# 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
```

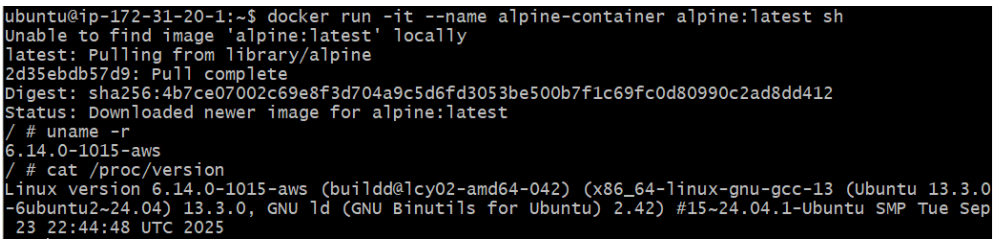
### Ubuntu container - 22.04

```
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 kernel |

# Task 2: **Process and PID Mapping**

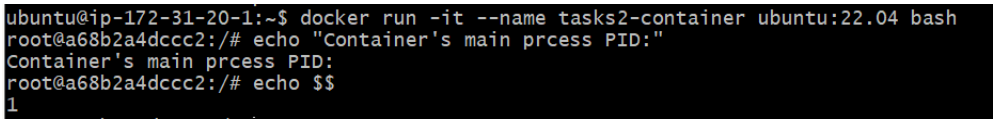
## Objective:

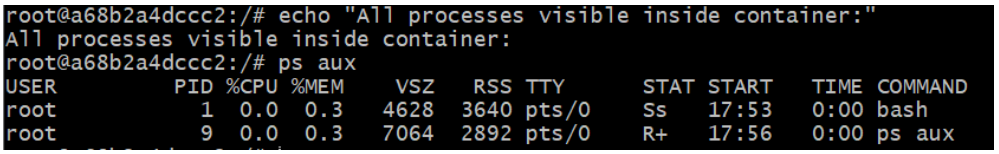
Demonstrate that a container's processes are regular host processes.

## 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.

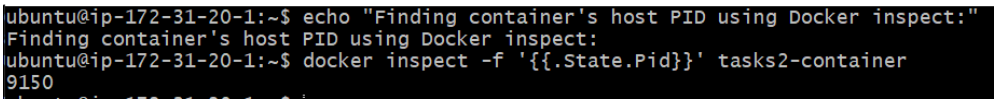
## Container’s Internal View

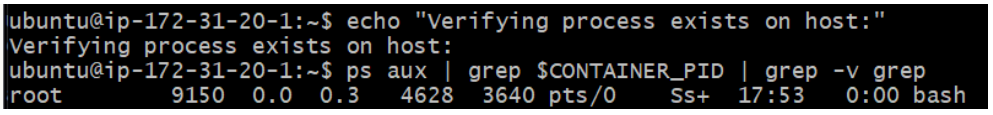



The container’s bash process reports PID: 1

## 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

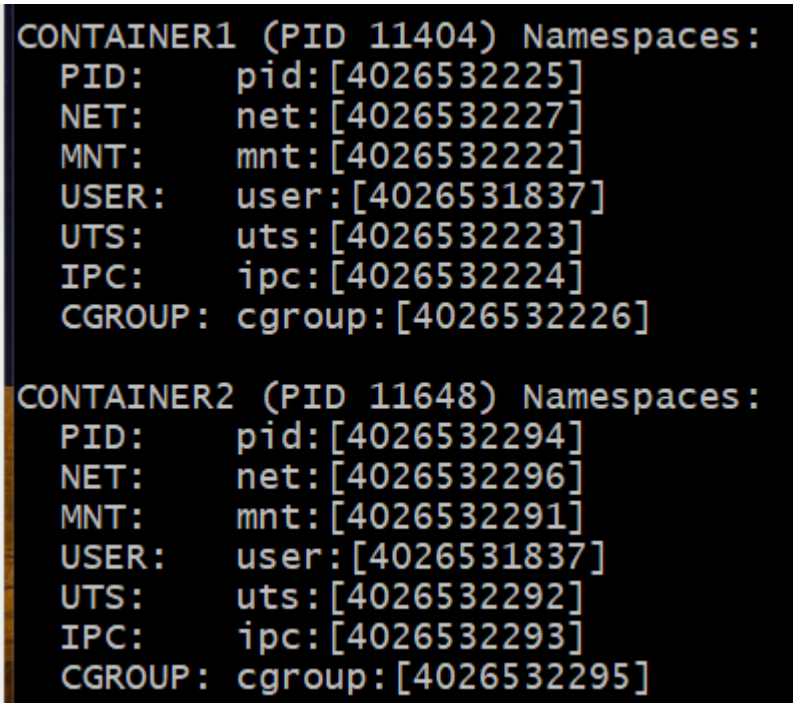
```
sudo lsns -p $CONTAINER1_PID

        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=0xc000078000, 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

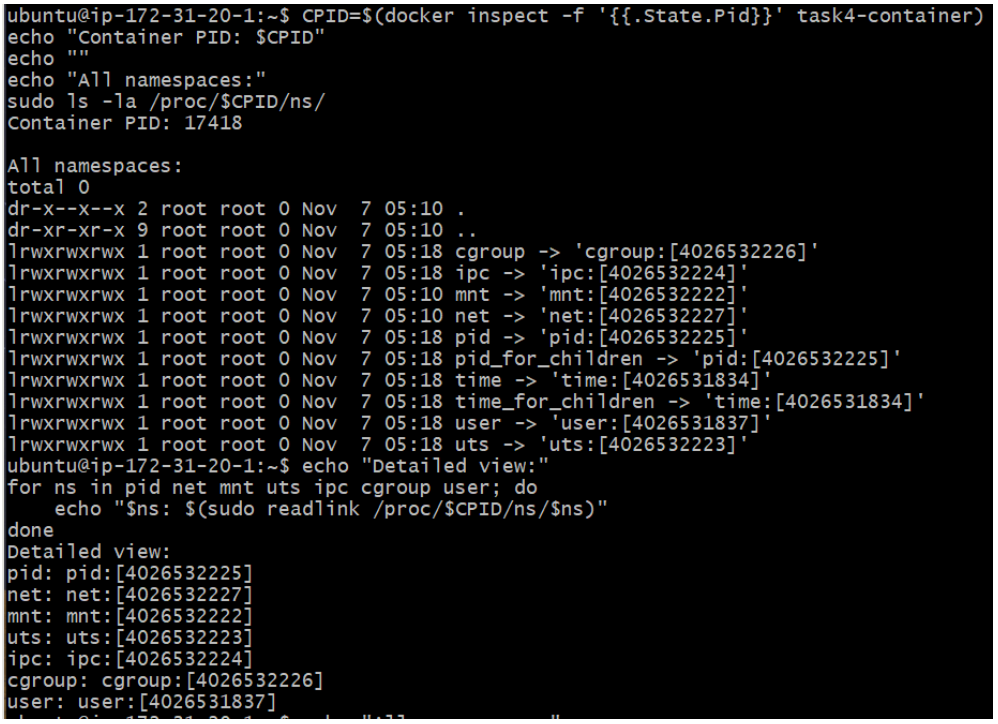
## Namespace Flags Detected

```
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:~$ CGROUP_PATH=$(cat /proc/$(docker inspect -f '{{.State.P
id}}' cgroup-test)/cgroup | cut -d: -f3)
PARENT_PATH=$(dirname $CGROUP_PATH)
ubuntu@ip-172-31-20-1:~$ cat /sys/fs/cgroup${PARENT_PATH}/cgroup.subtree_control
cpuset cpu io memory hugetlb pids rdma misc dmem
```

- 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 therefore container cannot exceed limits

# 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

**stress-ng** is a Linux stress testing tool that can exercise various system resources in a controlled manner. It allows us to:

- Generate precise CPU load (--cpu workers)
- Allocate specific amounts of memory (--vm workers, --vm-bytes)
- Test I/O, network, and other subsystems
- Set timeout durations for tests

We used stress-ng because it provides:

- Predictable resource consumption patterns
- Configurable stress levels
- Clean termination after timeout
- Detailed output for analysis

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

Command: stress-ng --vm 1 --vm-bytes 1G --timeout 30s

## 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
```

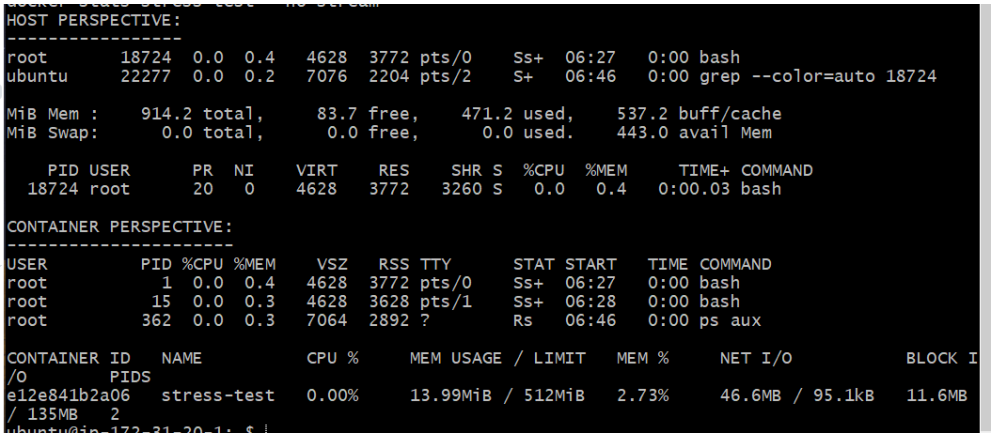
- Memory climbed to 100% of limit
- Process killed or severe slowdown

## 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

### stress-ng commands:

- cpu stress: stress-ng --cpu 2 --timeout 30s
- Memory stress: stress-ng —vm1 —vm-bytes 1g —timeout 30s

### 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:

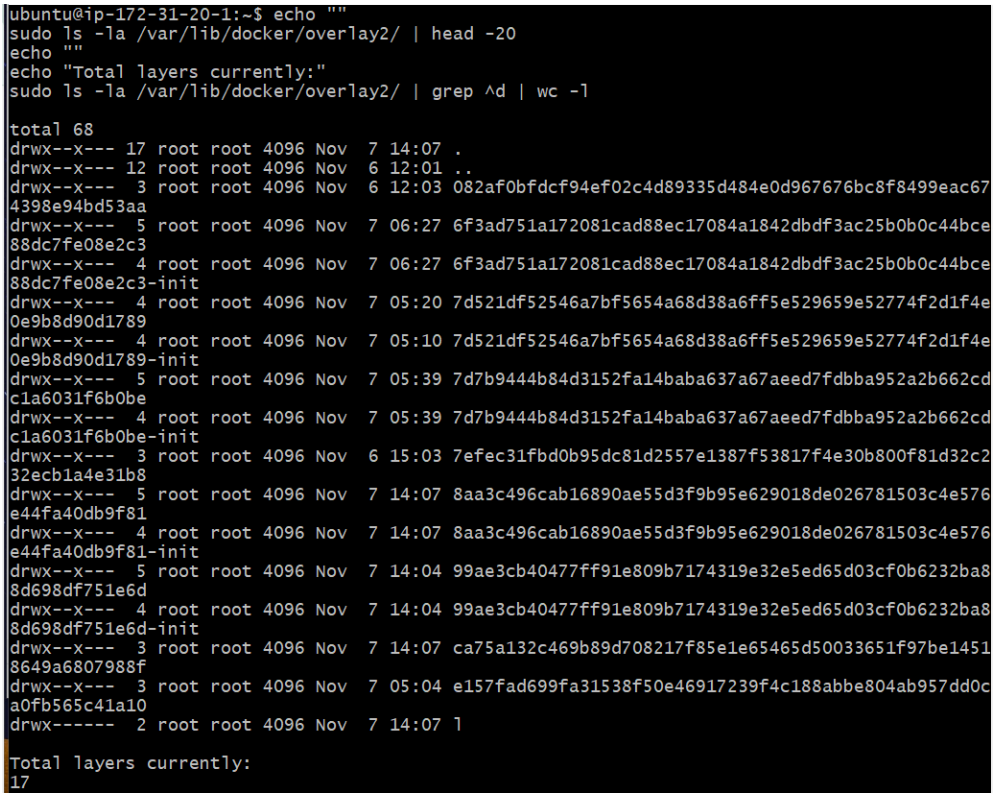
Investigate Docker's overlay2 filesystem to understand the relationship between read-only image layers and writable container layers, and demonstrate the lifecycle of container filesystem changes.

## 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

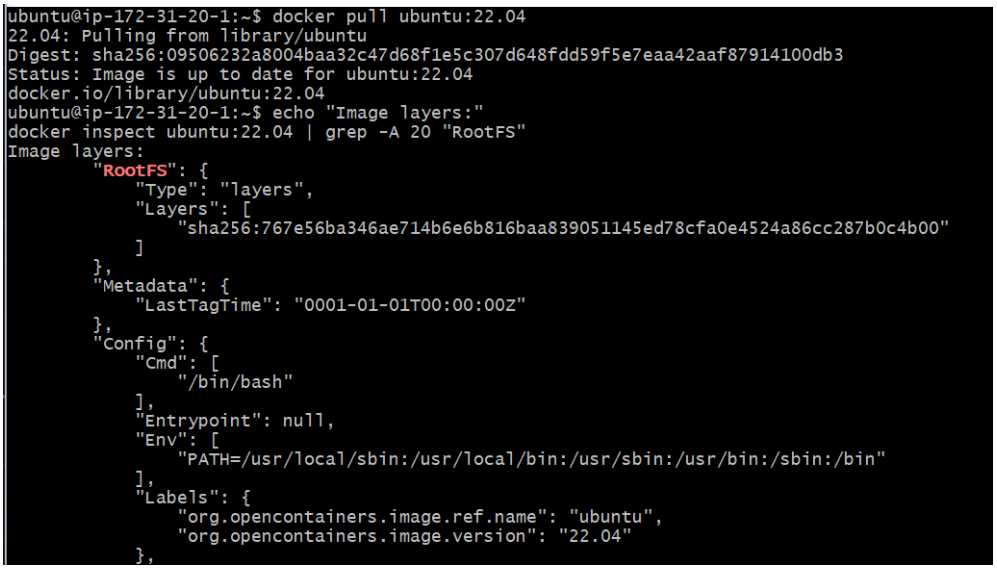
## Initial Overlay2 State

```
ubuntu@ip-172-31-20-1:~$ docker run -dit --name overlay2 ubuntu:22.04 sleep 600
b8cacb0f63c12a54440067f208cf2e6eb891bcb8facf941c841631e0422eae12
```



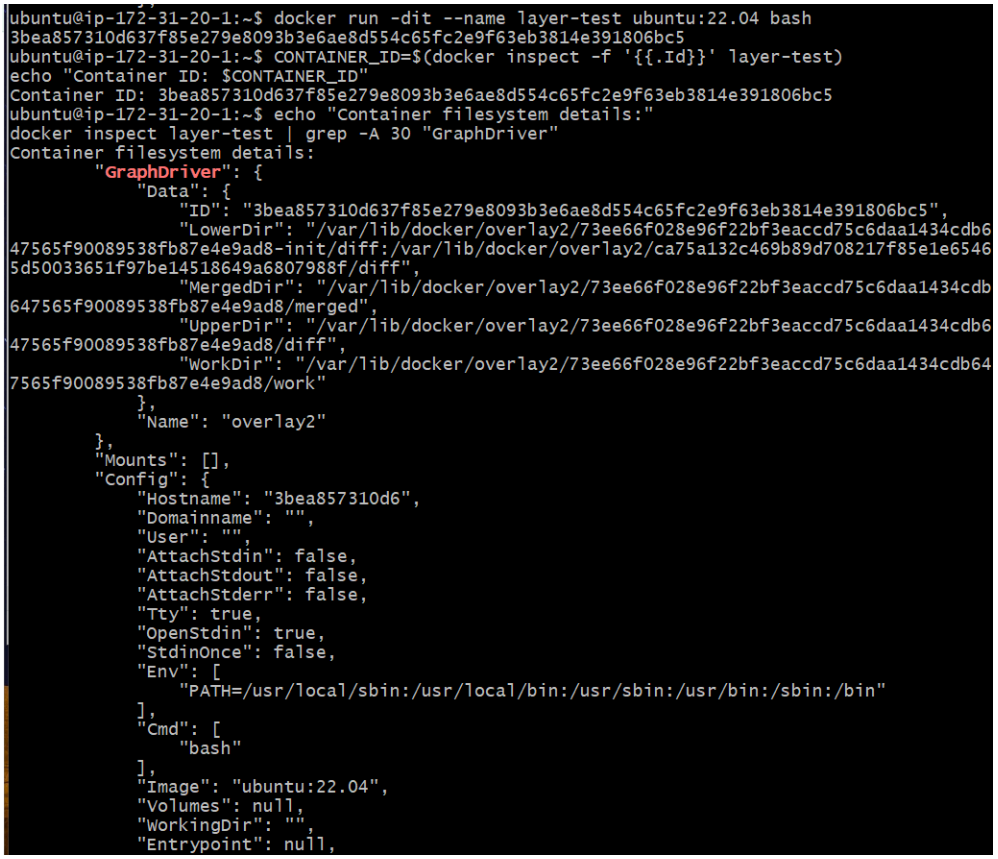

- Baseline directory listing of /var/lib/docker/overlay2/

## Image Pull and Layer Inspection

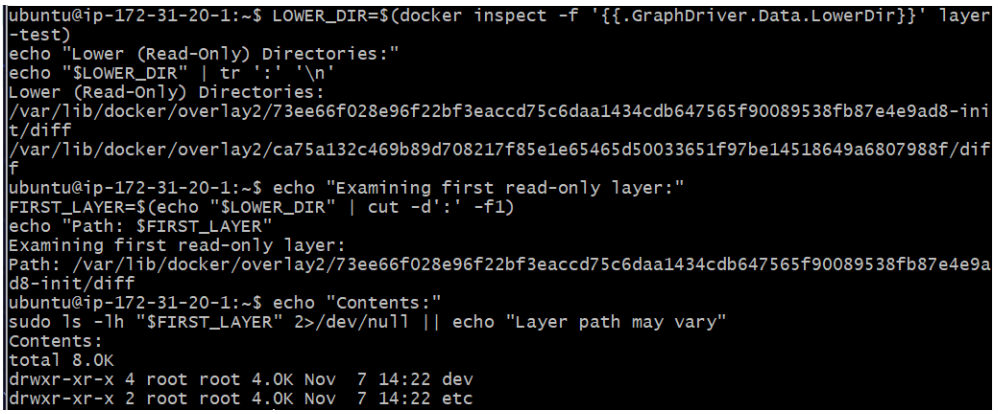



- Ubuntu image pulled with multiple layers visible

## Container Creation and GraphDriver Details




## Read-Only Image Layers




- 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

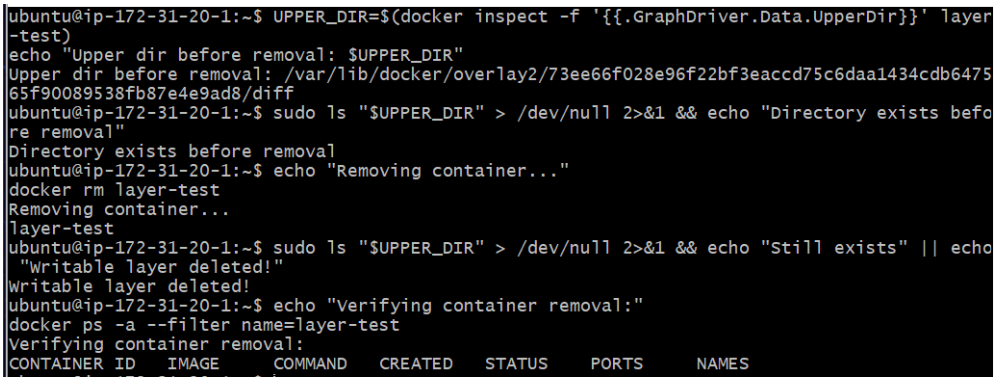
```
ubuntu@ip-172-31-20-1:~$ docker exec layer-test bash -c "echo 'MODIFIED BY CONTAINER' >> /etc
/bash.bashrc"
ubuntu@ip-172-31-20-1:~$ echo "Checking writable layer for copied file:"
UPPER_DIR=$(docker inspect -f '{{.GraphDriver.Data.UpperDir}}' layer-test)
sudo ls -lh "$UPPER_DIR/etc/bash.bashrc" 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

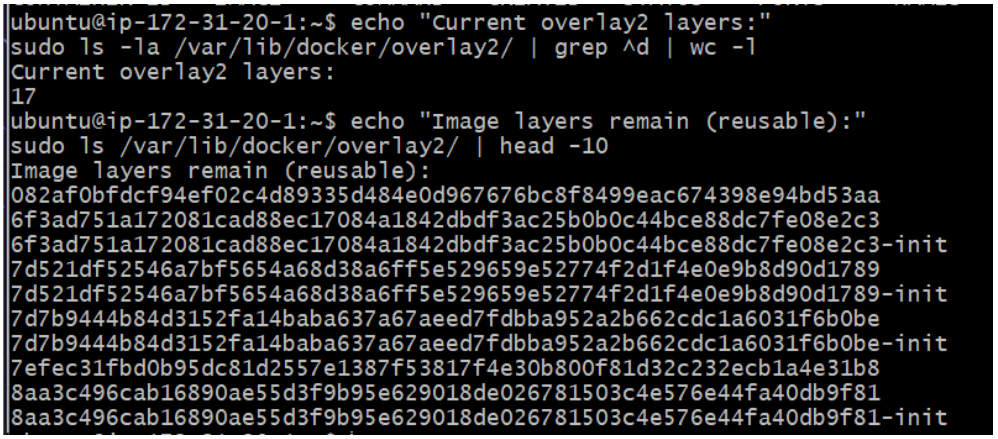
```
ubuntu@ip-172-31-20-1:~$ docker stop layer-test
layer-test
ubuntu@ip-172-31-20-1:~$ echo "Checking if writable layer still exists:"
UPPER_DIR=$(docker inspect -f '{{.GraphDriver.Data.UpperDir}}' layer-test)
echo "Upper dir path: $UPPER_DIR"
Checking if writable layer still exists:
Upper dir path: /var/lib/docker/overlay2/73ee66f028e96f22bf3eaccd75c6daa1434cdb647565f9008953
8fb87e4e9ad8/diff
ubuntu@ip-172-31-20-1:~$ 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 ancestry from dockerd to the container process and identify the roles of Docker components.

## Methodology:

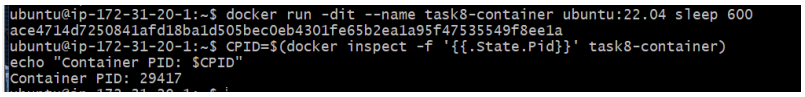
Identified dockerd and containerd processes

- Identified dockerd and containerd processes
- Created test container
- Used ps-ef to trace parent-child relationships
- Used pstree to visualize process tree
- Verified container sees itself as PID 1

## Docker Core Processes


- dockerd and containerd running as system services.

## Container Creation




- Test container created with known PID.

## Process Ancestry Chain

```
ubuntu@ip-172-31-20-1:~$ CPID=$(docker inspect -f '{{.State.Pid}}' task8-container)
echo "Container PID: $CPID"
Container PID: 29417
ubuntu@ip-172-31-20-1:~$ ps -f -p $CPID
UID          PID    PPID  C STIME TTY          TIME CMD
root       29417   29393  0 03:04 pts/0    00:00:00 sleep 600
ubuntu@ip-172-31-20-1:~$ echo "Parent processes:"
pstree -p -s $CPID
Parent processes:
systemd(1)───containerd-shim(29393)───sleep(29417)
ubuntu@ip-172-31-20-1:~$
```

- Shows parent-child relationships from systemd down to container process.

## Component Roles

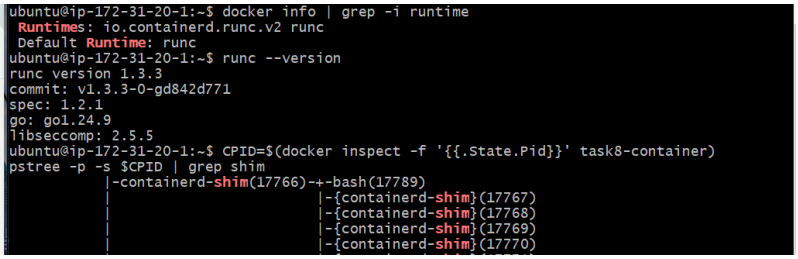
```
ubuntu@ip-172-31-20-1:~$ echo "dockerd:"
ps -f -p $(pgrep dockerd) | tail -1
dockerd:
root        6690       1  0 Nov06 ?        00:00:27 /usr/bin/dockerd -H fd:// --containerd=/
run/containerd/containerd.sock
ubuntu@ip-172-31-20-1:~$ echo "containerd:"
ps -f -p $(pgrep containerd) | tail -1
containerd:
root       29393       1  0 03:04 ?        Sl     0:00 /usr/bin/containerd-shim-runc-v2 -nam
espace moby -id ace4714d7250841afd18ba1d505bec0eb4301fe65b2ea1a95f47535549f8ee1a -address /r
un/containerd/containerd.sock
ubuntu@ip-172-31-20-1:~$ echo "containerd-shim:"
ps -ef | grep containerd-shim | grep task8-container | head -1
containerd-shim:
ubuntu@ip-172-31-20-1:~$ echo "Container process (PID $CPID):"
ps -f -p $CPID
Container process (PID 29417):
UID          PID    PPID  C STIME TTY          TIME CMD
root       29417   29393  0 03:04 pts/0    00:00:00 sleep 600
```

| 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 |

## Process Tree

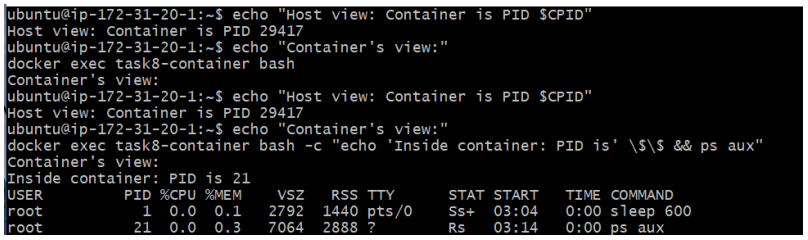
```
ubuntu@ip-172-31-20-1:~$ CPID=$(docker inspect -f '{{.State.Pid}}' task8-container)
ubuntu@ip-172-31-20-1:~$ pstree -p -a -s $CPID
systemd,1
  └─containerd-shim,29393 -namespace moby -idace4714d7250841afd18ba1d505bec0eb4301fe
      └─sleep,29417 600
ubuntu@ip-172-31-20-1:~$
```

## Understanding runc's Ephemeral Nature




runc is a short-lived process responsible for creating the container. It is invoked by containerd-shim, which uses runc to call the clone() system call with namespace flags. This creates the container's main process (PID 1) inside isolated namespaces. After starting the container, runc exits within milliseconds, leaving containerd-shim to manage the container. Hence, runc does not appear in the process tree for long.

## Container Internal View




- Inside container: Process sees itself as PID 1
- On host: Same process has different PID

## 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 clone() system call to create container in isolated namespaces, where it becomes PID 1
- The process tree shows clear chain: dockerd → containerd → containerd-shim → container.
- **Container process** runs isolated in its own namespaces as PID 1

## Task 9: **Comparative Namespace Experiment**

## Objective:

Compare container with default isolation versus container sharing host PID namespace, and explain how setns() joins existing namespaces.

## Methodology:

- Created container with default isolation
- Used strace to monitor setns() system calls
- Created container with --pid=host flag
- Compared PID namespace inodes
- Observed process visibility differences
- Explained setns() vs clone() system calls

## Isolated Container Creation

```
ubuntu@ip-172-31-20-1:~$ docker run -dit --name isolated-container ubuntu sleep
infinity
16059b90878a1c29fa5105076a8f3d966dc6a680e4dfc3ec379db7a7988b70cc
ubuntu@ip-172-31-20-1:~$ docker exec isolated-container ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.2  0.1   2696  1316 pts/0    Ss+  13:25   0:00 sleep infinit
y
root           8 70.0  0.4   7888  3896 ?        Rs   13:26   0:00 ps aux
ubuntu@ip-172-31-20-1:~$
```

- Cannot see host processes
- Complete PID isolation
- PID namespace inode: Different from host

```
[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]
```

- All namespaces show different inode numbers from host

## Capturing setns() with strace

```
ubuntu@ip-172-31-20-1:~$ sudo strace -f -e trace=setns,clone,clone3 -p $(pgrep dockerd) 2>&1 |
 tee /tmp/task9_raw.txt
strace: Process 6690 attached with 16 threads
[pid 13895] --- SIGURG {si_signo=SIGURG, si_code=SI_TKILL, si_pid=6690, si_uid=0} ---
[pid 13895] --- SIGURG {si_signo=SIGURG, si_code=SI_TKILL, si_pid=6690, si_uid=0} ---
[pid 13895] --- SIGURG {si_signo=SIGURG, si_code=SI_TKILL, si_pid=6690, si_uid=0} ---
[pid 13896] --- SIGURG {si_signo=SIGURG, si_code=SI_TKILL, si_pid=6690, si_uid=0} ---
[pid 13895] --- SIGURG {si_signo=SIGURG, si_code=SI_TKILL, si_pid=6690, si_uid=0} ---
[pid  7889] setns(29, CLONE_NEWNET)     = 0
[pid  7889] setns(30, CLONE_NEWNET)     = 0
[pid  7889] setns(29, CLONE_NEWNET)     = 0
[pid  7889] setns(30, CLONE_NEWNET)     = 0
[pid  7889] clone(child_stack=NULL, flags=CLONE_VM|CLONE_PIDFD|CLONE_VFORK|SIGCHLDstrace: Proc
ess 33866 attached
, parent_tid=[36]) = 33866
```

```
ubuntu@ip-172-31-20-1:~$ grep "setns" /tmp/task9_raw.txt | head -10
[pid  7889] setns(29, CLONE_NEWNET)     = 0
[pid  7889] setns(30, CLONE_NEWNET)     = 0
[pid  7889] setns(29, CLONE_NEWNET)     = 0
[pid  7889] setns(30, CLONE_NEWNET)     = 0
[pid  7889] setns(30, CLONE_NEWNET)     = 0
[pid  7889] setns(32, CLONE_NEWNET)     = 0
[pid  7889] setns(29, CLONE_NEWNET)     = 0
[pid  7889] setns(30, CLONE_NEWNET)     = 0
[pid  7889] setns(30, CLONE_NEWNET)     = 0
[pid  7889] setns(32, CLONE_NEWNET)     = 0
```

- setns() system calls when creating —pid=host container
- This proves Docker uses setns() to join host namespace instead of clone() to create new one.

## --pid=host Container

```
ubuntu@ip-172-31-20-1:~$ docker run -dit --name shared-pid-container --pid=host
ubuntu sleep infinity
1bc9c62b58c755a21721c29978ea0aff043275dc934a1c9fd41df6f2ea312e48
ubuntu@ip-172-31-20-1:~$ docker exec shared-pid-container ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  1.2  22660 11308 ?        Ss   Nov06   0:13 /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_workqu
ue_release]
root           4  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-r
u_gp]
root           5  0.0  0.0      0     0 ?        I<   Nov06   0:00 [kworker/R-s
nc_wq]
```

- PID namespace inode: **SAME as host** (identical number)
- Processes visible: ~100+ (all host processes visible)
- Can see and interact with host processes
- No PID isolation

## Analysis:

## Namespace Types and Their Functions:

| Namespace | Isolates | Default Container |
|---|---|---|
| **PID** | Process IDs | Isolated |
| **NET** | Network stack | Isolated |
| **MNT** | Filesystems | Isolated |
| **UTS** | Hostname | Isolated |
| **IPC** | Inter-process comm | Isolated |
| **USER** | User/group IDs | Isolated |
| **CGROUP** | Cgroup root | 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 |

## How setns() Joins Existing Namespaces

The setns() System Call: int setns(int fd, int nstype);

- fd : File descriptor to namespace (from /proc/[pid]/ns/pid)
- nstype: Namespace type (CLONE_NEWPID for PID namespace)

**Process:**

1. Docker opens host's namespace file: /proc/1/ns/pid
2. Gets file descriptor to host's PID namespace
3. Calls setns(fd, CLONE_NEPID)
4. Kernel moves process into host's namespace

5. Process now shares PID namespace with host

- setns() joins an existing namespace inode instead of creating a new one, allowing multiple processes to share it.

## Process Visibility Difference:

**Isolated container cannot see host processes because:**

1. clone(CLONE_NEWPID) created new PID namespace
2. New namespace has different inode number
3. Kernel isolates process visibility by namespace
4. Container only sees processes in its own namespace

- **-pid=host container sees all host processes because:**

1. setns(fd, CLONE_NEWPID) joined host's PID namespace
2. Container shares host's namespace inode
3. No isolation between container and host PIDs
4. Container sees everything host sees

## Conclusion:

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

Because clone() creates new isolated namespaces; setns() joins existing ones for selective sharing

This flexibility allows Docker to balance between:

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