Docker Assignment

Task 1: Verifying Kernel Sharing

Objective:

Prove that docker containers share the host's kernel instead of running their own kernel like in virtual machines.

Methodology:

1. Launched multiple containers with different Linux distributions - Ubuntu 22.04, Ubuntu 20.04, Alpine
2. Checked kernel version using "*uname -r*" and "*/proc/version*" from both host and containers.
3. Compare the kernel information across all systems

Host kernel information

```
ubuntu@ip-172-31-20-1:~$ uname -r
6.14.0-1015-aws
ubuntu@ip-172-31-20-1:~$ cat /proc/version
Linux version 6.14.0-1015-aws (buildd@lcy02-amd64-042) (x86_64-linux-gnu-gcc-13 (Ubuntu 13.3.0
-6ubuntu2~24.04) 13.3.0, GNU ld (GNU Binutils for Ubuntu) 2.42) #15~24.04.1-Ubuntu SMP Tue Sep
 23 22:44:48 UTC 2025
```

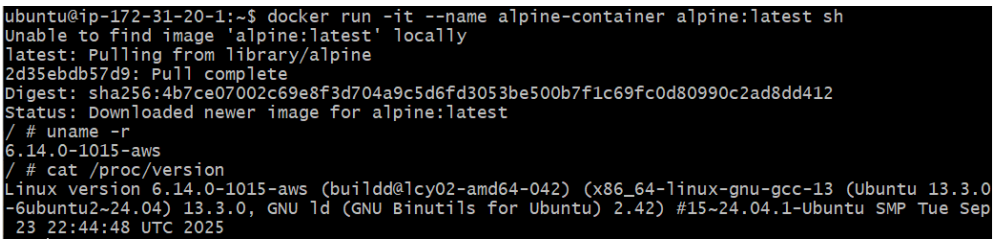
Ubuntu container - 22.04

```
ubuntu@ip-172-31-20-1:~$ docker run -it --name ubuntu-container ubuntu:22.04 bash
Unable to find image 'ubuntu:22.04' locally
22.04: Pulling from library/ubuntu
af6eca94c810: Pull complete
Digest: sha256:09506232a8004baa32c47d68f1e5c307d648fdd59f5e7eaa42aaf87914100db3
Status: Downloaded newer image for ubuntu:22.04
root@0a303cecf84b:/# uname -r
6.14.0-1015-aws
root@0a303cecf84b:/# cat /proc/version
Linux version 6.14.0-1015-aws (buildd@lcy02-amd64-042) (x86_64-linux-gnu-gcc-13 (Ubuntu 13.3.0
-6ubuntu2~24.04) 13.3.0, GNU ld (GNU Binutils for Ubuntu) 2.42) #15~24.04.1-Ubuntu SMP Tue Sep
 23 22:44:48 UTC 2025
```

Ubuntu container - 20.04

```
ubuntu@ip-172-31-20-1:~$ docker run -it --name ubuntu20-container ubuntu:20.04 bash
Unable to find image 'ubuntu:20.04' locally
20.04: Pulling from library/ubuntu
13b7e930469f: Pull complete
Digest: sha256:8feb4d8ca5354def3d8fce243717141ce31e2c428701f6682bd2fafe15388214
Status: Downloaded newer image for ubuntu:20.04
root@b4799102a52d:/# uname -r
6.14.0-1015-aws
root@b4799102a52d:/# cat /proc/version
Linux version 6.14.0-1015-aws (buildd@lcy02-amd64-042) (x86_64-linux-gnu-gcc-13 (Ubuntu 13.3.0
-6ubuntu2~24.04) 13.3.0, GNU ld (GNU Binutils for Ubuntu) 2.42) #15~24.04.1-Ubuntu SMP Tue Sep
 23 22:44:48 UTC 2025
```

Alpine container



All these 4 environments show **identical kernel information:**

- Same kernel version: 6.14.0-1015-aws
- Same build timestamp: Tue Sep 23 22:44:48 UTC 2025
- Same compiler: x86_64-linux-gnu-gcc-13 (Ubuntu 13.3.0)
- Same build machine: buildd@lcy02-amd64-042

Conclusion:

Despite running different Linux distributions and versions:

- All containers reported the **identical kernel version** as the host
- Each container has different **user-space tools**(Programs outside the kernel that manage system tasks) and **OS configurations**
- The kernel information is **exactly the same** across all systems in one machine.

This proves that:

1. Containers **do not have their own kernels**
2. All containers **share the host's kernel**
3. Containers are fundamentally different from virtual machines

Why containers are not Virtual Machines:

Virtual Machines:	Docker Containers:
Run a complete guest operating system	Only package application + dependencies
Have their own kernel	Share the host's kerne

Task 2: **Process and PID Mapping**

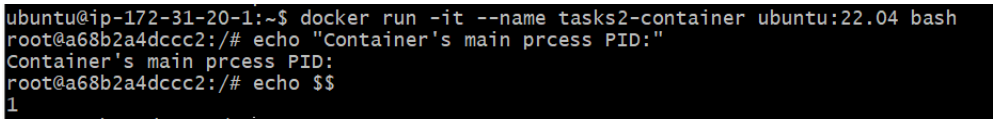
Objective:

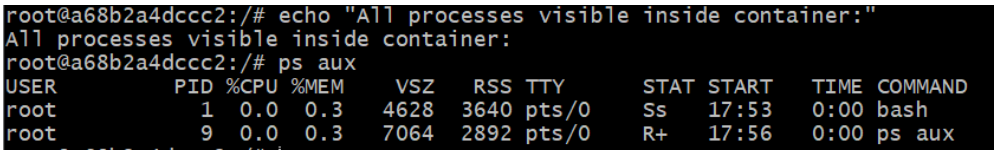
Demonstrate that a container's processes are regular host processes.

Methodology:

1. Launched a container interactively.
2. Identified the container's main PID inside using *$$*.
3. Found the corresponding host PID using *docker inspect*.
4. Verified the mapping using */proc* filesystem.
5. Observed the relationship between container PID 1 and host PID tree.
6. Drew process hierarchy showing the mapping.

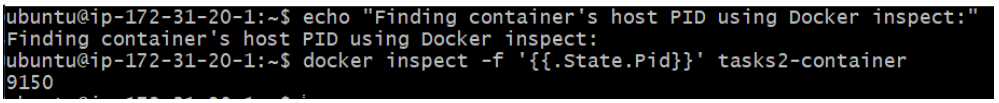
Container’s Internal View

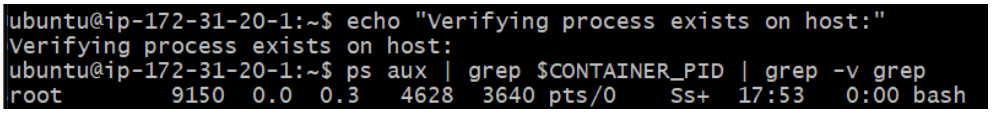




The container’s bash process reports PID: 1

Host’s External View





The bash process has host PID: 9150

The Dual PID Proof - /proc Filesystem

```
ubuntu@ip-172-31-20-1:~$ cat /proc/$CONTAINER_PID/status | grep -E "Name|Pid|PPid|NSpid"
Name:   bash
Pid:    9150
PPid:   9126
TracerPid:      0
NSpid:  9150    1
```

The ‘NSpid’ field shows 2 numbers:

- Host’s view of the PID-9150
- Container’s view-1

This proves it is the same process with different PID values.

Process Hierarchy

```
ubuntu@ip-172-31-20-1:~$ echo "Process tree showing container's ancestry:"
Process tree showing container's ancestry:
ubuntu@ip-172-31-20-1:~$ pstree -a -p -s $CONTAINER_PID
systemd,1
  └─containerd-shim,9126 -namespace moby -ida68b2a4dccc29e30a5d35b38cf04414090d714e6
      └─bash,9150
```

Process Hierarchy Diagram

```
systemd (PID 1)
  ├─ dockerd
  6690       1 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
  ├─ containerd
  6553       1 /usr/bin/containerd
  └─ containerd-shim
       └─ bash (Container Process)
    PID    PPID CMD
      1       0 /sbin/init
      2       0 [kthreadd]
      3       2 [pool_workqueue_release]
      4       2 [kworker/R-rcu_gp]
      5       2 [kworker/R-sync_wq]
      6       2 [kworker/R-kvfree_rcu_reclaim]
      7       2 [kworker/R-slub_flushwq]
      8       2 [kworker/R-netns]
     11       2 [kworker/0:0H-events_highpri]
     13       2 [kworker/R-mm_percpu_wq]
     14       2 [rcu_tasks_rude_kthread]
     15       2 [rcu_tasks_trace_kthread]
     16       2 [ksoftirqd/0]
```

Aspect	Container View	Host View
PID	1	9150
Process Type	Appear as init	Regular child process
Parent	None visible	containerd-shim
Visibility	Only container processes	All systems Processes

Conclusion:

1. **Same process, different views:**
    - Inside container: PID 1
    - On host: PID 9150
    - The /proc/[PID]/status NSpid field shows both values
2. **No virtualization involved:**
    - Container process is a regular Linux process
    - Runs directly on host kernel
    - Parent is containerd-shim, not a hypervisor
3. **PID namespace creates isolation:**
    - Container sees its own PID numbering starting from 1
    - Host sees the real PID in its global namespace
    - Container cannot see host processes
4. **Process hierarchy remains intact:**
    - Container process is managed by Docker's runtime stack
    - Clear chain: dockerd → containerd → containerd-shim → container

- Container processes are regular host processes
- The same process has two different PIDs (host and container view)
- Container PID 1 maps to a specific host PID
- The process hierarchy shows Docker's runtime architecture

Task 3: **Exploring Namespace Isolation**

Objective:

Identify and compare namespace instances for containers to understand how Docker achieves process, network, and system isolation.

Methodology:

1. Launched two containers with default isolation
2. Used *'sudo readlink /proc/<pid>/ns/'* to examine namespace inodes
3. Compared namespaces between containers and host
4. Launched a third container with shared network namespace
5. Recorded observations for PID, NET, MNT, USER, and UTS namespaces

Container1 Namespace

```
sudo lsns -p $CONTAINER1_PID

        NS TYPE   NPROCS   PID USER COMMAND
4026531834 time       118     1 root /sbin/init
4026531837 user       118     1 root /sbin/init
4026532222 mnt          1 11404 root sleep 3600
4026532223 uts          1 11404 root sleep 3600
4026532224 ipc          1 11404 root sleep 3600
4026532225 pid          1 11404 root sleep 3600
4026532226 cgroup       1 11404 root sleep 3600
4026532227 net          1 11404 root sleep 3600
```

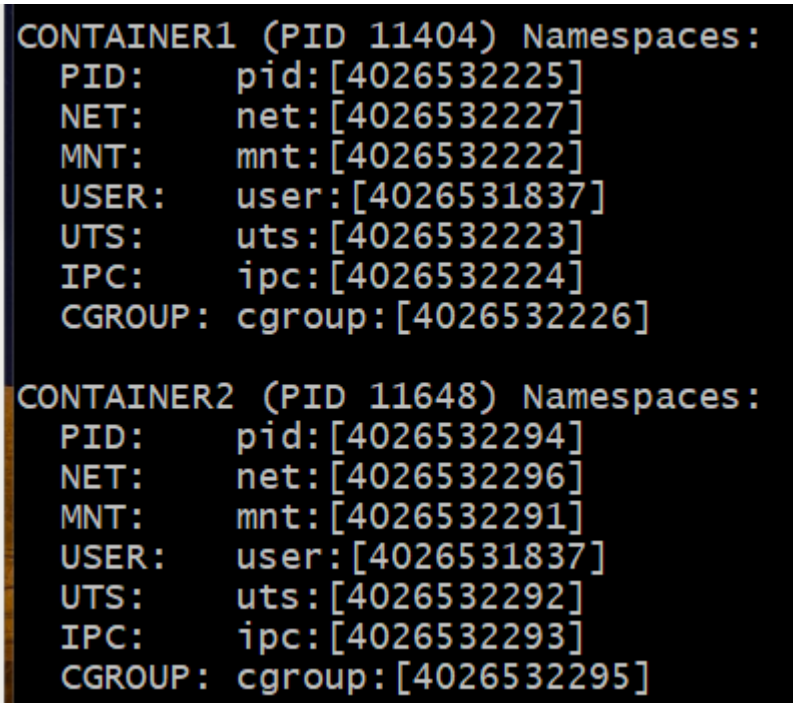
Host vs Container1 Comparison

```
HOST (PID 1) Namespaces:
  PID:    pid:[4026531836]
  NET:    net:[4026531840]
  MNT:    mnt:[4026531841]
  USER:   user:[4026531837]
  UTS:    uts:[4026531838]
  IPC:    ipc:[4026531839]

CONTAINER1 (PID 11404) Namespaces:
  PID:    pid:[4026532225]
  NET:    net:[4026532227]
  MNT:    mnt:[4026532222]
  USER:   user:[4026531837]
  UTS:    uts:[4026532223]
  IPC:    ipc:[4026532224]
```

- PID, NET, MNT, UTS, IPC namespaces are **different** from host

- USER namespace **may be same** as host (default behavior)
- This proves container has its own isolated view for most resources

Container1 vs Container2 Comparison



- Each container has **unique** PID, NET, MNT, UTS, and IPC namespaces, but may **share** the USER namespace with host.

Conclusion:

Unique Namespaces (Per Container):

1. **PID Namespace:**
    - Each container gets unique PID namespace
    - Container sees only its own processes
    - Process numbering starts from 1
2. **NET Namespace:**
    - Each container has own network stack
    - Separate IP addresses, routing tables, firewall rules
    - Virtual network interfaces
3. **MNT Namespace:**

- Each container has isolated filesystem tree
- Own root filesystem
- Independent mount points

4. **UTS Namespace:**
    - Each container has unique hostname
    - Independent domain name
5. **IPC Namespace:**
    - Isolated inter-process communication
    - Separate message queues, semaphores

Potentially Shared Namespaces:

1. **USER Namespace:**
    - Often shared with host by default
    - UIDs/GIDs map to host users
    - Can be isolated with user namespace mapping
2. **CGROUP Namespace:**
    - May be shared depending on configuration
    - Controls resource visibility

- Each container has unique PID, NET, MNT, UTS, and IPC namespaces.
- USER namespace is often shared with host
- Namespace inodes can be compared to verify isolation
- Namespaces can be selectively shared using Docker flags
- This namespace-based isolation is what makes containers lightweight yet secure

Task 4: Observing New Namespace Creation

Objective:

Monitor system calls during container creation to identify namespace creation flags.

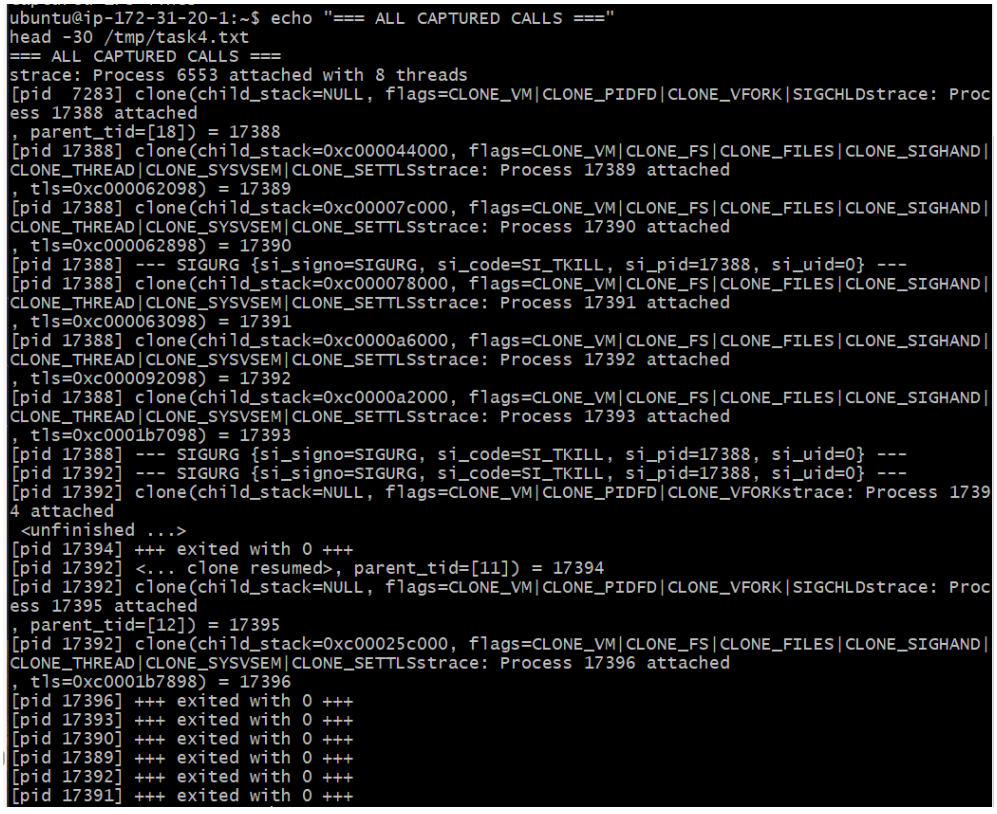
Methodology:

1. Used strace to monitor containerd process
2. Captured clone(), setns(), and unshare() system calls
3. Created container during active monitoring
4. Analyzed captured calls for namespace flags

Monitoring Setup

```
ubuntu@ip-172-31-20-1:~$ wc -l /tmp/task4.txt
echo "Captured $(wc -l < /tmp/task4.txt) lines"
178 /tmp/task4.txt
Captured 178 lines
ubuntu@ip-172-31-20-1:~$
```

Raw Captured calls



```
ubuntu@ip-172-31-20-1:~$ echo "=== ALL CAPTURED CALLS ==="
head -30 /tmp/task4.txt
=== ALL CAPTURED CALLS ===
strace: Process 6553 attached with 8 threads
[pid  7283] clone(child_stack=NULL, flags=CLONE_VM|CLONE_PIDFD|CLONE_VFORK|SIGCHLDstrace: Proc
ess 17388 attached
, parent_tid=[18]) = 17388
[pid 17388] clone(child_stack=0xc000044000, flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|
CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLSstrace: Process 17389 attached
, tls=0xc000062098) = 17389
[pid 17388] clone(child_stack=0xc00007c000, flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|
CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLSstrace: Process 17390 attached
, tls=0xc000062898) = 17390
[pid 17388] --- SIGURG {si_signo=SIGURG, si_code=SI_TKILL, si_pid=17388, si_uid=0} ---
[pid 17388] clone(child_stack=0xc0000780000, flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|
CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLSstrace: Process 17391 attached
, tls=0xc000063098) = 17391
[pid 17388] clone(child_stack=0xc0000a6000, flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|
CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLSstrace: Process 17392 attached
, tls=0xc000092098) = 17392
[pid 17388] clone(child_stack=0xc0000a2000, flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|
CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLSstrace: Process 17393 attached
, tls=0xc0001b7098) = 17393
[pid 17388] --- SIGURG {si_signo=SIGURG, si_code=SI_TKILL, si_pid=17388, si_uid=0} ---
[pid 17392] --- SIGURG {si_signo=SIGURG, si_code=SI_TKILL, si_pid=17388, si_uid=0} ---
[pid 17392] clone(child_stack=NULL, flags=CLONE_VM|CLONE_PIDFD|CLONE_VFORKstrace: Process 1739
4 attached
 <unfinished ...>
[pid 17394] +++ exited with 0 +++
[pid 17392] <... clone resumed>, parent_tid=[11]) = 17394
[pid 17392] clone(child_stack=NULL, flags=CLONE_VM|CLONE_PIDFD|CLONE_VFORK|SIGCHLDstrace: Proc
ess 17395 attached
, parent_tid=[12]) = 17395
[pid 17392] clone(child_stack=0xc00025c000, flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|
CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLSstrace: Process 17396 attached
, tls=0xc0001b7898) = 17396
[pid 17396] +++ exited with 0 +++
[pid 17393] +++ exited with 0 +++
[pid 17390] +++ exited with 0 +++
[pid 17389] +++ exited with 0 +++
[pid 17392] +++ exited with 0 +++
[pid 17391] +++ exited with 0 +++
```

- System calls during container creation

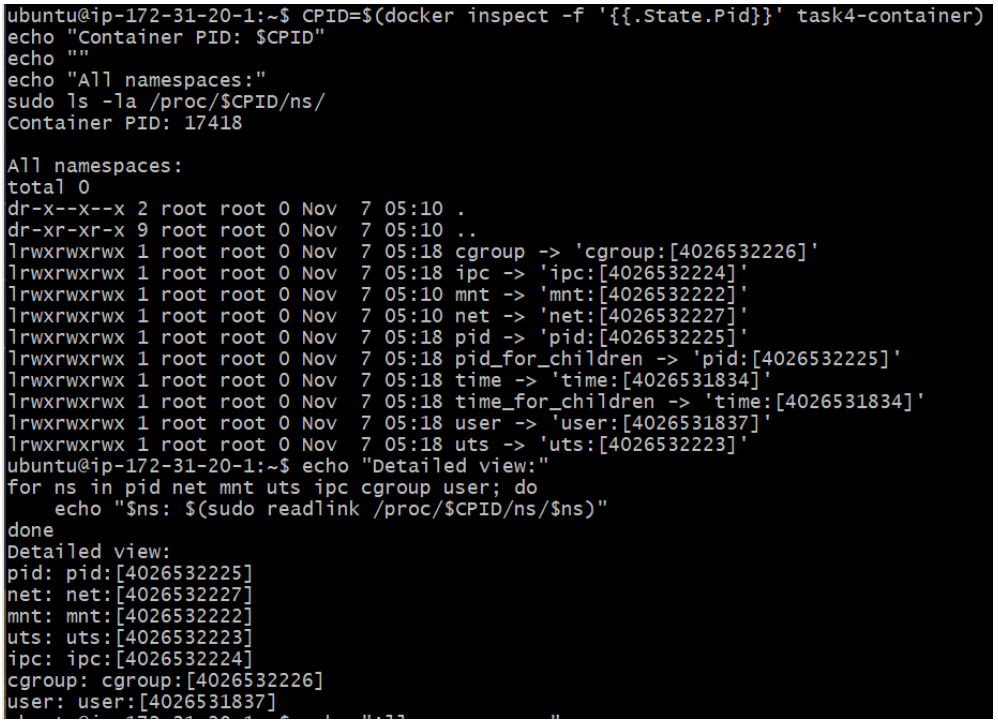
Namespace Flags Detected

```
ubuntu@ip-172-31-20-1:~$ echo "Looking for CLONE_NEW flags..."
echo ""
grep -i "CLONE_NEW" /tmp/task4.txt | head -10
echo ""
echo "Found $(grep -c 'CLONE_NEW' /tmp/task4.txt) namespace-related calls"
Looking for CLONE_NEW flags...

[pid 17417] unshare(CLONE_NEWNS|CLONE_NEWCGROUP|CLONE_NEWUTS|CLONE_NEWIPC|CLONE_NEWPID|CLONE_N
EWNET) = 0
[pid 17414] setns(19, CLONE_NEWNS)      = 0

Found 2 namespace-related calls
```

Call Type Breakdown

```
ubuntu@ip-172-31-20-1:~$ echo ""
echo "CLONE calls captured:"
grep "clone(" /tmp/task4.txt | wc -l
echo ""
echo "SETNS calls captured:"
grep "setns(" /tmp/task4.txt | wc -l
echo ""
echo "UNSHARE calls captured:"
grep "unshare(" /tmp/task4.txt | wc -l
echo ""
echo "Sample CLONE calls:"
grep "clone(" /tmp/task4.txt | head -3
echo ""
echo "Sample SETNS calls:"
grep "setns(" /tmp/task4.txt | head -3

CLONE calls captured:
25

SETNS calls captured:
1

UNSHARE calls captured:
2

Sample CLONE calls:
[pid  7283] clone(child_stack=NULL, flags=CLONE_VM|CLONE_PIDFD|CLONE_VFORK|SIGCHLDstrace: Proc
ess 17388 attached
[pid 17388] clone(child_stack=0xc000044000, flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|
CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLSstrace: Process 17389 attached
[pid 17388] clone(child_stack=0xc00007c000, flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|
CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLSstrace: Process 17390 attached

Sample SETNS calls:
[pid 17414] setns(19, CLONE_NEWNS)      = 0
```

Container Namespace Verification



- pid - Process ID isolation
- net - Network isolation
- mnt - Mount/filesystem isolation
- uts - Hostname isolation
- ipc - IPC isolation
- cgroup - Cgroup isolation

This proves the clone() call with CLONE_NEW* flags was executed successfully.

Conclusion

How Namespace Creation Works:

The Process:

1. User runs docker run
2. dockerd calls containerd
3. containerd calls runc
4. runc executes: clone() with multiple CLONE_NEW* flags
5. Kernel creates all namespaces simultaneously

6. New process starts in isolated environment

The System Call:

CLONE_NEWPID

- Creates new PID namespace
- Container sees only its own processes
- First process becomes PID 1
- **Security:** Can't see or affect host processes

CLONE_NEWNET

- Creates new network namespace
- Own network interfaces, IP addresses, routing
- Independent firewall rules
- **Security:** Complete network isolation

CLONE_NEWNS

- Creates new mount namespace
- Own root filesystem
- Separate mount points
- **Security:** Can't access host files

CLONE_NEWUTS

- Creates new UTS namespace
- Own hostname and domain name
- Independent system identity
- **Use case:** Service identification

CLONE_NEWIPC

- Creates new IPC namespace
- Separate message queues, semaphores, shared memory
- **Security:** Prevents IPC eavesdropping

CLONE_NEWCGROUP

- Creates new cgroup namespace

- Limited cgroup hierarchy view
- **Security:** Hides host resource information

Why This Matters:

Single Call, Complete Isolation:

- All namespaces created in one system call
- Atomic operation
- No multi-step process needed

No Virtualization Overhead:

- Kernel feature, not hardware emulation
- Creation happens in microseconds
- Minimal memory footprint
- Shares kernel with host

Security Through Isolation:

- Each namespace type protects different resources
- Multiple independent isolation layers
- Kernel-enforced boundaries
- Defense in depth

Efficiency:

- Fast container startup
- Low resource usage
- High container density possible
- This is why containers beat VMs for microservices

Container isolation is achieved through a single clone() system call with combined namespace flags. This is fundamentally different from virtualization - it's a lightweight Linux kernel feature that creates isolated views of system resources instantly, with near-zero overhead.

The magic is one system call transforms a regular process into a fully isolated container environment.

Task 5: **Investigating cgroup Assignments**

Objective:

Demonstrate how cgroups enforce CPU and memory limits on containers.

Methodology:

1. Created container with —cpus=”0.5 and —memory=”256m”
2. Located cgroup paths in /proc/[PID]/cgroup
3. Verified limits in /sys/fs/cgroup/
4. Monitored resource usage with docker stats

Container with Limits

```
ubuntu@ip-172-31-20-1:~$ docker run -dit --name cgroup-test --cpus="0.5" --memory="256m" ubunt
u:22.04 bash -c "while true; do echo test > /dev/null; done"
e8e49db1f7d28ab0dac7baba5241088eefd46a72dd9c0a01249d33591a653aa8
ubuntu@ip-172-31-20-1:~$
```

Cgroup Path

```
ubuntu@ip-172-31-20-1:~$ echo ""
cat /proc/$CPID/cgroup

0::/system.slice/docker-e8e49db1f7d28ab0dac7baba5241088eefd46a72dd9c0a01249d33591a653aa8.scope
ubuntu@ip-172-31-20-1:~$
```

CPU Limits

```
ubuntu@ip-172-31-20-1:~$ echo ""
CGROUP_PATH=$(docker inspect -f '{{.HostConfig.CgroupParent}}' cgroup-test)
CONTAINER_ID=$(docker inspect -f '{{.Id}}' cgroup-test)

ubuntu@ip-172-31-20-1:~$ echo "CPU quota and period:"
sudo cat /sys/fs/cgroup/system.slice/docker-${CONTAINER_ID}.scope/cpu.max 2>/dev/null || \
sudo cat /sys/fs/cgroup/cpu,cpuacct/docker/${CONTAINER_ID}/cpu.cfs_quota_us 2>/dev/null || \
echo "Checking alternative path..."
CPU quota and period:
50000 100000
ubuntu@ip-172-31-20-1:~$
ubuntu@ip-172-31-20-1:~$ docker stats cgroup-test --no-stream
CONTAINER ID   NAME          CPU %     MEM USAGE / LIMIT   MEM %     NET I/O         BLOCK I/O
   PIDS
e8e49db1f7d2   cgroup-test   50.03%    904KiB / 256MiB     0.34%     1.09kB / 126B   86kB / 0B
   1
```

Memory Limits

```
268435456

Current usage:
CONTAINER ID   NAME          CPU %     MEM USAGE / LIMIT   MEM %     NET I/O         BLOCK I/O
   PIDS
e8e49db1f7d2   cgroup-test   50.39%    648KiB / 256MiB     0.25%     1.23kB / 126B   86kB / 0B
   1
ubuntu@ip-172-31-20-1:~$
```

Resource Monitoring

```
Watching for 10 seconds...
CONTAINER ID   NAME          CPU %     MEM USAGE / LIMIT   MEM %     NET I/O         BLOCK I/O
   PIDS
e8e49db1f7d2   cgroup-test   50.03%    904KiB / 256MiB     0.34%     1.23kB / 126B   86kB / 0B
   1
CONTAINER ID   NAME          CPU %     MEM USAGE / LIMIT   MEM %     NET I/O         BLOCK I/O
   PIDS
e8e49db1f7d2   cgroup-test   50.38%    904KiB / 256MiB     0.34%     1.23kB / 126B   86kB / 0B
   1
CONTAINER ID   NAME          CPU %     MEM USAGE / LIMIT   MEM %     NET I/O         BLOCK I/O
   PIDS
e8e49db1f7d2   cgroup-test   49.99%    648KiB / 256MiB     0.25%     1.23kB / 126B   86kB / 0B
   1
```

- Docker stats showing enforcement.

Active Controllers

```
ubuntu@ip-172-31-20-1:~$ echo ""
cat /proc/$(docker inspect -f '{{.State.Pid}}' cgroup-test)/cgroup

0::/system.slice/docker-e8e49db1f7d28ab0dac7baba5241088eefd46a72dd9c0a01249d33591a653aa8.scope
ubuntu@ip-172-31-20-1:~$ ^C
```

- Active controllers are cpu, memory, io, pids

Quota Files

```
ubuntu@ip-172-31-20-1:~$ CONTAINER_ID=$(docker inspect -f '{{.Id}}' cgroup-test)

echo "CPU Limit (cpu.max):"
sudo cat /sys/fs/cgroup/system.slice/docker-${CONTAINER_ID}.scope/cpu.max

echo ""
echo "Memory Limit (memory.max):"
sudo cat /sys/fs/cgroup/system.slice/docker-${CONTAINER_ID}.scope/memory.max
CPU Limit (cpu.max):
50000 100000

Memory Limit (memory.max):
268435456
```

- cpu.max = 50000 / 100000 - 0.5 CPU
- memory.max = 268435456 - 256MB

Control groups - Linux kernel feature that limits and monitors resource usage.

Controllers observed:

- **cpu:** Limits CPU time
- **memory:** Limits RAM usage
- **pids:** Limits number of processes
- **io:** Limits disk I/O

How it works:

1. Docker creates cgroup for container
2. Sets limits in cgroup files (cpu.max, memory.max)
3. Kernel enforces limits automatically
4. Container cannot exceed limits

Significance:

- Prevents resource monopolization
- Enables multi-tenancy
- Guarantees fair resource sharing
- Foundation for Kubernetes resource management

Task 6: **Resource Behavior Under Load**

Objective:

Demonstrate cgroup enforcement through resource stress testing and observe throttling, OOM kills, and kernel-level resource management.

Methodology:

1. **Created container** with CPU (1 core) and memory (512MB) limits
2. **Installed stress-ng** for controlled resource pressure
3. **Monitored baseline** resource usage
4. **CPU stress test**: 2 workers on 1-core limit
5. **Memory stress test**: Attempted 1GB allocation on 512MB limit
6. **Observed kernel enforcement** via throttling and OOM killer
7. **Analyzed cgroup statistics** for evidence
8. **Compared host vs container** resource perspectives

Container Creation:

```
ubuntu@ip-172-31-20-1:~$ docker run -dit --name stress-test --cpus="1.0" --memory="512m" --me
mory-swap="512m" ubuntu:22.04 bash

echo "Container created with:"
echo "  CPU: 1.0 core"
echo "  Memory: 512MB"
echo "  Swap: 512MB (same as memory = no extra swap)"
e12e841b2a068bb403e8074048f71e19101e15ac5f19d2565125157b51af8730
Container created with:
  CPU: 1.0 core
  Memory: 512MB
  Swap: 512MB (same as memory = no extra swap)
```

- Limits explicitly set

Baseline State

```
ubuntu@ip-172-31-20-1:~$ echo ""
CGROUP_PATH=$(docker inspect -f '{{.HostConfig.CgroupParent}}' cgroup-test)
CONTAINER_ID=$(docker inspect -f '{{.Id}}' cgroup-test)

ubuntu@ip-172-31-20-1:~$ echo "CPU quota and period:"
sudo cat /sys/fs/cgroup/system.slice/docker-${CONTAINER_ID}.scope/cpu.max 2>/dev/null || \
sudo cat /sys/fs/cgroup/cpu,cpuacct/docker/${CONTAINER_ID}/cpu.cfs_quota_us 2>/dev/null || \
echo "Checking alternative path..."
CPU quota and period:
50000 100000
ubuntu@ip-172-31-20-1:~$
ubuntu@ip-172-31-20-1:~$ docker stats cgroup-test --no-stream
CONTAINER ID   NAME          CPU %     MEM USAGE / LIMIT   MEM %     NET I/O         BLOCK I/O
   PIDS
e8e49db1f7d2   cgroup-test   50.03%    904KiB / 256MiB     0.34%     1.09kB / 126B   86kB / 0B
   1
```

CPU Stress Test

```
Every 1.0s: docker stats stress-test --no-stream      ip-172-31-20-1: Fri Nov  7 06:36:32 2025

CONTAINER ID   NAME          CPU %     MEM USAGE / LIMIT   MEM %     NET I/O           BLOCK I
/O        PIDS
e12e841b2a06   stress-test   0.00%     13.04MiB / 512MiB   2.55%     46.6MB / 95.1kB   10.4MB
/ 135MB   2
```

```
ubuntu@ip-172-31-20-1:~$ echo "Starting CPU stress (2 workers on 1 CPU limit)...
"
docker exec stress-test stress-ng --cpu 2 --timeout 30s --metrics-brief
Starting CPU stress (2 workers on 1 CPU limit)...
stress-ng: info:  [346] setting to a 30 second run per stressor
stress-ng: info:  [346] dispatching hogs: 2 cpu
stress-ng: info:  [346] successful run completed in 30.00s
stress-ng: info:  [346] stressor       bogo ops real time  usr time  sys time
bogo ops/s     bogo ops/s
stress-ng: info:  [346]                           (secs)    (secs)    (secs)   (
real time) (usr+sys time)
stress-ng: info:  [346] cpu               28721     30.00     30.04      0.00
    957.35         956.09
ubuntu@ip-172-31-20-1:~$
```

- CPU% capped at ~100% despite 2 workers

CPU Throttling Proof

```
ubuntu@ip-172-31-20-1:~$ echo "CPU Statistics:"
sudo cat /sys/fs/cgroup/system.slice/docker-${CONTAINER_ID}.scope/cpu.stat 2>/de
v/null || \
sudo cat /sys/fs/cgroup/cpu,cpuacct/docker/${CONTAINER_ID}/cpu.stat 2>/dev/null
CPU Statistics:
usage_usec 39095654
user_usec 37898310
system_usec 1197344
nice_usec 0
core_sched.force_idle_usec 0
nr_periods 424
nr_throttled 314
throttled_usec 20713683
nr_bursts 0
burst_usec 0
ubuntu@ip-172-31-20-1:~$
```

nr_periods: 424
nr_throttled: 314
throttled_time: 20713683

Memory Stress Test

```
ubuntu@ip-172-31-20-1:~$ echo "Starting memory stress (trying to allocate 1GB on
 512MB limit)..."
docker exec stress-test stress-ng --vm 1 --vm-bytes 1G --timeout 20s --metrics-b
rief
Starting memory stress (trying to allocate 1GB on 512MB limit)...
stress-ng: info:  [354] setting to a 20 second run per stressor
stress-ng: info:  [354] dispatching hogs: 1 vm
stress-ng: error: [361] stress-ng-vm: gave up trying to mmap, no available memor
y
stress-ng: info:  [354] successful run completed in 10.02s
stress-ng: info:  [354] stressor       bogo ops real time  usr time  sys time
bogo ops/s     bogo ops/s
stress-ng: info:  [354]                           (secs)    (secs)    (secs)   (
real time) (usr+sys time)
stress-ng: info:  [354] vm                    0     10.01      0.00      0.00
      0.00           0.00
```

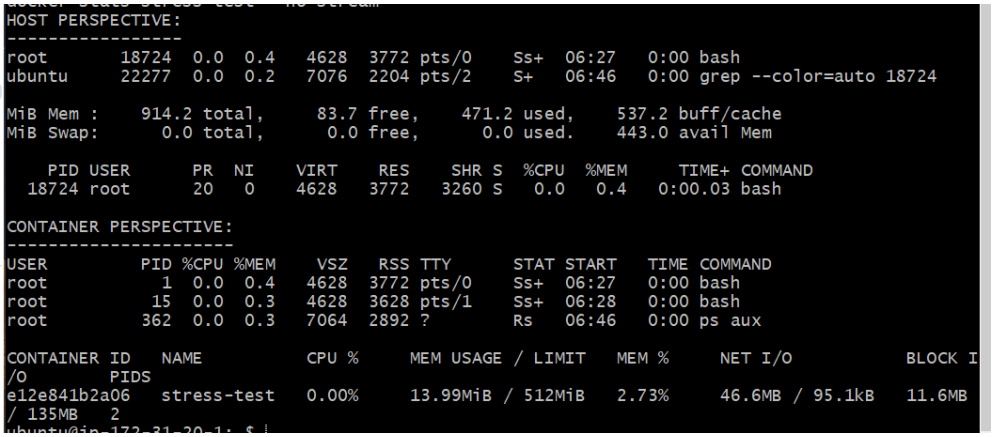
- Memory climbed to 100% of limit
- Process killed or severe slowdown

OOM Kill Evidence

```
ubuntu@ip-172-31-20-1:~$ docker exec oom-test bash -c "stress-ng --vm 1 --vm-bytes 500M --tim
eout 10s"
stress-ng: info:  [338] setting to a 10 second run per stressor
stress-ng: info:  [338] dispatching hogs: 1 vm
stress-ng: info:  [338] successful run completed in 10.03s
ubuntu@ip-172-31-20-1:~$ sudo dmesg | tail -20 | grep -i oom
[96072.591408] Memory cgroup out of memory: Killed process 24315 (stress-ng) total-vm:588268k
B, anon-rss:125392kB, file-rss:420kB, shmem-rss:0kB, UID:0 pgtables:1076kB oom_score_adj:1000
[96072.649665] Memory cgroup out of memory: Killed process 24316 (stress-ng) total-vm:588268k
B, anon-rss:125392kB, file-rss:288kB, shmem-rss:0kB, UID:0 pgtables:1076kB oom_score_adj:1000
[96072.700896] Memory cgroup out of memory: Killed process 24317 (stress-ng) total-vm:588268k
B, anon-rss:125392kB, file-rss:292kB, shmem-rss:0kB, UID:0 pgtables:1076kB oom_score_adj:1000
[96072.753098] Memory cgroup out of memory: Killed process 24318 (stress-ng) total-vm:588268k
B, anon-rss:125272kB, file-rss:420kB, shmem-rss:0kB, UID:0 pgtables:1076kB oom_score_adj:1000
```

- This proves the kernel OOM killer terminated the process when memory limit was exceeded.

Host vs Container View



- **Host**: Sees actual PID and resource usage
- **Container**: Sees PID 1 and limited resources
- **Key**: Same process, different perspectives

Analysis:

- CPU: cgroups enforce CPU quotas by throttling; nr_throttling and throttled_usec in cpu.stat are definitive evidence.
- Memory: when the container requests more memory than permitted, allocations fail (mmap errors) or the kernel may OOM-kill processes.
- Docker stats provides a high-level view; cgroup files and kernel logs provide canonical proof.

Conclusion

- Under CPU load the kernel throttled the container (nr_throttled > 0).
- Under memory pressure the container hit memory limits and allocation failed.
- These behaviors are visible in stress-ng output, docker stats, cgroup files, and kernel logs.

Task 7: **Filesystem Layer Analysis**

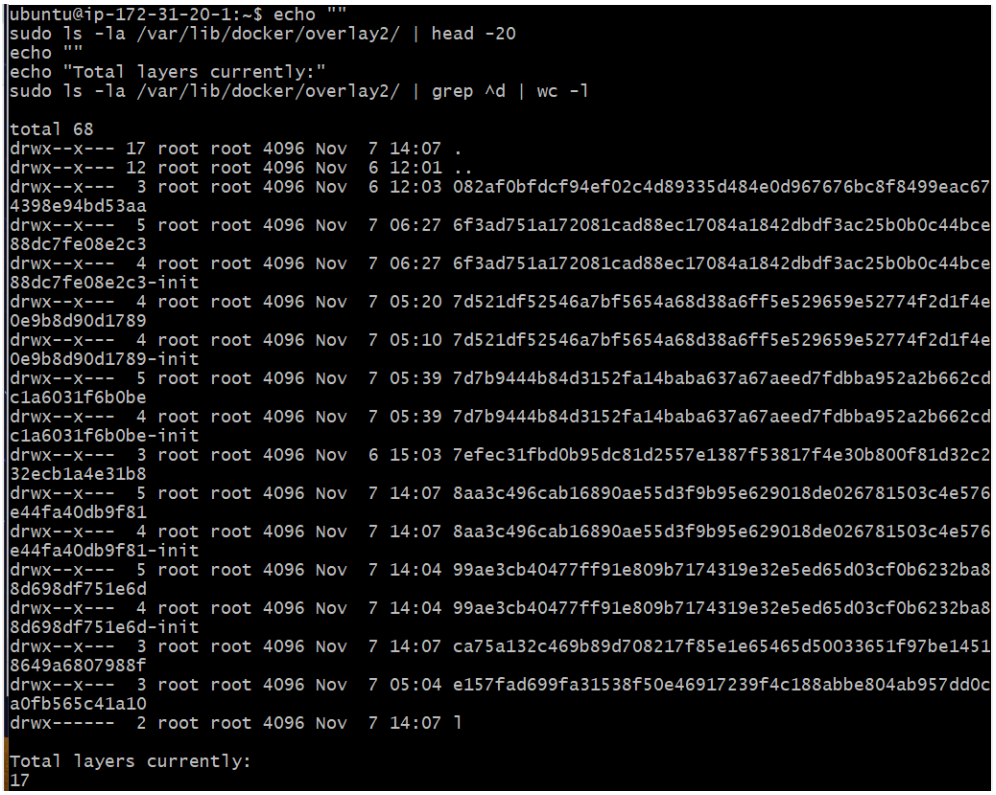
Objective:

Investigate Docker's overlay2 filesystem to understand the relationship between read-only image layers and writable container layers, and demonstrate the lifecycle of container filesystem changes.

Methodology:

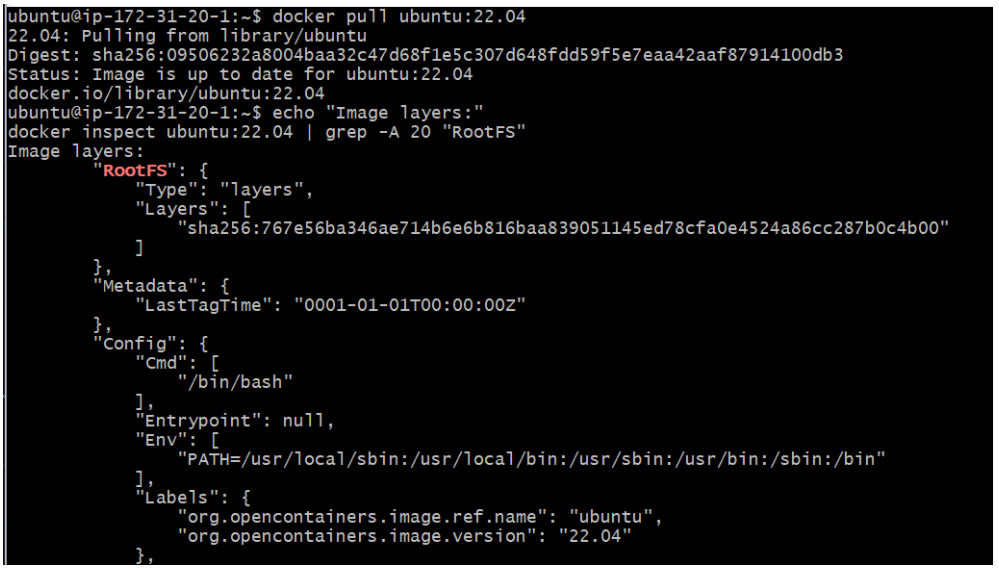
1. Inspected /var/lib/docker/overlay2/ directory structure
2. Pulled Ubuntu image and examined its layer composition
3. Created container and identified:
    - **LowerDir** (read-only image layers)
    - **UpperDir** (writable container layer)
    - **MergedDir** (unified view)
    - **WorkDir** (overlay filesystem working directory)
4. Created and modified files inside the container
5. Verified file presence in writable layer from host system
6. Demonstrated copy-on-write mechanism
7. Stopped container and verified data persistence
8. Removed container and confirmed writable layer deletion
9. Verified image layers remain intact for reuse

Initial Overlay2 State



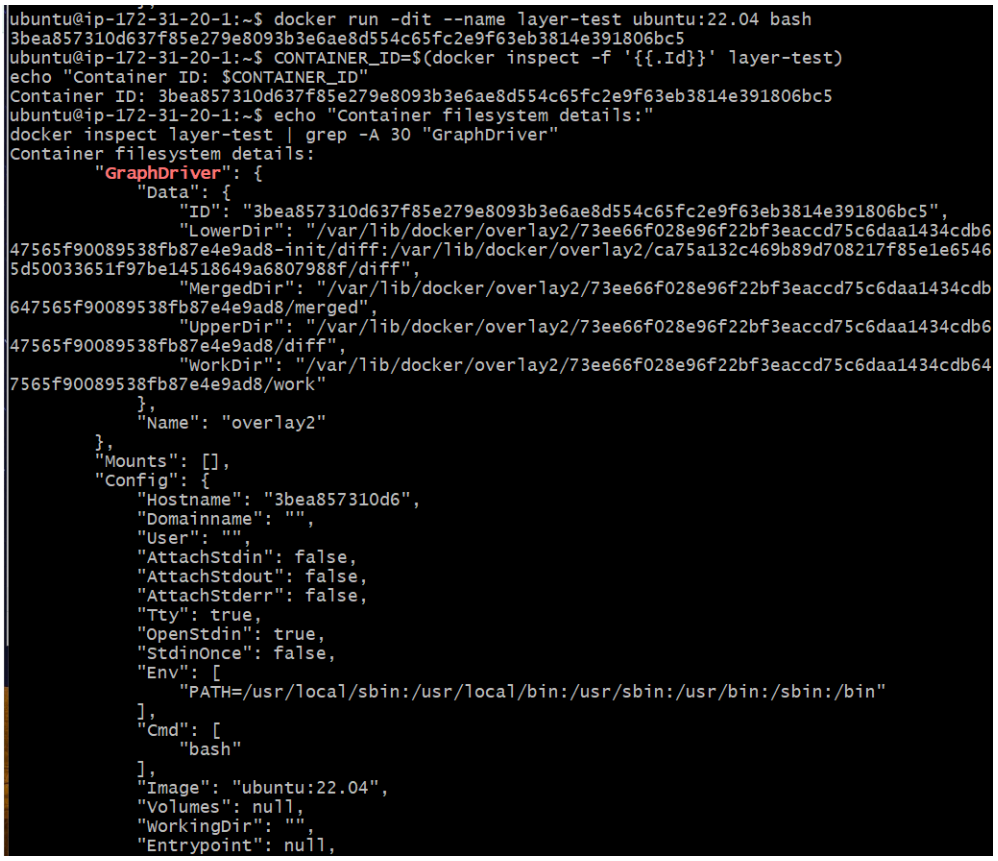
- Baseline directory listing of /var/lib/docker/overlay2/

Image Pull and Layer Inspection



- Ubuntu image pulled with multiple layers visible

Container Creation and GraphDriver Details



Read-Only Image Layers

```
ubuntu@ip-172-31-20-1:~$ LOWER_DIR=$(docker inspect -f '{{.GraphDriver.Data.LowerDir}}' layer
-test)
echo "Lower (Read-Only) Directories:"
echo "$LOWER_DIR" | tr ':' '\n'
Lower (Read-Only) Directories:
/var/lib/docker/overlay2/73ee66f028e96f22bf3eaccd75c6daa1434cdb647565f90089538fb87e4e9ad8-ini
t/diff
/var/lib/docker/overlay2/ca75a132c469b89d708217f85e1e65465d50033651f97be14518649a6807988f/dif
f
ubuntu@ip-172-31-20-1:~$ echo "Examining first read-only layer:"
FIRST_LAYER=$(echo "$LOWER_DIR" | cut -d':' -f1)
echo "Path: $FIRST_LAYER"
Examining first read-only layer:
Path: /var/lib/docker/overlay2/73ee66f028e96f22bf3eaccd75c6daa1434cdb647565f90089538fb87e4e9a
d8-init/diff
ubuntu@ip-172-31-20-1:~$ echo "Contents:"
sudo ls -lh "$FIRST_LAYER" 2>/dev/null || echo "Layer path may vary"
Contents:
total 8.0K
drwxr-xr-x 4 root root 4.0K Nov  7 14:22 dev
drwxr-xr-x 2 root root 4.0K Nov  7 14:22 etc
```

- LowerDir showing stacked read-only layers

Writable Container Layer Structure

```
ubuntu@ip-172-31-20-1:~$ UPPER_DIR=$(docker inspect -f '{{.GraphDriver.Data.UpperDir}}' layer
-test)
echo "Upper (Writable) Directory:"
echo "$UPPER_DIR"
Upper (Writable) Directory:
/var/lib/docker/overlay2/73ee66f028e96f22bf3eaccd75c6daa1434cdb647565f90089538fb87e4e9ad8/dif
f
ubuntu@ip-172-31-20-1:~$ echo "Initial contents of writable layer:"
sudo ls -la "$UPPER_DIR"
Initial contents of writable layer:
total 8
drwxr-xr-x 2 root root 4096 Nov  7 14:22 .
drwx--x--- 5 root root 4096 Nov  7 14:22 ..
ubuntu@ip-172-31-20-1:~$ echo "WorkDir (used by overlay filesystem):"
WORK_DIR=$(docker inspect -f '{{.GraphDriver.Data.WorkDir}}' layer-test)
echo "$WORK_DIR"
WorkDir (used by overlay filesystem):
/var/lib/docker/overlay2/73ee66f028e96f22bf3eaccd75c6daa1434cdb647565f90089538fb87e4e9ad8/wor
k
ubuntu@ip-172-31-20-1:~$ echo "MergedDir (unified view):"
MERGED_DIR=$(docker inspect -f '{{.GraphDriver.Data.MergedDir}}' layer-test)
echo "$MERGED_DIR"
MergedDir (unified view):
/var/lib/docker/overlay2/73ee66f028e96f22bf3eaccd75c6daa1434cdb647565f90089538fb87e4e9ad8/mer
ged
```

- UpperDir, WorkDir, and MergedDir paths identified

File Creation Inside Container

```
ubuntu@ip-172-31-20-1:~$ docker exec layer-test bash -c "echo 'This is a test file' > /test-f
ile.txt"
ubuntu@ip-172-31-20-1:~$ docker exec layer-test bash -c "mkdir -p /mydata && echo 'Container
data' > /mydata/data.txt"
ubuntu@ip-172-31-20-1:~$ docker exec layer-test bash -c "echo 'Modified root file' >> /etc/ho
stname"
ubuntu@ip-172-31-20-1:~$ echo "Files created in container:"
docker exec layer-test ls -lh /test-file.txt /mydata/data.txt
Files created in container:
-rw-r--r-- 1 root root 15 Nov  7 14:28 /mydata/data.txt
-rw-r--r-- 1 root root 20 Nov  7 14:28 /test-file.txt
ubuntu@ip-172-31-20-1:~$ echo "Verifying content:"
docker exec layer-test cat /test-file.txt
docker exec layer-test cat /mydata/data.txt
Verifying content:
This is a test file
Container data
```

- Test files created: /test-file.txt, /mydata/data.txt

Files Visible in Writable Layer from Host

```
ubuntu@ip-172-31-20-1:~$ echo "Verifying content:"
docker exec layer-test cat /test-file.txt
docker exec layer-test cat /mydata/data.txt
Verifying content:
This is a test file
Container data
ubuntu@ip-172-31-20-1:~$ UPPER_DIR=$(docker inspect -f '{{.GraphDriver.Data.UpperDir}}' layer
-test)
ubuntu@ip-172-31-20-1:~$ echo "Contents of writable layer after file creation:"
sudo ls -lR "$UPPER_DIR"
Contents of writable layer after file creation:
/var/lib/docker/overlay2/73ee66f028e96f22bf3eaccd75c6daa1434cdb647565f90089538fb87e4e9ad8/dif
f:
total 8
drwxr-xr-x 2 root root 4096 Nov  7 14:28 mydata
-rw-r--r-- 1 root root   20 Nov  7 14:28 test-file.txt

/var/lib/docker/overlay2/73ee66f028e96f22bf3eaccd75c6daa1434cdb647565f90089538fb87e4e9ad8/dif
f/mydata:
total 4
-rw-r--r-- 1 root root 15 Nov  7 14:28 data.txt
ubuntu@ip-172-31-20-1:~$ echo "Reading test file from host:"
sudo cat "$UPPER_DIR/test-file.txt"
Reading test file from host:
This is a test file
ubuntu@ip-172-31-20-1:~$ echo "Reading directory data from host:"
sudo cat "$UPPER_DIR/mydata/data.txt" 2>/dev/null || echo "Check nested path"
Reading directory data from host:
Container data
ubuntu@ip-172-31-20-1:~$ echo "Modified system files (copy-on-write):"
sudo ls -lh "$UPPER_DIR/etc/" 2>/dev/null || echo "etc modified via CoW"
Modified system files (copy-on-write):
etc modified via CoW
```

- Host filesystem showing container files in UpperDir

Copy-on-Write Demonstration

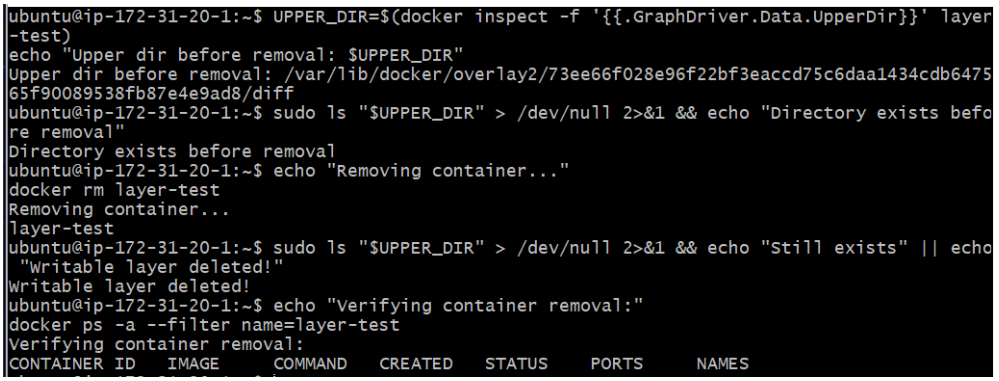
```
ubuntu@ip-172-31-20-1:~$ docker exec layer-test bash -c "echo 'MODIFIED BY CONTAINER' >> /etc
/bash.bashrc"
ubuntu@ip-172-31-20-1:~$ echo "Checking writable layer for copied file:"
UPPER_DIR=$(docker inspect -f '{{.GraphDriver.Data.UpperDir}}' layer-test)
sudo ls -lh "$UPPER_DIR/etc/bash.bashrc" 2>/dev/null || \
sudo find "$UPPER_DIR" -name "bash.bashrc" -exec ls -lh {} \;
Checking writable layer for copied file:
-rw-r--r-- 1 root root 2.4K Nov  7 14:34 /var/lib/docker/overlay2/73ee66f028e96f22bf3eaccd75c
6daa1434cdb647565f90089538fb87e4e9ad8/diff/etc/bash.bashrc
```

- Modified /etc/bash.bashsrc copied to writable layer

Persistence After Stopping Container

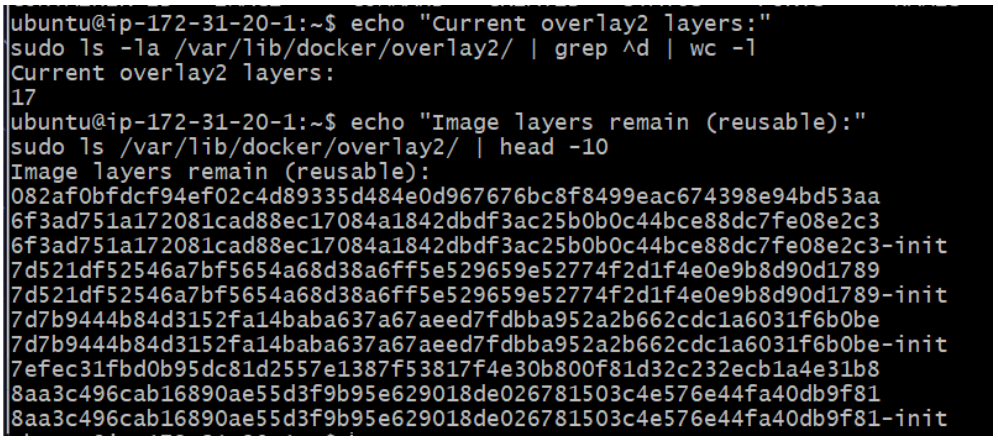
```
ubuntu@ip-172-31-20-1:~$ docker stop layer-test
layer-test
ubuntu@ip-172-31-20-1:~$ echo "Checking if writable layer still exists:"
UPPER_DIR=$(docker inspect -f '{{.GraphDriver.Data.UpperDir}}' layer-test)
echo "Upper dir path: $UPPER_DIR"
Checking if writable layer still exists:
Upper dir path: /var/lib/docker/overlay2/73ee66f028e96f22bf3eaccd75c6daa1434cdb647565f9008953
8fb87e4e9ad8/diff
ubuntu@ip-172-31-20-1:~$ sudo ls -lh "$UPPER_DIR/test-file.txt" 2>/dev/null && echo "File per
sists!" || echo "Path check needed"
-rw-r--r-- 1 root root 20 Nov  7 14:28 /var/lib/docker/overlay2/73ee66f028e96f22bf3eaccd75c6d
aa1434cdb647565f90089538fb87e4e9ad8/diff/test-file.txt
File persists!
ubuntu@ip-172-31-20-1:~$ echo "Container state:"
docker ps -a --filter name=layer-test
Container state:
CONTAINER ID   IMAGE          COMMAND   CREATED          STATUS                              PO
RTS     NAMES
3bea857310d6   ubuntu:22.04   "bash"    14 minutes ago   Exited (137) About a minute ago
        layer-test
```

Writable Layer Deletion After Container Removal



- UpperDir no longer exists after docker rm

Final Overlay2 Comparison



Analysis

- Overlay2 is Docker's default storage driver that uses the Linux OverlayFS to create a union mount of multiple filesystem layers.

Component	Type	Purpose	Persistence
LowerDir	Read-only	Image layers (shared)	Permanent
UpperDir	Read-write	Container changes	Container lifetime
WorkDir	Internal	OverlayFS operations	Container lifetime
MergedDir	Union mount	Unified view	Container lifetime

Copy-on-Write (CoW) Mechanism:

1. **File Read:** If file exists in lower layers, read directly

2. **File Modification:**
    - File copied from LowerDir to UpperDir
    - Modifications applied to UpperDir copy
    - Original in LowerDir remains unchanged
3. **New File Creation:** Directly written to UpperDir
4. **File Deletion:** Whiteout file created in UpperDir (hides lower layer file)

Observations:

1. **Image Layer Sharing:**
    - Multiple containers from same image share read-only layers
    - Saves disk space and speeds up container creation
2. **Container Isolation:**
    - Each container gets unique writable layer
    - Changes isolated from other containers
3. **Data Lifecycle:**
    - **Running:** Files in UpperDir accessible
    - **Stopped:** UpperDir preserved
    - **Removed:** UpperDir deleted (data lost unless in volume)
4. **Performance Implications:**
    - First write to file incurs CoW overhead
    - Subsequent writes to same file are fast
    - Heavy I/O workloads benefit from volumes

Conclusion:

Docker's layered filesystem is fundamental to container efficiency.
The **overlay2 driver** creates a union of read-only image layers and a writable container layer, enabling:

- **Fast container startup** (no data duplication)
- **Efficient storage** (shared base layers)

- **Isolated changes** (per-container writable layer)
- **Data lifecycle management** (layers persist only as needed)

Task 8: **Process Creation Flow Examination**

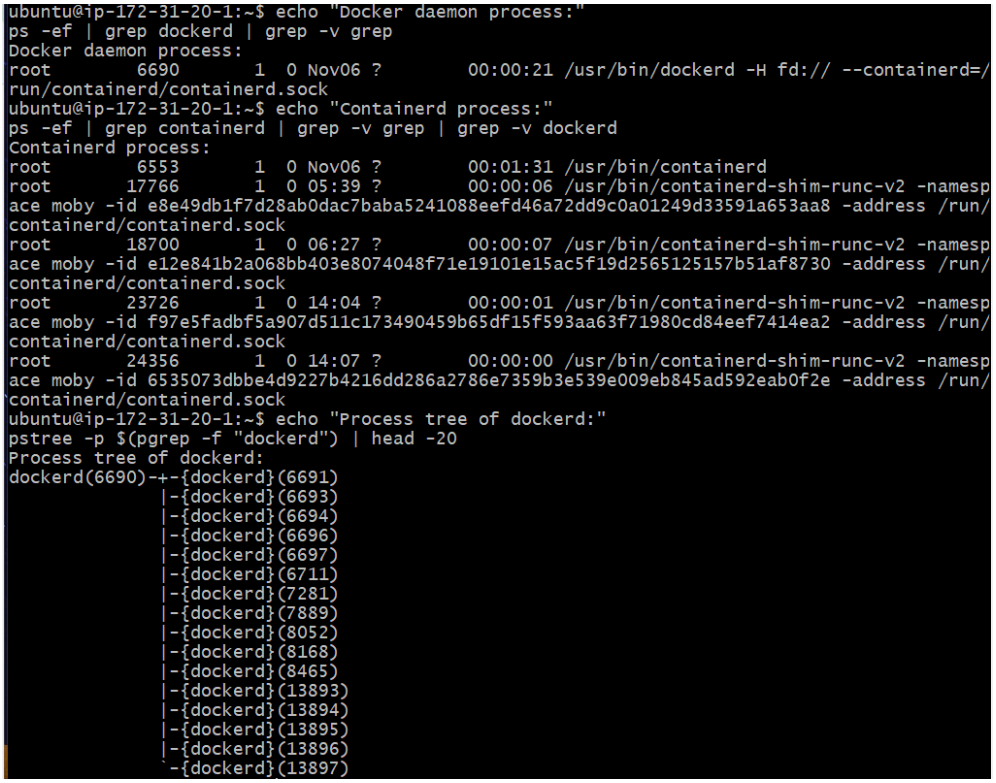
Objective:

Trace the complete process ancestry from the host Docker daemon through containerd and runc to the container's main process, identifying the role of each component in the container creation flow.

Methodology:

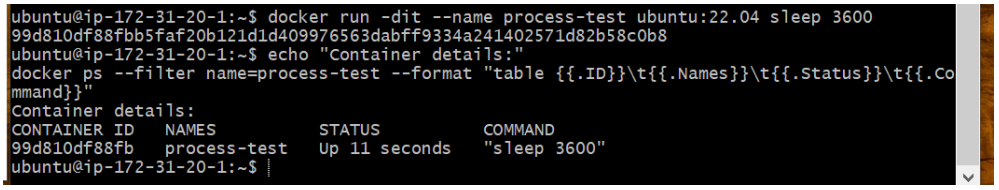
1. Identified Docker daemon (dockerd) and container runtime (containerd) processes
2. Created test container running sleep 3600
3. Retrieved container's host PID using docker inspect
4. Traced process ancestry using ps -ef and parent PID traversal
5. Identified key components: dockerd, containerd, containerd-shim, and runc
6. Visualized process tree using pstree
7. Examined namespace isolation as evidence of clone() system call
8. Compared process view from inside container vs. host
9. Created multiple containers to demonstrate shared infrastructure pattern
10. Documented complete process creation flow

Docker Daemon and Core Processes

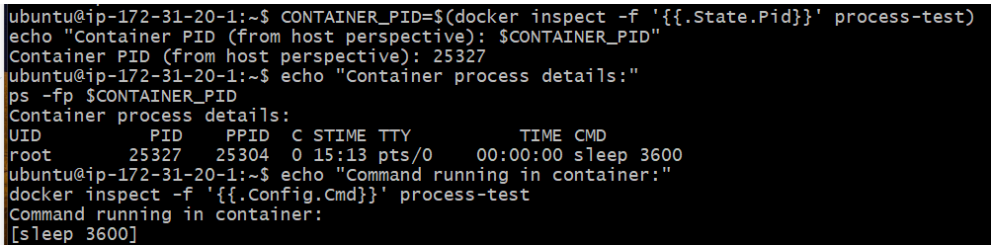


- dockerd and containerd running as system services

Test Container Creation



Container PID Identification



- Container process visible from host with specific PID

Process Ancestry Chain

```
ubuntu@ip-172-31-20-1:~$ ^C
ubuntu@ip-172-31-20-1:~$ CURRENT_PID=$CONTAINER_PID
echo "Tracing ancestry:"
for i in {1..10}; do
    if [ "$CURRENT_PID" -eq 1 ] || [ -z "$CURRENT_PID" ]; then
        break
    fi

    P_PID=$(ps -o ppid= -p $CURRENT_PID | tr -d ' ')
    PNAME=$(ps -o comm= -p $CURRENT_PID)
    echo "PID: $CURRENT_PID | Name: $PNAME | Parent PID: $P_PID"

    CURRENT_PID=$P_PID
done
Tracing ancestry:
PID: 25327 | Name: sleep | Parent PID: 25304
PID: 25304 | Name: containerd-shim | Parent PID: 1
```

- Parent-child relationship traced from container to init

Key Process Components

```
ubuntu@ip-172-31-20-1:~$ echo "1. DOCKERD (Docker Daemon):"
echo "   Role: High-level container management, API, image management"
DOCKERD_PID=$(pgrep -f "dockerd" | head -1)
ps -fp $DOCKERD_PID
1. DOCKERD (Docker Daemon):
   Role: High-level container management, API, image management
UID          PID    PPID  C STIME TTY          TIME CMD
root        6690       1  0 Nov06 ?        00:00:21 /usr/bin/dockerd -H fd:// --containerd=/
ubuntu@ip-172-31-20-1:~$ echo "2. CONTAINERD (Container Runtime):"
echo "   Role: Container lifecycle management, image distribution"
CONTAINERD_PID=$(pgrep -x containerd | head -1)
ps -fp $CONTAINERD_PID
2. CONTAINERD (Container Runtime):
   Role: Container lifecycle management, image distribution
UID          PID    PPID  C STIME TTY          TIME CMD
root        6553       1  0 Nov06 ?        00:01:31 /usr/bin/containerd
ubuntu@ip-172-31-20-1:~$ echo "   Role: Decouples running container from containerd, allows
daemon restart"
CONTAINER_ID=$(docker inspect -f '{{.Id}}' process-test)
SHIM_PID=$(pgrep -f "containerd-shim.*$CONTAINER_ID" | head -1)
if [ -z "$SHIM_PID" ]; then
    SHIM_PID=$(pgrep -f "containerd-shim" | head -1)
fi
ps -fp $SHIM_PID 2>/dev/null || echo "Shim process (integrated in newer versions)"
   Role: Decouples running container from containerd, allows daemon restart
UID          PID    PPID  C STIME TTY          TIME CMD
root       25304       1  0 15:13 ?        00:00:00 /usr/bin/containerd-shim-runc-v2 -namesp
ubuntu@ip-172-31-20-1:~$ echo "4. CONTAINER PROCESS (sleep 3600):"
CONTAINER_PID=$(docker inspect -f '{{.State.Pid}}' process-test)
ps -fp $CONTAINER_PID
4. CONTAINER PROCESS (sleep 3600):
UID          PID    PPID  C STIME TTY          TIME CMD
root       25327   25304  0 15:13 pts/0    00:00:00 sleep 3600
```

Process Tree Visualization

```
ubuntu@ip-172-31-20-1:~$ CONTAINER_PID=$(docker inspect -f '{{.State.Pid}}' process-test)
ubuntu@ip-172-31-20-1:~$ echo "Process tree from container process:"
pstree -p -s $CONTAINER_PID
Process tree from container process:
systemd(1)───containerd-shim(25304)───sleep(25327)
ubuntu@ip-172-31-20-1:~$ echo "Detailed process tree showing Docker infrastructure:"
DOCKERD_PID=$(pgrep -f "dockerd" | head -1)
pstree -p $DOCKERD_PID | grep -A 5 -B 5 $CONTAINER_PID || pstree -p $DOCKERD_PID | head -30
Detailed process tree showing Docker infrastructure:
dockerd(6690)-+-{dockerd}(6691)
              |-{dockerd}(6693)
              |-{dockerd}(6694)
              |-{dockerd}(6696)
              |-{dockerd}(6697)
              |-{dockerd}(6711)
              |-{dockerd}(7281)
              |-{dockerd}(7889)
              |-{dockerd}(8052)
              |-{dockerd}(8168)
              |-{dockerd}(8465)
              |-{dockerd}(13893)
              |-{dockerd}(13894)
              |-{dockerd}(13895)
              |-{dockerd}(13896)
              `-{dockerd}(13897)
```

runc Role and Runtime Information

```
ubuntu@ip-172-31-20-1:~$ echo "runc binary location:"
which runc
runc --version
runc binary location:
/usr/bin/runc
runc version 1.3.3
commit: v1.3.3-0-gd842d771
spec: 1.2.1
go: go1.24.9
libseccomp: 2.5.5
ubuntu@ip-172-31-20-1:~$ ps -ef | grep runc | grep -v grep || echo "runc already exited"
root       17766       1  0 05:39 ?        00:00:06 /usr/bin/containerd-shim-runc-v2 -namesp
ace moby -id e8e49db1f7d28ab0dac7baba5241088eefd46a72dd9c0a01249d33591a653aa8 -address /run/
containerd/containerd.sock
root       18700       1  0 06:27 ?        00:00:07 /usr/bin/containerd-shim-runc-v2 -namesp
ace moby -id e12e841b2a068bb403e8074048f71e19101e15ac5f19d2565125157b51af8730 -address /run/
containerd/containerd.sock
root       23726       1  0 14:04 ?        00:00:01 /usr/bin/containerd-shim-runc-v2 -namesp
ace moby -id f97e5fadbf5a907d511c173490459b65df15f593aa63f71980cd84eef7414ea2 -address /run/
containerd/containerd.sock
root       24356       1  0 14:07 ?        00:00:00 /usr/bin/containerd-shim-runc-v2 -namesp
ace moby -id 6535073dbbe4d9227b4216dd286a2786e7359b3e539e009eb845ad592eab0f2e -address /run/
containerd/containerd.sock
root       25304       1  0 15:13 ?        00:00:00 /usr/bin/containerd-shim-runc-v2 -namesp
ace moby -id 99d810df88fbb5faf20b121d1d409976563dabff9334a241402571d82b58c0b8 -address /run/
containerd/containerd.sock
ubuntu@ip-172-31-20-1:~$ echo "Container runtime information:"
docker info | grep -i runtime -A 5
Container runtime information:
 Runtimes: runc io.containerd.runc.v2
 Default Runtime: runc
 Init Binary: docker-init
 containerd version: 442cb34bda9a6a0fed82a2ca7cade05c5c749582
 runc version: v1.3.3-0-gd842d771
 init version: de40ad0
 Security Options:
ubuntu@ip-172-31-20-1:~$
```

- runc version and runtime configuration

Namespace Isolation Evidence

```
ubuntu@ip-172-31-20-1:~$ CONTAINER_PID=$(docker inspect -f '{{.State.Pid}}' process-test)
ubuntu@ip-172-31-20-1:~$ echo "Container process namespaces:"
sudo ls -la /proc/$CONTAINER_PID/ns/
Container process namespaces:
total 0
dr-x--x--x 2 root root 0 Nov  7 15:13 .
dr-xr-xr-x 9 root root 0 Nov  7 15:13 ..
lrwxrwxrwx 1 root root 0 Nov  7 15:21 cgroup -> 'cgroup:[4026532490]'
lrwxrwxrwx 1 root root 0 Nov  7 15:21 ipc -> 'ipc:[4026532488]'
lrwxrwxrwx 1 root root 0 Nov  7 15:13 mnt -> 'mnt:[4026532422]'
lrwxrwxrwx 1 root root 0 Nov  7 15:13 net -> 'net:[4026532491]'
lrwxrwxrwx 1 root root 0 Nov  7 15:21 pid -> 'pid:[4026532489]'
lrwxrwxrwx 1 root root 0 Nov  7 15:24 pid_for_children -> 'pid:[4026532489]'
lrwxrwxrwx 1 root root 0 Nov  7 15:21 time -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Nov  7 15:24 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Nov  7 15:21 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 Nov  7 15:21 uts -> 'uts:[4026532487]'
ubuntu@ip-172-31-20-1:~$ echo "Host init (PID 1) namespaces for comparison:"
sudo ls -la /proc/1/ns/
Host init (PID 1) namespaces for comparison:
total 0
dr-x--x--x 2 root root 0 Nov  6 11:32 .
dr-xr-xr-x 9 root root 0 Nov  6 11:25 ..
lrwxrwxrwx 1 root root 0 Nov  6 17:11 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 Nov  6 17:11 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 root root 0 Nov  6 17:11 mnt -> 'mnt:[4026531841]'
lrwxrwxrwx 1 root root 0 Nov  6 17:11 net -> 'net:[4026531840]'
lrwxrwxrwx 1 root root 0 Nov  6 11:32 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 Nov  7 15:24 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 Nov  6 17:11 time -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Nov  7 15:24 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Nov  6 17:11 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 Nov  6 17:11 uts -> 'uts:[4026531838]'
```

- Different namespace inodes proving clone() system call

Multiple Containers Pattern

```
ubuntu@ip-172-31-20-1:~$ docker ps --filter name=process-test --format "table {{.Names}}\t{{
.ID}}\t{{.Status}}"
NAMES          CONTAINER ID   STATUS
process-test-3   38c5ebff8226   Up About a minute
process-test-2   3f696d9fbd1d   Up About a minute
process-test     99d810df88fb   Up 16 minutes
ubuntu@ip-172-31-20-1:~$ DOCKERD_PID=$(pgrep -f "dockerd" | head -1)
pstree -p $DOCKERD_PID | grep -E "sleep|containerd|shim" | head -20
echo "Container 1: $(docker inspect -f '{{.State.Pid}}' process-test)"
echo "Container 2: $(docker inspect -f '{{.State.Pid}}' process-test-2)"
echo "Container 3: $(docker inspect -f '{{.State.Pid}}' process-test-3)"
head: cannot open 'echo' for reading: No such file or directory
head: cannot open 'Container 1: 25327' for reading: No such file or directory
Container 2: 26316
Container 3: 26383
```

Container Internal vs External View

```
ubuntu@ip-172-31-20-1:~$ docker exec process-test ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.1   2792  1500 pts/0    Ss+  15:13   0:00 sleep 3600
root           7  0.0  0.2   7064  2776 ?        Rs   15:32   0:00 ps aux
ubuntu@ip-172-31-20-1:~$ docker exec process-test bash -c "apt-get update -qq && apt-get ins
tall -y psmisc -qq 2>/dev/null && pstree -p" 2>/dev/null || \
docker exec process-test ps -ef
Selecting previously unselected package psmisc.
(Reading database ... 4393 files and directories currently installed.)
Preparing to unpack .../psmisc_23.4-2build3_amd64.deb ...
Unpacking psmisc (23.4-2build3) ...
Setting up psmisc (23.4-2build3) ...
sleep(1)
ubuntu@ip-172-31-20-1:~$ CONTAINER_PID=$(docker inspect -f '{{.State.Pid}}' process-test)
ps -fp $CONTAINER_PID
UID          PID    PPID  C STIME TTY          TIME CMD
root       25327   25304  0 15:13 pts/0    00:00:00 sleep 3600
```

- Same process with different PID in different namespaces

Analysis:

Component Roles:

Component	Type	Responsibilities	Lifecycle
dockerd	Daemon	API server, image management, high-level orchestration	Always running
containerd	Daemon	Container lifecycle, image distribution, runtime supervision	Always running
containerd-shim	Per-container	Daemonless containers, I/O forwarding, exit status	Container lifetime
runc	Ephemeral	Low-level runtime, namespace/cgroup setup, process execution	Seconds (exits after start)
Container Process	Application	User workload (e.g., sleep, nginx, app)	Until stopped/crashed

Observations:

1. **Layered Architecture:**
   - **High-level (dockerd):** User interface, orchestration
   - **Mid-level (containerd):** Container lifecycle, standards
   - **Low-level (runc):** Linux primitives (namespaces, cgroups)
2. **Ephemeral runc:**
   - runc exists only during container creation (~100-500ms)
   - Exits after handing off process to containerd-shim
   - Allows for lightweight, stateless operation
3. **Daemonless Containers:**
   - containerd-shim enables containers to survive daemon restarts
   - Docker/containerd can be upgraded without killing containers
   - Shim acts as "init" from host perspective
4. **PID Namespace Isolation:**
   - Same process, different PID namespaces
   - Container process thinks it's PID 1 (init)

- Host sees actual PID in global namespace

5. **clone() System Call Evidence:**
    - Different inode numbers prove separate namespaces
    - Created by clone() with CLONE_NEWPID flag

Conclusion:

Docker's process creation flow is a **layered architecture** where:

- **dockerd** provides high-level API and orchestration
- **containerd** manages container lifecycle following OCI standards
- **containerd-shim** enables daemonless containers
- **runc** executes low-level Linux primitives (clone(), namespaces, cgroups)
- **Container process** runs isolated in its own namespaces as PID 1

Task 9: **Comparative Namespace Experiment**

Objective:

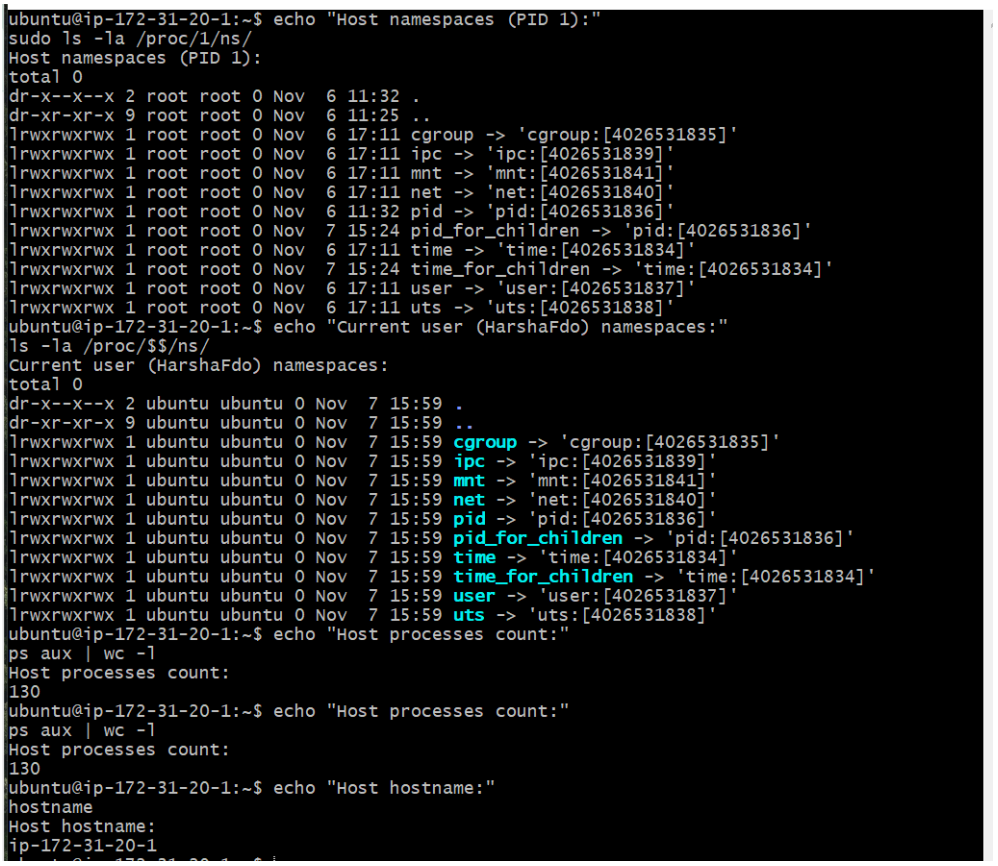
Compare container isolation by running containers with default namespaces versus containers sharing host namespaces (--pid=host, --network=host), and explain the difference between clone() and setns() system calls in namespace management.

Methodology:

1. Established host namespace baseline
2. Created container with **default isolation** (all namespaces isolated)
3. Inspected isolated container's namespaces using /proc/[pid]/ns/
4. Tested process visibility and network stack in isolated container
5. Created container with **-pid=host** (shared PID namespace)
6. Verified process visibility in PID-shared container
7. Compared PID namespace inodes to confirm sharing
8. Created container with **-network=host** (shared network namespace)
9. Tested network stack accessibility in network-shared container

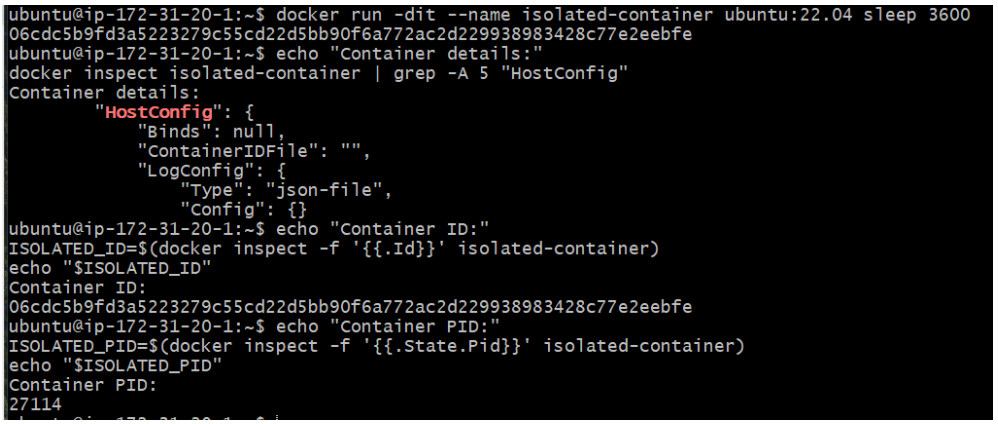
10. Explained clone() vs etns() system calls

Host Namespace Baseline



- Host namespaces, processes, and network interfaces recorded

Isolated Container Creation



Isolated Container Namespace Comparison

```
total 0
dr-x--x--x 2 root root 0 Nov  7 16:01 .
dr-xr-xr-x 9 root root 0 Nov  7 16:01 ..
lrwxrwxrwx 1 root root 0 Nov  7 16:02 cgroup -> 'cgroup:[4026532490]'
lrwxrwxrwx 1 root root 0 Nov  7 16:02 ipc -> 'ipc:[4026532488]'
lrwxrwxrwx 1 root root 0 Nov  7 16:01 mnt -> 'mnt:[4026532422]'
lrwxrwxrwx 1 root root 0 Nov  7 16:01 net -> 'net:[4026532491]'
lrwxrwxrwx 1 root root 0 Nov  7 16:02 pid -> 'pid:[4026532489]'
lrwxrwxrwx 1 root root 0 Nov  7 16:02 pid_for_children -> 'pid:[4026532489]'
lrwxrwxrwx 1 root root 0 Nov  7 16:02 time -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Nov  7 16:02 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Nov  7 16:02 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 Nov  7 16:02 uts -> 'uts:[4026532487]'
ubuntu@ip-172-31-20-1:~$ for ns in ipc mnt net pid uts cgroup; do
    HOST_NS=$(sudo readlink /proc/1/ns/$ns)
    CONTAINER_NS=$(sudo readlink /proc/$ISOLATED_PID/ns/$ns)

    if [ "$HOST_NS" = "$CONTAINER_NS" ]; then
        echo "[$ns] SHARED: $HOST_NS"
    else
        echo "[$ns] ISOLATED:"
        echo "  Host:      $HOST_NS"
        echo "  Container: $CONTAINER_NS"
    fi
    echo ""
done
[ipc] ISOLATED:
  Host:      ipc:[4026531839]
  Container: ipc:[4026532488]

[mnt] ISOLATED:
  Host:      mnt:[4026531841]
  Container: mnt:[4026532422]

[net] ISOLATED:
  Host:      net:[4026531840]
  Container: net:[4026532491]

[pid] ISOLATED:
  Host:      pid:[4026531836]
  Container: pid:[4026532489]

[uts] ISOLATED:
  Host:      uts:[4026531838]
  Container: uts:[4026532487]

[cgroup] ISOLATED:
  Host:      cgroup:[4026531835]
  Container: cgroup:[4026532490]
```

- All namespaces show different inode numbers from host

Process Visibility in Isolated Container

```
ubuntu@ip-172-31-20-1:~$ echo "Processes visible from HOST:"
ps aux | head -15
HOST_PROCESS_COUNT=$(ps aux | wc -l)
Processes visible from HOST:
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  1.4  22660 13772 ?        Ss   Nov06   0:08 /sbin/init
root           2  0.0  0.0      0     0 ?        S    Nov06   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S    Nov06   0:00 [pool_workqueue_release]
root           4  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-rcu_gp]
root           5  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-sync_wq]
root           6  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-kvfree_rcu_rec
laim]
root           7  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-slub_flushwq]
root           8  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-netns]
root          11  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/0:0H-events_high
pri]
root          13  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-mm_percpu_wq]
root          14  0.0  0.0      0     0 ?        I    Nov06   0:00 [rcu_tasks_rude_kthread]
root          15  0.0  0.0      0     0 ?        I    Nov06   0:00 [rcu_tasks_trace_kthread]
root          16  0.0  0.0      0     0 ?        S    Nov06   0:00 [ksoftirqd/0]
root          17  0.0  0.0      0     0 ?        I    Nov06   0:02 [rcu_sched]
ubuntu@ip-172-31-20-1:~$ echo "Processes visible from ISOLATED CONTAINER:"
docker exec isolated-container ps aux
CONTAINER_PROCESS_COUNT=$(docker exec isolated-container ps aux | wc -l)
Processes visible from ISOLATED CONTAINER:
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.1   2792  1420 pts/0    Ss+  16:01   0:00 sleep 3600
root           6  0.0  0.3   7064  2992 ?        Rs   16:06   0:00 ps aux
ubuntu@ip-172-31-20-1:~$ echo "  Host can see $HOST_PROCESS_COUNT processes"
echo "  Container can only see $CONTAINER_PROCESS_COUNT processes (its own)"
  Host can see 136 processes
  Container can only see 3 processes (its own)
```

- Container can only see its own processes (2-5 processes)
- Host can see 100+ processes

Network Isolation in Isolated Container

```
ubuntu@ip-172-31-20-1:~$ echo "Host network interfaces:"
ip addr show | grep -E "^[0-9]+:"
HOST_INTERFACE_COUNT=$(ip addr show | grep -E "^[0-9]+:" | wc -l)
echo "Total host interfaces: $HOST_INTERFACE_COUNT"
Host network interfaces:
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
2: ens5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP group default qlen 100
0
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
44: veth9827e8b@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0
 state UP group default
49: vethc58977a@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0
 state UP group default
52: veth0041fc7@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0
 state UP group default
53: vethac31c35@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0
 state UP group default
66: vethc5580a2@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0
 state UP group default
Total host interfaces: 8
ubuntu@ip-172-31-20-1:~$ echo "Container network interfaces:"
docker exec isolated-container ip addr show 2>/dev/null || \
docker exec isolated-container cat /proc/net/dev
Container network interfaces:
OCI runtime exec failed: exec failed: unable to start container process: exec: "ip": executa
ble file not found in $PATH: unknown
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop
 fifo colls carrier compressed
    lo:       0       0    0    0    0     0          0         0        0       0    0    0
    0     0       0             0
  eth0:     936      12    0    0    0     0          0         0      126       3    0    0
    0     0       0             0
ubuntu@ip-172-31-20-1:~$ echo "Container IP address:"
docker inspect isolated-container -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end
}}'
Container IP address:
172.17.0.6
ubuntu@ip-172-31-20-1:~$ echo "Host IP address:"
hostname -I
Host IP address:
172.31.20.1 172.17.0.1
```

- Container has isolated network stack (lo, eth0)

- Different IP address from host

PID-Shared Container Creation

```
ubuntu@ip-172-31-20-1:~$ docker run -dit --name pid-shared-container --pid=host ubuntu:22.04
 sleep 3600
c7fdd252cb9c50a9cd7af5f5d5ca2ac28226719c81f9353faa71a96eb9a83caf
ubuntu@ip-172-31-20-1:~$ echo "Container created with --pid=host"
echo "Container configuration:"
docker inspect pid-shared-container | grep -i "pidmode"
Container created with --pid=host
Container configuration:
            "PidMode": "host",
ubuntu@ip-172-31-20-1:~$ echo "Container PID:"
PID_SHARED_PID=$(docker inspect -f '{{.State.Pid}}' pid-shared-container)
echo "$PID_SHARED_PID"
Container PID:
27436
```

Process Visibility in PID-Shared Container

```
HOST_COUNT=$(ps aux | wc -l)
echo "Total: $HOST_COUNT processes"
Processes visible from HOST:
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  1.4  22660 13772 ?        Ss   Nov06   0:08 /sbin/init
root           2  0.0  0.0      0     0 ?        S    Nov06   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S    Nov06   0:00 [pool_workqueue_release]
root           4  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-rcu_gp]
root           5  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-sync_wq]
root           6  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-kvfree_rcu_rec
laim]
root           7  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-slub_flushwq]
root           8  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-netns]
root          11  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/0:0H-events_high
pri]
root          13  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-mm_percpu_wq]
root          14  0.0  0.0      0     0 ?        I    Nov06   0:00 [rcu_tasks_rude_kthread]
root          15  0.0  0.0      0     0 ?        I    Nov06   0:00 [rcu_tasks_trace_kthread]
root          16  0.0  0.0      0     0 ?        S    Nov06   0:00 [ksoftirqd/0]
root          17  0.0  0.0      0     0 ?        I    Nov06   0:02 [rcu_sched]
Total: 135 processes
ubuntu@ip-172-31-20-1:~$ echo "Processes visible from PID-SHARED CONTAINER:"
docker exec pid-shared-container ps aux | head -15
SHARED_COUNT=$(docker exec pid-shared-container ps aux | wc -l)
echo "Total: $SHARED_COUNT processes"
Processes visible from PID-SHARED CONTAINER:
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  1.4  22660 13772 ?        Ss   Nov06   0:08 /sbin/init
root           2  0.0  0.0      0     0 ?        S    Nov06   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S    Nov06   0:00 [pool_workqueue_release]
root           4  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-rcu_gp]
root           5  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-sync_wq]
root           6  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-kvfree_rcu_rec
laim]
root           7  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-slub_flushwq]
root           8  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-netns]
root          11  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/0:0H-events_high
pri]
root          13  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-mm_percpu_wq]
root          14  0.0  0.0      0     0 ?        I    Nov06   0:00 [rcu_tasks_rude_kthread]
root          15  0.0  0.0      0     0 ?        I    Nov06   0:00 [rcu_tasks_trace_kthread]
root          16  0.0  0.0      0     0 ?        S    Nov06   0:00 [ksoftirqd/0]
root          17  0.0  0.0      0     0 ?        I    Nov06   0:02 [rcu_sched]
Total: 136 processes
ubuntu@ip-172-31-20-1:~$ echo "  Host: $HOST_COUNT processes"
echo "  PID-shared container: $SHARED_COUNT processes"
  Host: 135 processes
  PID-shared container: 136 processes
```

- Container can see ALL host processes
- Same process count as host

PID Namespace Verification

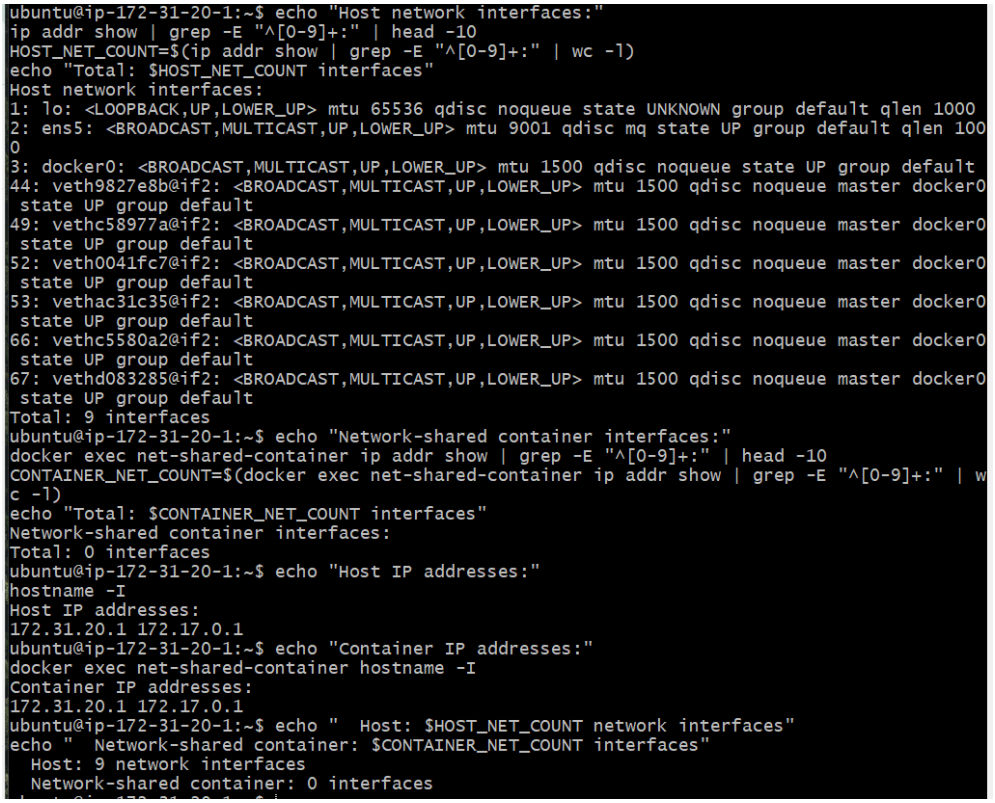
```
ubuntu@ip-172-31-20-1:~$ PID_SHARED_PID=$(docker inspect -f '{{.State.Pid}}' pid-shared-cont
ainer)
ubuntu@ip-172-31-20-1:~$ echo "Host PID namespace (PID 1):"
HOST_PID_NS=$(sudo readlink /proc/1/ns/pid)
echo "$HOST_PID_NS"
Host PID namespace (PID 1):
pid:[4026531836]
ubuntu@ip-172-31-20-1:~$ echo "PID-shared container namespace:"
CONTAINER_PID_NS=$(sudo readlink /proc/$PID_SHARED_PID/ns/pid)
echo "$CONTAINER_PID_NS"
PID-shared container namespace:
pid:[4026531836]
ubuntu@ip-172-31-20-1:~$ if [ "$HOST_PID_NS" = "$CONTAINER_PID_NS" ]; then
    echo "SAME NAMESPACE - Container joined host's PID namespace!"
    echo "   This was done using setns() system call"
else
    echo "Different namespaces"
fi
SAME NAMESPACE - Container joined host's PID namespace!
   This was done using setns() system call
ubuntu@ip-172-31-20-1:~$ echo "Other namespaces still isolated:"
for ns in net mnt uts; do
    HOST_NS=$(sudo readlink /proc/1/ns/$ns)
    CONTAINER_NS=$(sudo readlink /proc/$PID_SHARED_PID/ns/$ns)

    if [ "$HOST_NS" = "$CONTAINER_NS" ]; then
        echo "[$ns] SHARED"
    else
        echo "[$ns] ISOLATED"
    fi
done
Other namespaces still isolated:
[net] ISOLATED
[mnt] ISOLATED
[uts] ISOLATED
```

- Host and container share same PID namespace inode
- Other namespaces remain isolated

Network-Shared Container Creation

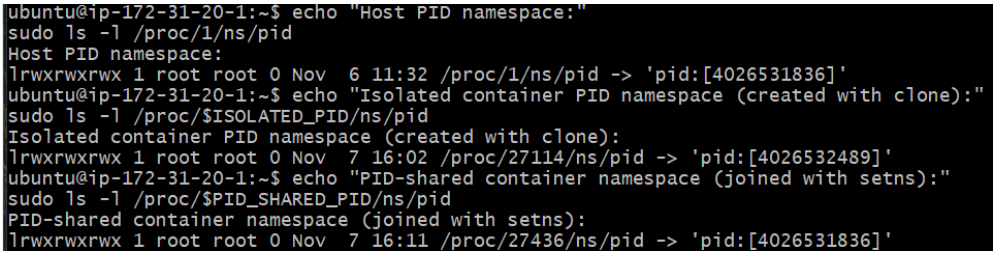
```
ubuntu@ip-172-31-20-1:~$ docker run -dit --name net-shared-container --network=host ubuntu:2
2.04 sleep 3600
0fd4c2b0c626e5a4074b952ffbea3df782c7d5f9722da56d67e81e9bc560e60f
ubuntu@ip-172-31-20-1:~$ echo "Container network configuration:"
docker inspect net-shared-container | grep -i "networkmode"
Container network configuration:
            "NetworkMode": "host",
ubuntu@ip-172-31-20-1:~$ echo "Container details:"
docker ps --filter name=net-shared-container --format "table {{.Names}}\t{{.Ports}}\t{{.Netw
orks}}"
Container details:
NAMES                PORTS     NETWORKS
net-shared-container           host
```

Network Stack in Network-Shared Container



- Container can see all host network interfaces
- Same IP addresses as host

setns() vs clone() Demonstration



Analysis:

Namespace Types and Their Functions:

Namespace	Isolates	Default Container	--pid=host	--network=host
PID	Process IDs	Isolated	Shared	Isolated
NET	Network stack	Isolated	Isolated	Shared
MNT	Filesystems	Isolated	Isolated	Isolated

Namespace	Isolates	Default Container	--pid=host	--network=host
UTS	Hostname	Isolated	Isolated	Isolated
IPC	Inter-process comm	Isolated	Isolated	Isolated
USER	User/group IDs	Isolated	Isolated	Isolated
CGROUP	Cgroup root	Isolated	Isolated	Isolated

Comparison Table:

Aspect	clone()	setns()
Purpose	Create new namespace	Join existing namespace
Inode	New unique inode	Same inode as target
Isolation	Complete isolation	Shared resources
When Used	Default container creation	--pid=host, --network=host, docker exec
Visibility	Cannot see parent resources	Can see all namespace resources
Security	High (isolated)	Lower (shared)
Performance	Overhead of new namespace	No creation overhead

Conclusion:

Container isolation is achieved through Linux namespaces, which can be **created** with clone() or **joined** with setns():

- **clone()** with CLONE_NEW* flags creates brand new, isolated namespaces for default containers, providing strong security boundaries
- **setns()** allows processes to join existing namespaces (like the host's), enabling selective sharing for specific use cases like monitoring (--pid=host) or high-performance networking (--network=host)

This flexibility allows Docker to balance between:

- **Security** (default isolation)
- **Functionality** (debugging, monitoring)
- **Performance** (avoiding network overhead)