becomes paramount. Tools designed to capture this dynamic memory usage provide invaluable insights into how programs allocate, utilize, and release memory resources over time, helping developers optimize their applications for efficiency and reliability.

Dynamic memory consumption tools serve as essential aids for both developers and system administrators, bridging the gap between software and hardware. These tools enable the comprehensive tracking of memory allocation and deallocation patterns, highlighting potential memory leaks, inefficient memory management practices, and the resulting impact on system responsiveness and stability. Armed with this information, developers can identify and rectify issues that could lead to poor performance or crashes, ultimately enhancing user experience.

In this rapidly evolving landscape, developers have a rich array of tools at their disposal, catering to various programming languages, runtime environments, and platforms. For instance, tools like Valgrind, a powerful instrumentation framework, offer real-time monitoring and detection of memory issues, such as memory leaks and illegal memory access, within C and C++ applications. Additionally, AddressSanitizer, a feature of the Clang compiler, identifies memory-related bugs during compile time and runtime, enhancing the quality of code by exposing issues early in the development process.

IDEs (Integrated Development Environments) also play a pivotal role in dynamic memory management. Visual Studio's Memory Profiler, JetBrains' dotMemory, and Eclipse MAT (Memory Analyzer Tool) are examples of tools that integrate directly into the development workflow. They provide detailed visualizations of memory usage patterns, allowing developers to identify memory hogs, detect reference cycles, and optimize memory-intensive parts of the codebase.

For specific languages, runtime environments, and platforms, specialized tools abound. For instance, Node.js developers can leverage Node.js's built-in memory profiler to capture heap snapshots and analyze memory usage trends. Java developers can use tools like VisualVM or YourKit Java Profiler to monitor memory consumption and performance of Java applications in real-time.

In our exploration of tools to capture dynamic memory consumption, we will delve into the diverse landscape of available tools, showcasing their distinctive features, benefits, and real-world applications. Understanding the significance of monitoring dynamic memory consumption equips developers with the means to create software that excels in both efficiency and reliability, meeting the demanding expectations of modern computing environments.

Tools designed to capture dynamic memory consumption operate by closely monitoring and analyzing memory allocation and deallocation activities within software applications. These tools provide insights into how memory is managed, utilized, and released during the execution of programs. Here's a general overview of how these tools work:

**1. Instrumentation and Profiling:**

Many memory consumption tools use instrumentation techniques to insert monitoring code into the application's source code, bytecode, or binary executable. This code tracks memory-related events, such as memory allocations, deallocations, and function calls. Profiling is a common technique where the tool collects runtime data to create a detailed memory usage profile.

**2. Memory Tracking:**

As the application runs, the memory consumption tool records each memory allocation and deallocation event, along with associated metadata like memory addresses, sizes, and timestamps. This tracking enables the tool to build a comprehensive picture of memory usage patterns over time.

**3. Memory Leak Detection:**

Tools identify memory leaks by monitoring memory allocations that are not properly deallocated. They keep track of allocated memory blocks and their corresponding references. If a memory block becomes inaccessible due to missing deallocation calls, the tool flags it as a potential memory leak.

**4. Memory Access Monitoring:**

Some tools can monitor memory access, detecting illegal memory reads or writes. This is crucial for identifying buffer overflows, null pointer dereferences, and other memory-related bugs that can lead to crashes or security vulnerabilities.

**5. Heap Analysis and Visualization:**

Many tools provide visualizations of the application's memory usage, typically in the form of graphs, charts, or heatmaps. These visual representations help developers identify memory hotspots, fragmentation, and trends in memory consumption.

**6. Statistical Analysis:**

Tools often collect statistical data about memory usage, such as the total memory allocated, peak memory usage, average memory consumption, and more. This information aids in understanding the overall memory behavior of the application.

**7. Real-time Monitoring:**

Some tools offer real-time monitoring, allowing developers to observe memory consumption as the application runs. This can be crucial for identifying sudden spikes in memory usage or gradual memory leaks.

**8. Reporting and Alerts:**

When memory issues are detected, tools generate reports, logs, or alerts that highlight the problematic areas in the code. These reports guide developers to the precise location of memory-related problems, facilitating their resolution.

**9. Integration with Development Environments:**

Many tools integrate seamlessly with IDEs or other development tools. They provide interactive features and offer guidance on memory optimizations.

**10. Guided Debugging:**

Some advanced tools provide guided debugging by identifying memory-related issues and suggesting potential solutions. They can guide developers through the process of fixing memory problems step by step.

In summary, tools to capture dynamic memory consumption work by closely monitoring memory-related activities, analyzing patterns, and providing insights into memory usage behaviors. This empowers developers to detect and address memory-related issues, optimize memory usage, and enhance the overall quality and performance of software applications.

Memory Management Dashboard:

A Memory Management Dashboard is a specialized tool designed to monitor, analyze, and optimize memory usage within software applications and systems. It provides developers, system administrators, and performance engineers with valuable insights into how memory resources are allocated, utilized, and released. This proactive approach to memory management enhances software performance, prevents memory-related issues, and ensures efficient resource utilization.

**Functionality:**

A Memory Management Dashboard offers a range of features to facilitate effective memory management:

1. Real-time Monitoring: The dashboard continuously tracks memory usage patterns in real time, allowing users to observe how memory is being allocated and utilized as the application runs.
2. Memory Allocation Analysis: Developers can identify instances of excessive memory allocation, detect memory leaks (unreleased memory), and pinpoint potential memory fragmentation issues.
3. Memory Usage Visualization: The dashboard displays memory consumption trends using charts, graphs, and heatmaps, enabling users to quickly identify memory-intensive areas.
4. Memory Leak Detection: By tracking allocated memory blocks and their references, the dashboard can detect memory leaks, helping developers find and fix these issues before they impact system stability.
5. Resource Optimization: Users can identify opportunities to optimize memory usage, reduce memory fragmentation, and improve overall application performance.
6. Alerting Mechanisms: The dashboard can be configured to send alerts or notifications when memory consumption crosses predefined thresholds, allowing for timely intervention.

**Advantages:**

1. Early Issue Detection: Memory Management Dashboards enable the early detection of memory-related issues, such as memory leaks or excessive memory consumption, before they lead to system crashes or performance degradation.
2. Performance Optimization: By identifying memory-hungry portions of code, developers can optimize memory usage, leading to more responsive and efficient software.
3. Enhanced Stability: Effective memory management reduces the risk of out-of-memory errors and application crashes due to memory-related problems.
4. Resource Efficiency: Optimizing memory usage ensures efficient utilization of system resources, allowing for better scalability and reduced hardware requirements.
5. Cost Savings: Effective memory management leads to improved performance, which can reduce the need for hardware upgrades and ultimately lower costs.

**Applications and Uses:**

1. Web Applications: Memory Management Dashboards are used to monitor the memory usage of web applications to ensure consistent performance and prevent memory leaks that can impact user experience.
2. Mobile Apps: Developers use these dashboards to monitor memory usage on mobile devices, optimizing app performance and battery consumption.
3. Embedded Systems: In resource-constrained environments like embedded systems, monitoring memory usage is crucial to avoid memory-related crashes and failures.
4. Gaming: Memory Management Dashboards help game developers optimize memory usage, leading to smoother gameplay and improved user experiences.
5. Data Centers: System administrators use memory management tools to monitor and optimize memory usage across servers, ensuring efficient resource allocation in data center environments.
6. Performance Testing: QA teams utilize memory management tools to stress-test applications and identify memory-related bottlenecks before deployment.
7. In essence, a Memory Management Dashboard is a powerful tool that empowers developers and administrators to proactively manage memory resources, optimize software performance, and ensure system stability across various applications and environments.

**The Need for Memory Management Dashboards:**

In dynamic and resource-intensive applications, memory-related issues such as memory leaks, excessive memory usage, and fragmentation can lead to degraded performance and even system crashes. Detecting and addressing these issues manually can be time-consuming and error-prone. Memory Management Dashboards offer a solution by providing real-time visibility into memory consumption, facilitating early issue detection, and enabling proactive optimization.

**Conclusion:**

Memory Management Dashboards play a pivotal role in enhancing software performance, stability, and resource optimization. By providing real-time insights into memory usage, enabling early issue detection, and promoting proactive optimization, these dashboards empower developers and administrators to deliver efficient and reliable applications across a range of domains. The evolution of computing systems demands advanced memory management tools, making Memory Management Dashboards an indispensable component of modern software development and system management strategies.

**Real-time Data Update Module:**

The Real-time Data Update Module is a component in software systems that enables the automatic and immediate display of data changes as they occur. This module ensures that users receive up-to-the-moment information without having to manually refresh or request updates. It is particularly useful in applications where timely insights are critical, such as monitoring dashboards, collaborative platforms, financial systems, and IoT environments.

**Functionality:**

The main purpose of the Real-time Data Update Module is to keep users informed with the latest data in real time. Instead of relying on users to initiate updates, this module automates the process by pushing data changes from the server to the client whenever new information becomes available.

**Applications:**

1. Monitoring Dashboards: The module ensures that monitoring dashboards display real-time data from various sources, such as system performance metrics, network status, or website traffic.
2. Collaborative Platforms: Real-time collaborative editing and communication in applications like Google Docs and messaging platforms rely on this module.
3. IoT Environments: The module plays a crucial role in IoT systems by enabling real-time monitoring and control of devices, such as in smart homes or industrial automation.
4. Financial Systems: In financial platforms, real-time stock market updates, currency exchange rates, and financial news dissemination benefit from this module.
5. Social Media Feeds: Instant updates of social media posts, comments, and notifications are facilitated by real-time data updates.

The Real-time Data Update Module are powerful tool that ensures users have access to the latest information as it becomes available, enhancing user experiences and enabling efficient interactions with dynamic data.

**Tools used:**

**os module:**

The os module in Python provides a way to interact with the operating system. It allows you to execute system-level commands, retrieve environment variables, and perform various system-related tasks. In this code, the os.system() function is used to clear the console screen by executing the appropriate command based on the operating system type (Windows or Unix-like).

**time module:**

The time module provides various time-related functions in Python. It allows you to work with time intervals, sleep for specific durations, and measure time intervals between events. In this code, the time.sleep() function is used to introduce a delay of 2 seconds between memory usage updates in the monitor\_memory\_usage() function.

**psutil library:**

The psutil (process and system utilities) library is an external third-party Python library that provides an interface to retrieve information on system utilization (CPU, memory, disks, network, sensors) and system uptime. It's used to access memory-related information and monitor memory usage of specific applications.

**psutil.virtual\_memory():**

This function retrieves memory usage statistics, including total memory, available memory, used memory, and memory usage percentage. It returns a namedtuple containing various memory-related attributes.

psutil.process\_iter(): This function returns an iterator over all running processes. It is used to retrieve information about running processes, including process names, memory information, and other attributes.

**Explanation:**

The os, time, and psutil tools and libraries collectively enable the code to achieve the following:

Clear the console screen for a clean display using the os.system() function.

Introduce delays between memory usage updates using the time.sleep() function.

Access memory-related information such as total memory and memory usage percentage using psutil.virtual\_memory().

Retrieve process information, including process names and memory information, using psutil.process\_iter().

These tools and libraries enhance the code's functionality by allowing it to interact with the operating system, manage time intervals, and access memory and process-related data. This combination of tools makes it possible to create a basic memory management dashboard that displays memory information and monitors memory usage of specific applications.

**CODE:**

*import* os

*import* time

*import* psutil

def clear\_screen():

    os.system('cls' *if* os.name == 'nt' *else* 'clear')

def display\_dashboard():

    clear\_screen()

    print("Memory Management Dashboard\n")

    print("1. Display Memory Information")

    print("2. Monitor Memory Usage of a Specific Application")

    print("3. Stop a Specific Application")

    print("4. Terminate Highest Memory Consuming Application")

    print("5. Exit")

    choice = input("\nEnter your choice: ")

*return* choice

def display\_memory\_info():

    memory\_info = psutil.virtual\_memory()

    print("\nMemory Information:")

    print(f"Total Memory: {memory\_info.total} bytes")

    print(f"Available Memory: {memory\_info.available} bytes")

    print(f"Used Memory: {memory\_info.used} bytes")

    print(f"Memory Usage Percentage: {memory\_info.percent:.2f}%")

    input("\nPress Enter to continue...")

def monitor\_memory\_usage(*application\_name*):

*try*:

*while* True:

            memory\_info = psutil.virtual\_memory()

            clear\_screen()

            print(

                f"Monitoring Memory Usage of '{*application\_name*}' (Press Ctrl+C to stop)...\n")

            processes = [proc *for* proc *in* psutil.process\_iter(

*attrs*=['pid', 'name', 'memory\_info']) *if* *application\_name* in proc.info['name']]

*if* processes:

                total\_memory = sum(

                    proc.info['memory\_info'].rss *for* proc *in* processes)

                application\_memory = total\_memory

                total\_system\_memory = memory\_info.total

                memory\_percentage = (application\_memory /

                                     total\_system\_memory) \* 100

                print(f"Total System Memory: {total\_system\_memory} bytes")

                print(

                    f"\n'{*application\_name*}' Memory Usage: {application\_memory} bytes")

                print(

                    f"Memory Usage Percentage by '{*application\_name*}' : {memory\_percentage:.2f}%")

*else*:

                print(f"No process found with name '{*application\_name*}'")

            time.sleep(1.5)  *# Update every 2 seconds*

*except* KeyboardInterrupt:

*pass*

def stop\_application(*application\_name*):

*for* proc *in* psutil.process\_iter(*attrs*=['pid', 'name']):

*if* *application\_name*.lower() in proc.info['name'].lower():

*try*:

                process = psutil.Process(proc.info['pid'])

                process.terminate()

                print(

                    f"Process '{*application\_name*}' (PID: {proc.info['pid']}) has been terminated.")

*except* (psutil.NoSuchProcess, psutil.AccessDenied):

*pass*  *# Handle cases where the process is not found or termination is denied*

def get\_user\_applications():

    user\_applications = []

*for* proc *in* psutil.process\_iter(*attrs*=['pid', 'name', 'memory\_info', 'username']):

*if* proc.info['username'] and proc.info['username'] != 'SYSTEM':

            user\_applications.append(proc)

*return* user\_applications

def terminate\_highest\_memory\_application():

*try*:

*while* True:

            user\_applications = get\_user\_applications()

            highest\_memory\_process = max(

                user\_applications, *key*=lambda *proc*: *proc*.info['memory\_info'].rss, *default*=None)

*if* highest\_memory\_process:

                clear\_screen()

                print("Terminating Highest Memory Consuming User Application\n")

                print(

                    f"Application Name: {highest\_memory\_process.info['name']}")

                print(

                    f"Memory Usage: {highest\_memory\_process.info['memory\_info'].rss} bytes")

                choice = input("\nTerminate this application? (y/n): ").lower()

*if* choice == 'y':

                    process = psutil.Process(

                        highest\_memory\_process.info['pid'])

                    process.terminate()

                    print(

                        f"Process '{highest\_memory\_process.info['name']}' (PID: {highest\_memory\_process.info['pid']}) has been terminated.")

*elif* choice == 'n':

                    print("Application termination cancelled.")

*break*  *# Exit the loop and return to normal execution*

*else*:

                    print("Invalid choice. Please select 'y' or 'n'.")

*else*:

                print("No running user applications found.")

            time.sleep(2)  *# Pause for a moment before updating again*

*except* KeyboardInterrupt:

*pass*

def main():

*while* True:

        choice = display\_dashboard()

*if* choice == "1":

            display\_memory\_info()

*elif* choice == "2":

            all\_process\_names = [proc.info['name']

*for* proc *in* psutil.process\_iter(*attrs*=['name'])]

            print("All running process names:", all\_process\_names)

            application\_name = input(

                "\nEnter the name of the application to monitor: ")

            monitor\_memory\_usage(application\_name)

*elif* choice == "3":

            application\_name = input(

                "Enter the name of the application to stop: ")

            stop\_application(application\_name)

*elif* choice == "4":

            terminate\_highest\_memory\_application()

*elif* choice == "5":

            print("Exiting the Memory Management Dashboard...")

*break*

*else*:

            print("Invalid choice. Please select a valid option.")

*if* \_\_name\_\_ == "\_\_main\_\_":

    main()

**Algorithm:**

1. **Function Definitions**:
   * Define the following functions:
     + **clear\_screen()**: Clears the terminal screen.
     + **display\_dashboard()**: Displays the main menu of the memory management dashboard and prompts the user for a choice.
     + **display\_memory\_info()**: Displays system memory information.
     + **monitor\_memory\_usage(application\_name)**: Monitors memory usage of a specific application.
     + **stop\_application(application\_name)**: Attempts to stop a specific application.
     + **get\_user\_applications()**: Fetches a list of user-specific running applications.
     + **terminate\_highest\_memory\_application()**: Monitors and terminates the highest memory-consuming user application.
     + **main()**: The main entry point of the script.
2. **Main Execution**:
   * Start the main execution by checking if the script is run directly (**if \_\_name\_\_ == "\_\_main\_\_":**).
   * Call the **main()** function to start the memory management dashboard loop.
3. **Main Loop (main())**:
   * Enter an infinite loop that displays the dashboard and waits for user input.
   * Depending on the user's choice:
     + If choice is "1": Call **display\_memory\_info()** to show system memory information.
     + If choice is "2": Display all running process names, ask for an application name, and call **monitor\_memory\_usage(application\_name)** to track memory usage.
     + If choice is "3": Ask for an application name, and call **stop\_application(application\_name)** to terminate the application.
     + If choice is "4": Call **terminate\_highest\_memory\_application()** to terminate the highest memory-consuming user application.
     + If choice is "5": Print an exit message and break the loop to exit the dashboard.
     + If choice is invalid: Print an error message.
4. **Display Memory Information (display\_memory\_info())**:
   * Fetch memory information using **psutil.virtual\_memory()**.
   * Display total memory, available memory, used memory, and memory usage percentage.
   * Wait for the user to press Enter before returning.
5. **Monitor Memory Usage (monitor\_memory\_usage(application\_name))**:
   * Use a try-except block to handle interruptions (Ctrl+C).
   * Continuously loop:
     + Fetch system memory information using **psutil.virtual\_memory()**.
     + Clear the screen.
     + Display monitoring information for the specified application:
       - Sum up memory usage of all processes with the provided **application\_name**.
       - Calculate memory usage percentage.
     + If no processes are found with the specified name, display a message.
     + Sleep for a short duration (1.5 seconds) before the next update.
6. **Stop Application (stop\_application(application\_name))**:
   * Iterate through running processes using **psutil.process\_iter(attrs=['pid', 'name'])**.
   * Check if the application name matches any running process name.
   * If found, attempt to terminate the process.
   * Handle exceptions for cases where the process is not found or termination is denied.
7. **Get User Applications (get\_user\_applications())**:
   * Fetch a list of user-specific running applications (processes) based on the **username** attribute.
   * Filter out processes owned by the 'SYSTEM' user.
8. **Terminate Highest Memory Application (terminate\_highest\_memory\_application())**:
   * Use a try-except block to handle interruptions (Ctrl+C).
   * Continuously loop:
     + Fetch a list of user applications using **get\_user\_applications()**.
     + Find the application with the highest memory consumption.
     + If a high-memory process is found:
       - Display its name and memory usage.
       - Ask the user if they want to terminate the process.
       - If yes, terminate the process; if no, break the loop.
       - Handle invalid input.
     + If no user applications are found, display a message.
     + Sleep for a short duration (2 seconds) before the next update.

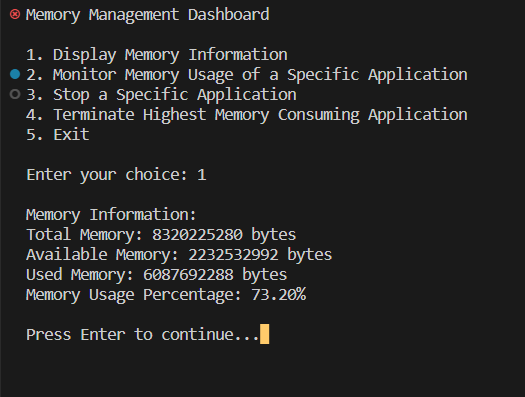
**Description:**

1. **clear\_screen()**:
   * Clears the terminal screen based on the operating system.
   * Creates a clean interface for displaying different sections of the dashboard.
2. **display\_dashboard()**:
   * Displays the main menu of the memory management dashboard.
   * Prompts the user to select a choice and returns the chosen option.
3. **display\_memory\_info()**:
   * Fetches and displays system memory information.
   * Presents data such as total memory, available memory, used memory, and memory usage percentage.
   * Waits for user input before continuing.
4. **monitor\_memory\_usage(application\_name)**:
   * Monitors the memory usage of a specific application.
   * Displays memory usage information continuously.
   * Calculates memory usage percentage of the application in relation to total system memory.
5. **stop\_application(application\_name)**:
   * Attempts to stop a specific application by finding and terminating its process.
   * Handles cases where the process is not found or termination is denied.
6. **get\_user\_applications()**:
   * Retrieves a list of user-specific running applications (processes).
   * Filters out processes owned by the 'SYSTEM' user.
7. **terminate\_highest\_memory\_application()**:
   * Monitors and terminates the highest memory-consuming user application.
   * Displays information about the application with the highest memory consumption.
   * Offers the option to terminate the application or cancel the termination.
8. **main()**:
   * The main execution function that runs the memory management dashboard.
   * Displays the dashboard loop and processes user input based on their choices.
   * Calls various functions based on the user's selected option.
9. **Main Execution** (**if \_\_name\_\_ == "\_\_main\_\_":**):
   * Ensures that the **main()** function is executed only when the script is run directly.
   * The main loop of the memory management dashboard begins.

The code offers a text-based user interface that allows users to:

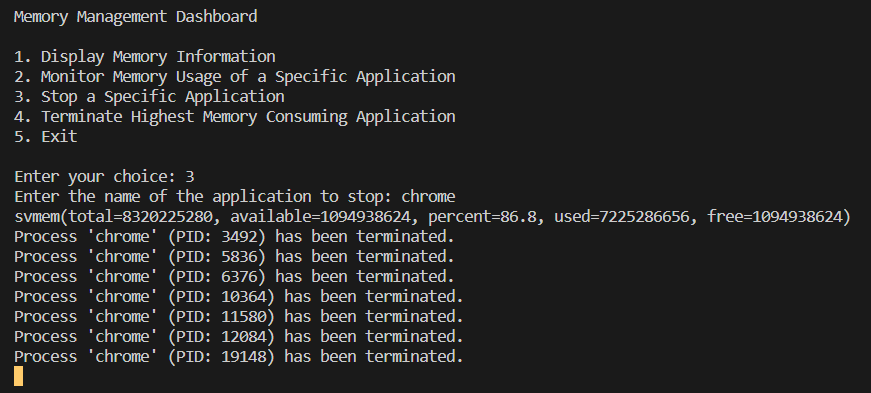
* Display memory information.
* Monitor memory usage of a specific application.
* Stop a specific application.
* Terminate the highest memory-consuming user application.
* Exit the dashboard.

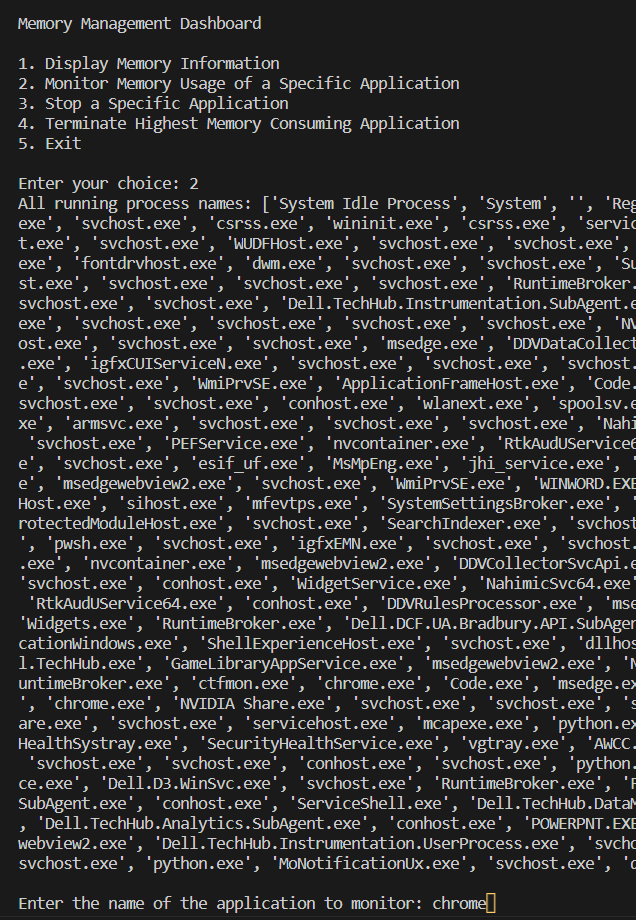
**Results and Test Cases:**



A computer screen shot of a black screen

Description automatically generated





A computer screen shot of a black background

Description automatically generatedA computer screen shot of a black screen

Description automatically generatedA screenshot of a computer

Description automatically generated