Final Project Report Documentation

1. Project Title

The Virtualized Kernel: An Analysis of Scheduling, Containerization, and Resource Management Across Diverse OS Environments.

2. Team Information

- **Team Name: *Avengers***
- **Team Members:**
  - **Priyanshu Joshi(Team Lead) -230112733**
  - **Aviral Atray -211111009**
  - **Devang Sharma -230111521**
  - **Gourab Mohanty- 211111001**

3. Abstract

This project offers a practical, detailed look at important operating system concepts in a virtualized setting. The main goal was to show the "invisible" management layers of modern computing: the OS kernel and the hypervisor. We built a multi-layered virtual lab using a Type 2 hypervisor (VMware Fusion) that hosts different guest operating systems (Ubuntu and Kali Linux).

Our analysis focuses on six key areas: (1) Thread Scheduling, showing how the Linux kernel prioritizes CPU-bound and I/O-bound tasks; (2) Hypervisor Resource Management, illustrating how physical CPU, memory, disk, and network resources are allocated and isolated in real-time; (3) Deadlock Analysis, where we created a circular wait condition and simulated Deadlock Avoidance using the Banker's Algorithm; (4) Kernel Containerization, demonstrating the running of multiple isolated OS environments (Alpine and Debian) on a single shared kernel using Docker; and (5) Inter-Process Communication, achieved through an API cross-call between isolated containers. This project serves as a clear demonstration of the principles driving modern cloud and virtualized infrastructure.

4. System Architecture & Tools

4.1 Infrastructure Hierarchy

Our experimental setup mimics a production virtualization stack:

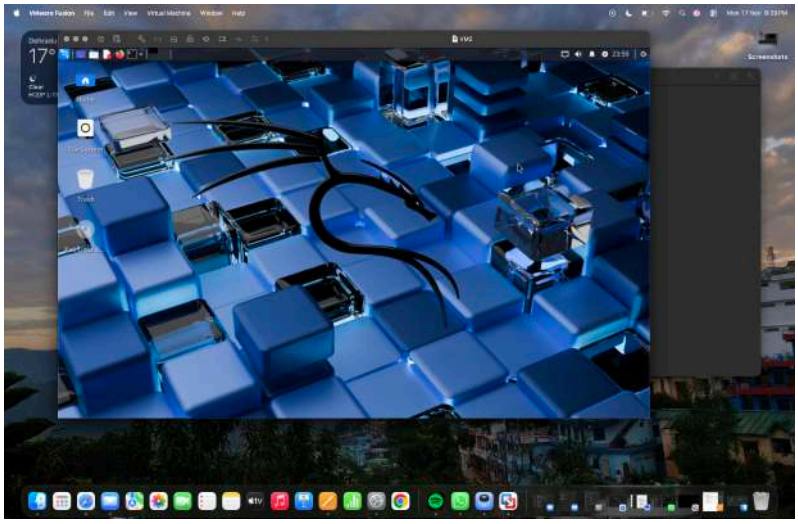
- **Layer 1 (Physical Host):** Apple MacBook (macOS) providing the physical CPU, RAM, and I/O hardware.
- **Layer 2 (Hypervisor): VMware Fusion** (Type 2 Hosted Hypervisor). It acts as the resource manager, partitioning physical hardware into virtual components.
- **Layer 3 (Guest Virtual Machines):**
    - **VM 1: Ubuntu Linux** (General purpose, used for scheduling, containerization, and API tests).
    - **VM 2: Kali Linux** (Specialized security OS, used to demonstrate hypervisor flexibility).
- **Layer 4 (Container Engine): Docker**, running inside VM 1.
- **Layer 5 (Containers):** Alpine Linux and Debian containers running as isolated user-space environments sharing the VM 1 kernel.

VMWare Fusion(setup) in Host Device (Macos)

Two Machines VM21 &VM2

Virtual Machine1 Named As vm1

with Linux Operating System

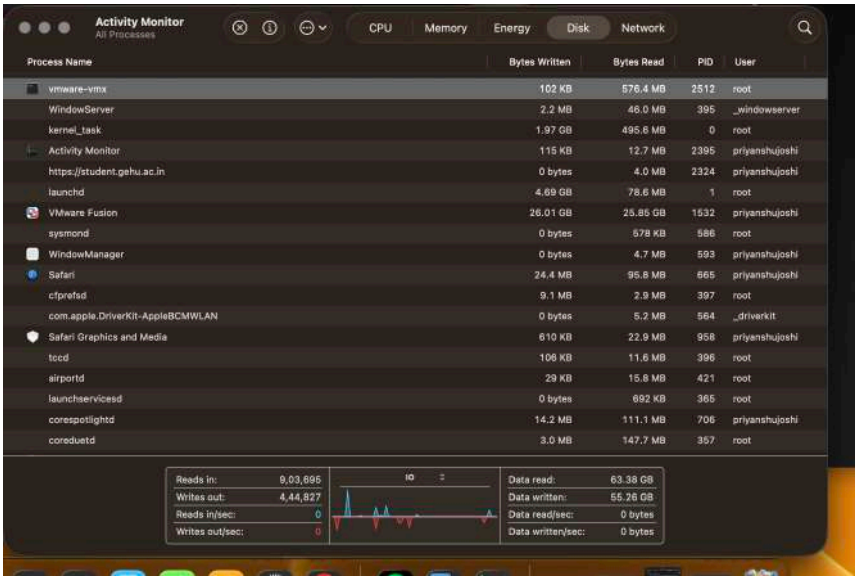


Virtual Machine1 Named As vm2 With Kali Linux Operating System.

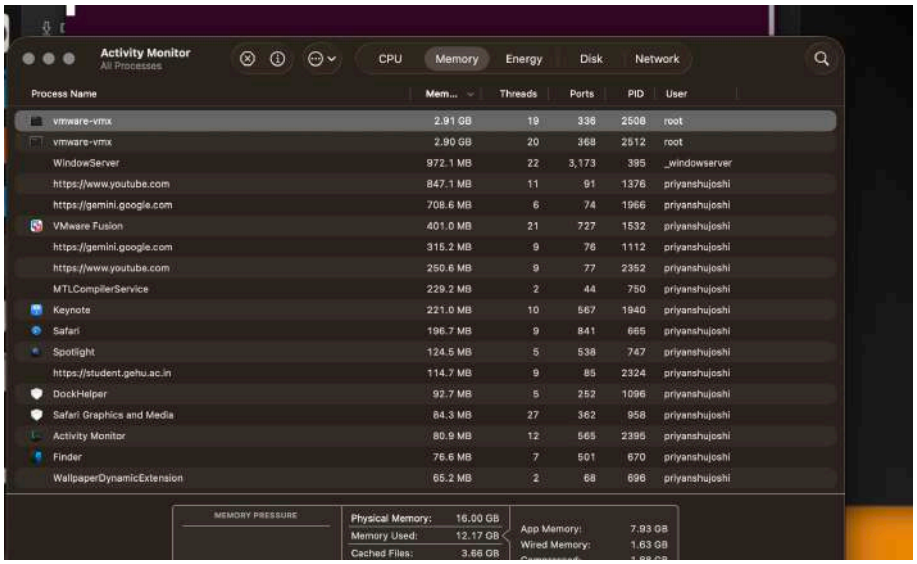
Module 2: Hypervisor Resource Management

- **Objective:** To demonstrate the hypervisor's role as an arbitrator for physical hardware.
- **CPU Contention:** Ran simultaneous CPU loads in VM1 and VM2. **Result:** The hypervisor divided host CPU cycles between the two VMs, proving fair scheduling.
- **Memory Management:** Observed host RAM usage. **Result:** RAM was allocated as a fixed block upon VM boot and fully reclaimed by the host OS upon VM shutdown.
- **Disk Virtualization:** Executed a sustained file creation loop (`dd` command with `sync`). **Result:** Host Activity Monitor showed a matching sustained spike in "Bytes Written," proving the hypervisor was translating virtual write requests to physical disk I/O.
- **Network Virtualization:** Performed a speed test inside the VM. **Result:** Host traffic analysis revealed a dedicated process (`vmnet-natd`) managing the high-bandwidth data transfer, confirming network isolation and NAT traversal.

Pictures Demonstrating Hypervisors Role:



VMware Fusion: This process has written **26.01 GB** and read **25.85 GB**. This Is For Disk Management.



This Image Describes When Both The Virtual Machines Are On .The The Memory Used By Them. This Shows The Memory Management in Hypervisor.

This Image Describing Network Usage when We Started a WiFi Speed Test By Oakla In Vm1.

5. Methodology & Implementation

Module 1: Thread Scheduling Analysis

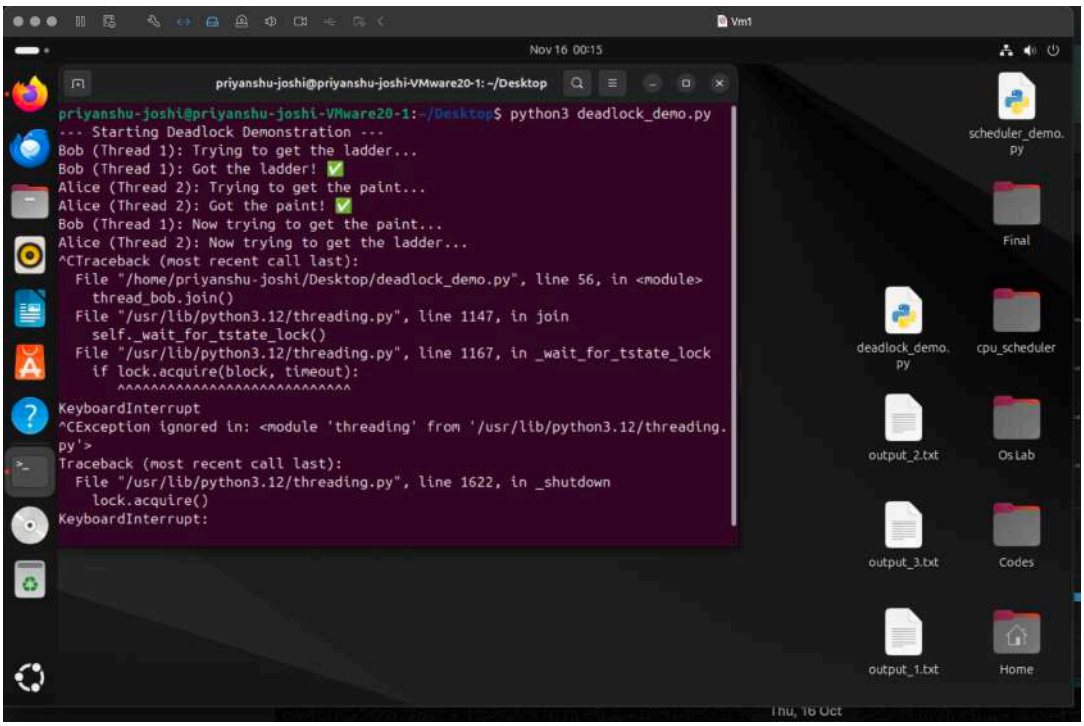
- **Objective:** To visualize the Linux Completely Fair Scheduler (CFS) managing mixed workloads.
- **Implementation:** We developed a Python script (`scheduler_demo.py`) creating multi-threaded workloads: CPU-intensive threads (prime number calculation) and I/O-intensive threads (file writing).
- **Observation:** Using `htop`, we verified that CPU-bound threads were flagged as `R` (Running) with near-100% CPU utilization, while I/O-bound threads were frequently flagged as `S` (Sleeping) with near-0% utilization.
- **Conclusion:** The OS scheduler correctly identified thread behavior and prioritized CPU allocation to active tasks, maximizing processor efficiency.

Module 2: Hypervisor Resource Management

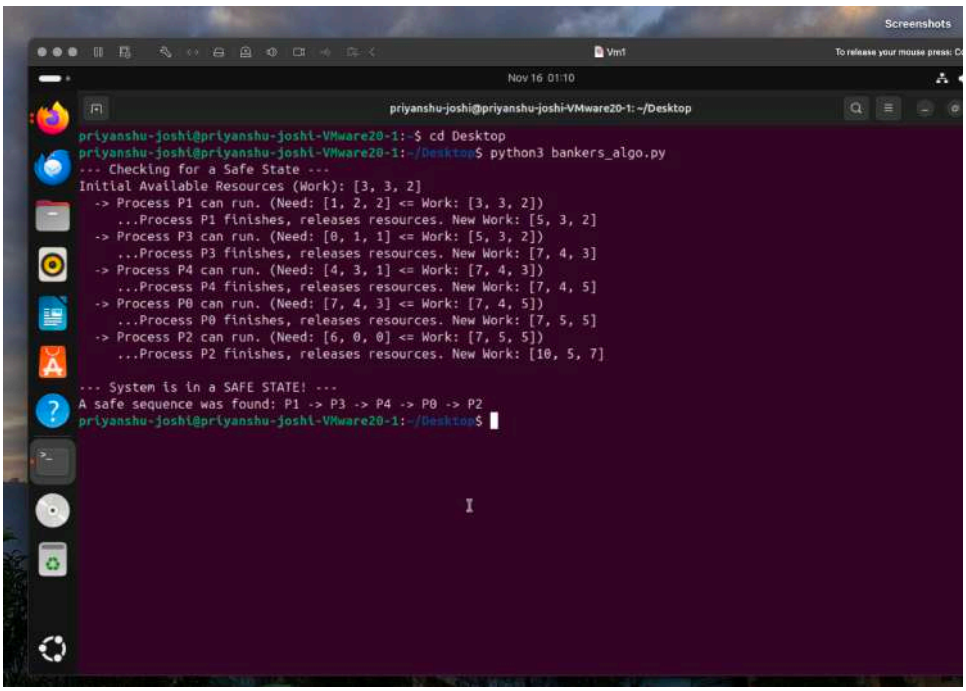
- **Objective:** To demonstrate the hypervisor's role as an arbitrator for physical hardware.
- **CPU Contention:** Ran simultaneous CPU loads in VM1 and VM2. **Result:** The hypervisor divided host CPU cycles between the two VMs, proving fair scheduling.
- **Memory Management:** Observed host RAM usage. **Result:** RAM was allocated as a fixed block upon VM boot and fully reclaimed by the host OS upon VM shutdown.
- **Disk Virtualization:** Executed a sustained file creation loop (`dd` command with `sync`). **Result:** Host Activity Monitor showed a matching sustained spike in "Bytes Written," proving the hypervisor was translating virtual write requests to physical disk I/O.
- **Network Virtualization:** Performed a speed test inside the VM. **Result:** Host traffic analysis revealed a dedicated process (`vmnet-natd`) managing the high-bandwidth data transfer, confirming network isolation and NAT traversal.

Module 3: Deadlock Analysis

- **Deadlock Creation:** We wrote a script (`deadlock_demo.py`) implementing two threads and two locks (Mutexes). By forcing Thread A to wait for Lock B while Thread B held Lock B and waited for Lock A, we successfully created a **Circular Wait**, freezing the program.

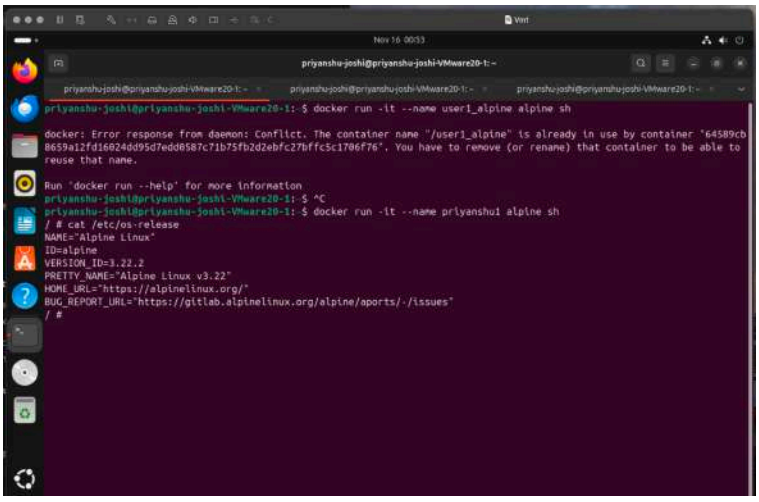


- **Deadlock Avoidance (Banker's Algorithm):** We implemented a simulation (`bankers_algo.py`) defining a set of processes and available resources. The algorithm successfully calculated a "Safe Sequence," proving that the system could allocate resources without entering an unsafe (deadlocked) state.



Module 4: Kernel Containerization

- **Objective:** To demonstrate running multiple OS environments on a single kernel.
- **Implementation:** Inside the Ubuntu VM, we deployed two Docker containers: one running **Alpine Linux** and another running **Debian**.
- **Observation:** Using the host VM's process monitor (`htop`), we observed processes from both the Alpine and Debian environments running side-by-side.
- **Conclusion:** This proved that unlike VMs, which require separate kernels, containers share the host kernel while maintaining user-space isolation.



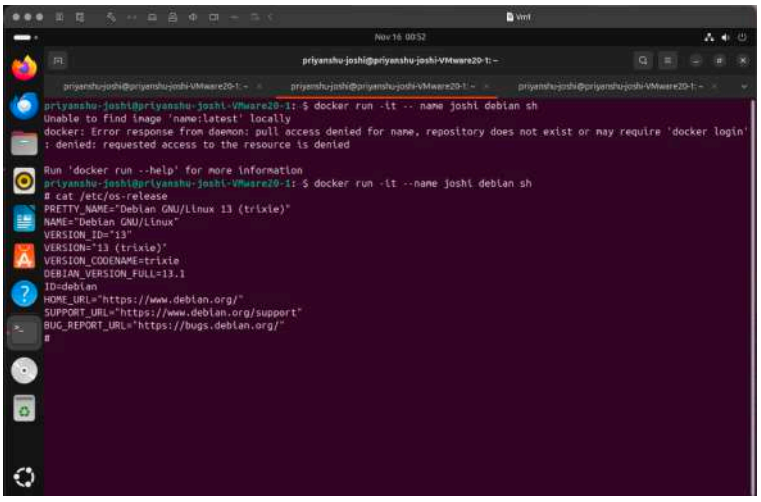
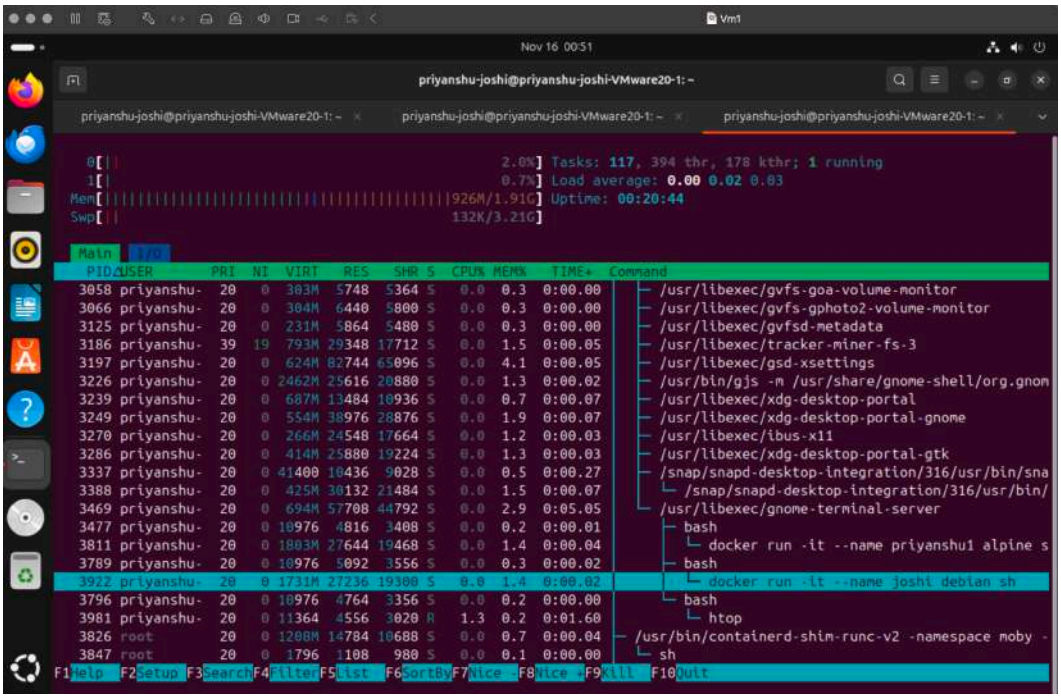


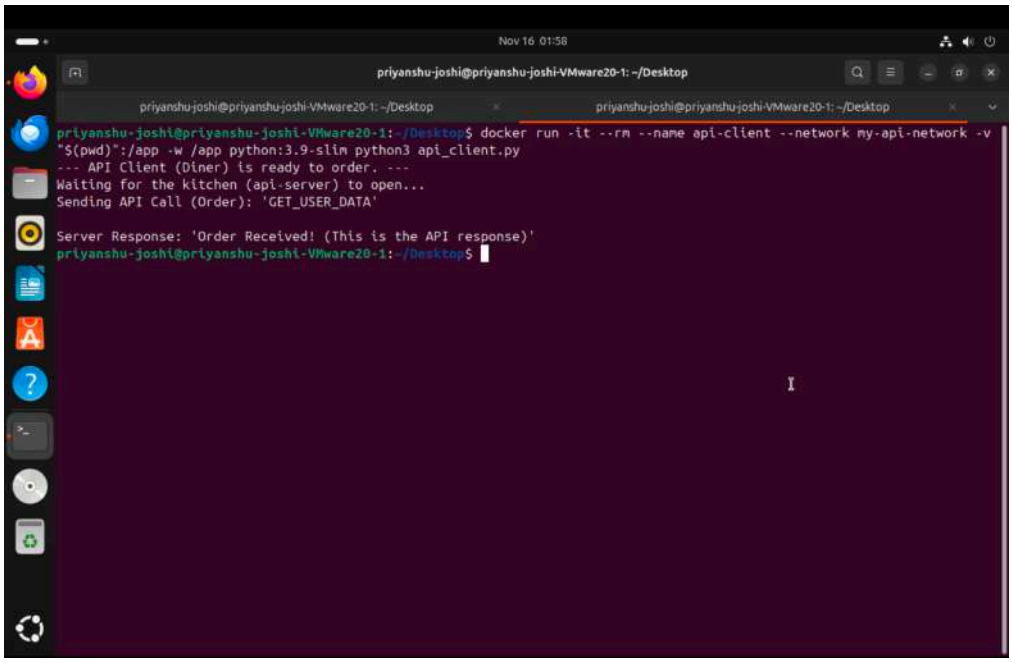
Image1:Alipine Linux. **Image2:Debian Linux**



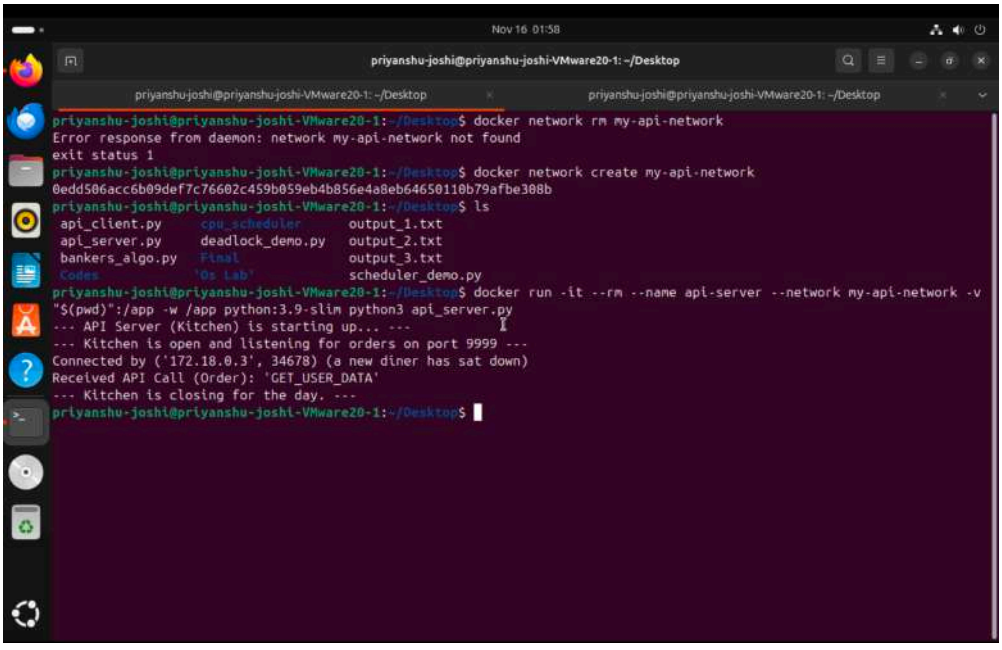
This Image Depicts The Tree View By Using Stop command Showing Both Users Sharing The Same Kernel.

Module 5: API Cross-Call (IPC)

- **Objective:** To demonstrate communication between isolated execution environments.
- **Implementation:** We built a client-server architecture using two separate containers connected via a private Docker network.
    - **Container A (Server):** Listened on Port 9999.
    - **Container B (Client):** Sent a `GET_USER_DATA` request to the server's hostname.
- **Result:** The request successfully crossed the container boundary, was processed by the server, and a response was returned to the client.



Client Side



Server Side

6. Challenges & Solutions

1. **Resource Contention:** Initially, running both VMs caused host instability. **Solution:** We optimized the resource provisioning, limiting each VM to 2 CPU cores and 2GB RAM to ensure the host OS remained responsive.
2. **Disk Caching:** Initial disk write tests showed no activity on the host due to Linux RAM caching. **Solution:** We modified our scripts to use the `sync` command, forcing immediate physical disk writes for accurate observation.
3. **Network Visibility:** Standard monitoring did not attribute network traffic to the main VM process. **Solution:** We identified the specialized `vmnet-natd` background process used by VMware Fusion for network handling, allowing us to correctly measure traffic.

7. Conclusion

This project successfully demystified the complex layers of virtualization. We moved beyond theoretical concepts to provide concrete, visual proof of how operating systems schedule threads and how hypervisors manage the contention for physical hardware. By extending the scope to include containerization and API communication, we demonstrated the evolution of virtualization technology—from heavy, isolated Virtual Machines to lightweight, shared-kernel Containers. The successful implementation of the Banker's Algorithm and Deadlock creation further reinforced our understanding of critical OS safety mechanisms.

8. References / Bibliography

- *Operating System Concepts*, GeeksforGeeks, YouTube.
- *VMware Fusion Documentation*.
- *Docker Official Documentation*.
- *Linux Kernel Scheduling Documentation*.

Submitted To-Ms. Neha Pokhriyal

Submitted By: Team Avengers

B.Tech CSE, 5th Semester