

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

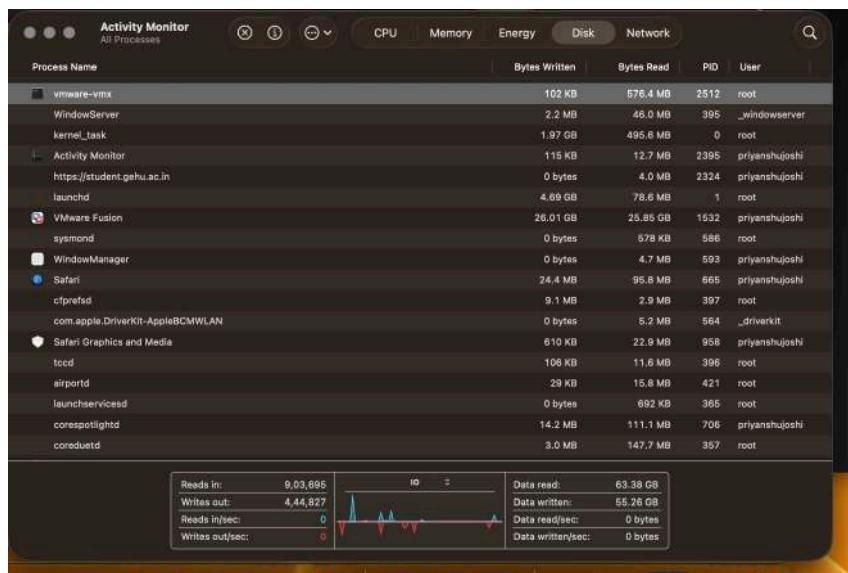


Virtual Machine1 Named As vm2 With Kali Linux Operating System.

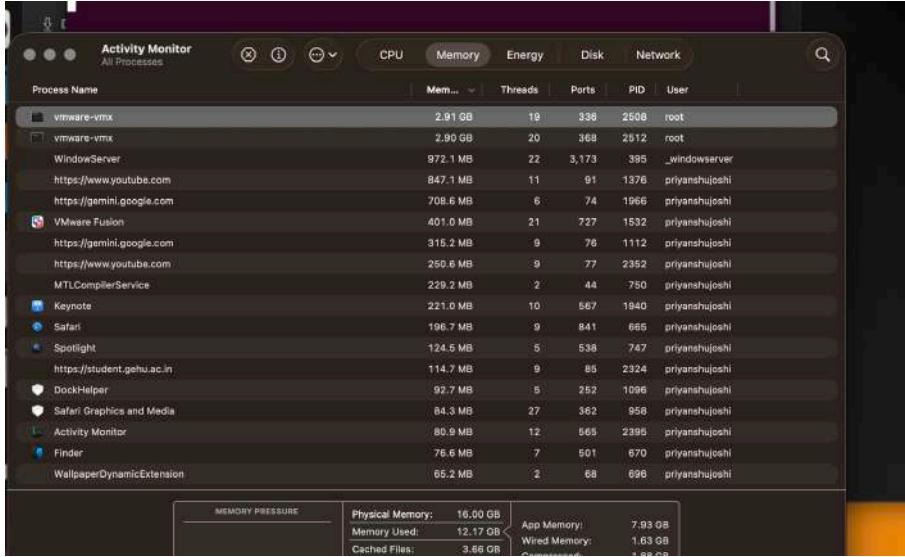
Module 2: Hypervisor Resource Management

- 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.

```

priyanshu-joshi@priyanshu-joshi:~/Desktop$ python3 deadlock_demo.py
... Starting Deadlock Demonstration ...
Bob (Thread 1): Trying to get the ladder...
Bob (Thread 1): Got the ladder! ✓
Alice (Thread 2): Trying to get the paint...
Alice (Thread 2): Got the paint! ✓
Bob (Thread 1): Now trying to get the paint...
Alice (Thread 2): Now trying to get the ladder...
^CTraceback (most recent call last):
  File "/home/priyanshu-joshi/Desktop/deadlock_demo.py", line 56, in <module>
    thread.bob.join()
  File "/usr/lib/python3.12/threading.py", line 1147, in join
    self._wait_for_tstate_lock()
  File "/usr/lib/python3.12/threading.py", line 1167, in _wait_for_tstate_lock
    if lock.acquire(block, timeout):
      ~~~~~~
KeyboardInterrupt
^CException ignored in: <module 'threading' from '/usr/lib/python3.12/threading.py'>
> Traceback (most recent call last):
  File "/usr/lib/python3.12/threading.py", line 1622, in _shutdown
    lock.acquire()
KeyboardInterrupt:

```

- 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.

```

priyanshu-joshi@priyanshu-joshi:~/Desktop$ cd Desktop
priyanshu-joshi@priyanshu-joshi:~/Desktop$ python3 bankers_algo.py
... Checking for a Safe State ...
Initial Available Resources (Work): [3, 3, 2]
-> Process P1 can run. (Need: [1, 2, 2] <= Work: [3, 3, 2])
...Process P1 finishes, releases resources. New Work: [5, 3, 2]
-> Process P3 can run. (Need: [0, 1, 1] <= Work: [5, 3, 2])
...Process P3 finishes, releases resources. New Work: [7, 4, 3]
-> Process P4 can run. (Need: [4, 3, 1] <= Work: [7, 4, 3])
...Process P4 finishes, releases resources. New Work: [7, 4, 5]
-> Process P0 can run. (Need: [7, 4, 3] <= Work: [7, 4, 5])
...Process P0 finishes, releases resources. New Work: [7, 5, 5]
-> Process P2 can run. (Need: [6, 0, 0] <= Work: [7, 5, 5])
...Process P2 finishes, releases resources. New Work: [10, 5, 7]

... System is in a SAFE STATE! ...
A safe sequence was found: P1 -> P3 -> P4 -> P0 -> P2
priyanshu-joshi@priyanshu-joshi:~/Desktop$ 

```

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.

```

Nov 16 00:53
priyanshu-joshi@priyanshu-joshi-VMware20-1:~$ docker run -it --name user1_alpine alpine sh
docker: Error response from daemon: Conflict. The container name "/user1_alpine" is already in use by container "64589cb8659a12fd18924d95dred0587c7b75fb2dzebc27bfcc5c170ef7b". You have to remove (or rename) that container to be able to reuse that name.

Run 'docker run --help' for more information
priyanshu-joshi@priyanshu-joshi-VMware20-1:~$ cat /etc/os-release
PRETTY_NAME="Alpine Linux"
NAME="Alpine Linux"
VERSION_ID=3.22.2
PRETTY_NAME="Alpine Linux v3.22"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://gitlab.alpinelinux.org/alpine/aports/-/issues"
/ #

```

Image1:Alipine Linux.

```

Nov 16 00:52
priyanshu-joshi@priyanshu-joshi-VMware20-1:~$ docker run -it --name joshi debian sh
Unable to find image 'name:latest' locally
docker: Error response from daemon: pull access denied for name, repository does not exist or may require 'docker login'
Run 'docker run --help' for more information
priyanshu-joshi@priyanshu-joshi-VMware20-1:~$ docker run -it --name joshi debian sh
# cat /etc/os-release
PRETTY_NAME="Debian GNU/Linux 13 (trixie)"
NAME="Debian GNU/Linux"
VERSION_ID="13"
VERSION="13 (trixie)"
VERSION_CODENAME=trixie
DEBIAN_VERSION_FULL=13.1
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
# 
```

Image2:Debian Linux

```

Nov 16 00:51
priyanshu-joshi@priyanshu-joshi-VMware20-1:~$ htop
Tasks: 117, 394 thr, 178 kthr; 1 running
0.7% Load average: 0.00 0.02 0.03
Mem: 926M/1.91G Uptime: 00:20:44
Swap: 132K/3.21G

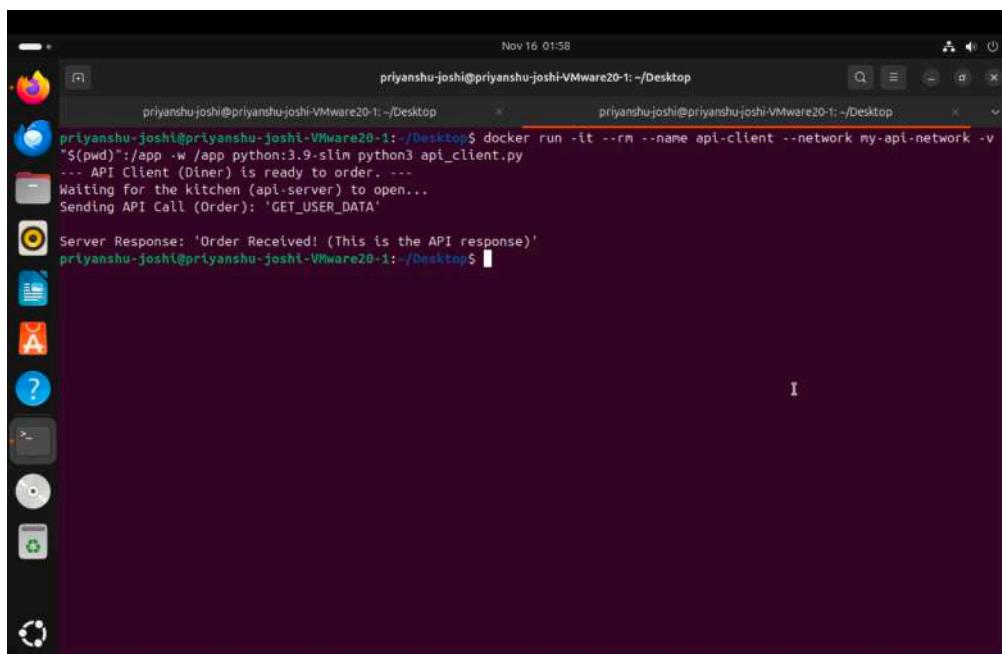
Main CPU: PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
3058 priyanshu 20 0 303M 5748 5364 S 0.0 0.3 0:00.00
3066 priyanshu 20 0 304M 6440 5800 S 0.0 0.3 0:00.00
3125 priyanshu 20 0 231M 5864 5480 S 0.0 0.3 0:00.00
3186 priyanshu 39 19 793M 29348 17712 S 0.0 1.5 0:00.05
3197 priyanshu 20 0 624M 82744 65096 S 0.0 4.1 0:00.05
3226 priyanshu 20 0 2462M 25616 20880 S 0.0 1.3 0:00.02
3239 priyanshu 20 0 687M 13484 10936 S 0.0 0.7 0:00.07
3249 priyanshu 20 0 554M 38976 28876 S 0.0 1.9 0:00.07
3270 priyanshu 20 0 266M 24548 17664 S 0.0 1.2 0:00.03
3286 priyanshu 20 0 414M 25880 19224 S 0.0 1.3 0:00.03
3337 priyanshu 20 0 4140M 10436 9028 S 0.0 0.5 0:00.27
3388 priyanshu 20 0 425M 30132 21484 S 0.0 1.5 0:00.07
3469 priyanshu 20 0 694M 57708 44792 S 0.0 2.9 0:05.05
3477 priyanshu 20 0 10976 4816 3468 S 0.0 0.2 0:00.01
3811 priyanshu 20 0 1803M 27644 19468 S 0.0 1.4 0:00.04
3789 priyanshu 20 0 10976 5092 3556 S 0.0 0.3 0:00.02
3922 priyanshu 20 0 1731M 27236 19380 S 0.0 1.4 0:00.02
3796 priyanshu 20 0 10976 4764 3356 S 0.0 0.2 0:00.00
3981 priyanshu 20 0 11364 4556 3078 R 1.3 0.2 0:01.60
3826 root 20 0 1280M 14784 10688 S 0.0 0.7 0:00.04
3847 root 20 0 1796 1108 980 S 0.0 0.1 0:00.00

```

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.



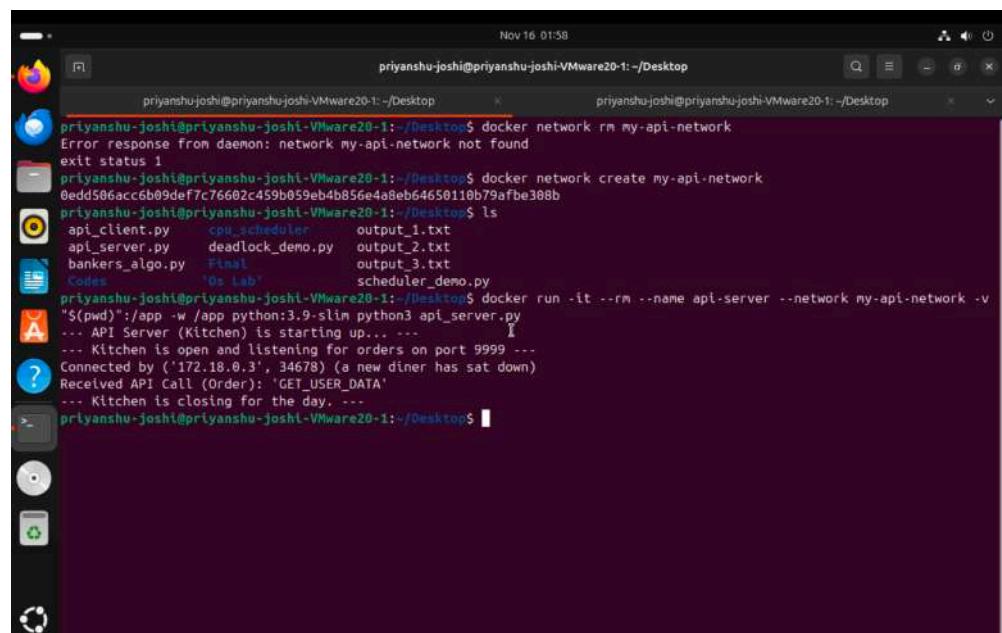
```

Nov 16 01:58 priyanshu-joshi@priyanshu-joshi-VMware20-1:~/Desktop
priyanshu-joshi@priyanshu-joshi-VMware20-1:~/Desktop$ docker run -it --rm --name api-client --network my-api-network -v
$(pwd)":/app -w /app python:3.9-slim python3 api_client.py
... API Client (Diner) is ready to order. ...
Waiting for the kitchen (api-server) to open...
Sending API Call (Order): 'GET_USER_DATA'

Server Response: 'Order Received! (This is the API response)'
priyanshu-joshi@priyanshu-joshi-VMware20-1:~/Desktop$ 

```

Client Side



```

Nov 16 01:58 priyanshu-joshi@priyanshu-joshi-VMware20-1:~/Desktop
priyanshu-joshi@priyanshu-joshi-VMware20-1:~/Desktop$ docker network rm my-api-network
Error response from daemon: network my-api-network not found
exit status 1
priyanshu-joshi@priyanshu-joshi-VMware20-1:~/Desktop$ docker network create my-api-network
0edd506acc6b09def7c76602c459b059eb4bb856e4a8eb6465010b79afbe308b
priyanshu-joshi@priyanshu-joshi-VMware20-1:~/Desktop$ ls
api_client.py    cpu_scheduler        output_1.txt
api_server.py    deadlock_demo.py   output_2.txt
bankers_algo.py  final               output_3.txt
Codes           'Os Lab'            scheduler_demo.py
priyanshu-joshi@priyanshu-joshi-VMware20-1:~/Desktop$ docker run -it --rm --name api-server --network my-api-network -v
$(pwd)":/app -w /app python:3.9-slim python3 api_server.py
... API Server (Kitchen) is starting up...
... Kitchen is open and listening for orders on port 9999 ...
Connected by ('172.18.0.3', 34678) (a new diner has sat down)
Received API Call (Order): 'GET_USER_DATA'
... Kitchen is closing for the day. ...
priyanshu-joshi@priyanshu-joshi-VMware20-1:~/Desktop$ 

```

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