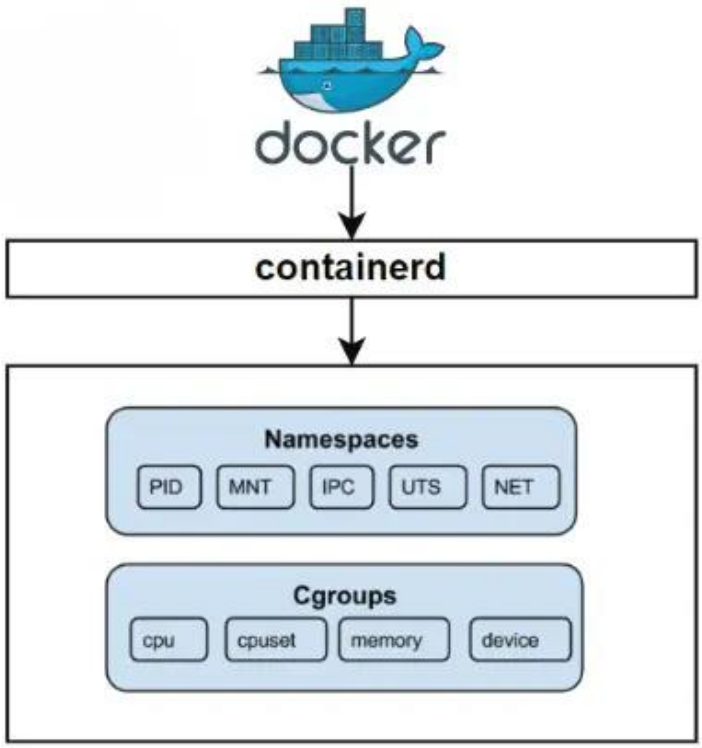
4.2 RESOURCE LIMITS AND PROCESS ISOLATION

Introduction:

- Docker uses Linux kernel features called **Namespaces** for **process isolation** and **Control Groups (cgroups)** for **resource limiting**.
- The below figure shows the **architecture of a Docker container** and how **isolation and resource management** are achieved in the Docker environment.



Container

- At the top is **Docker**, which is a platform used to **create, deploy, and manage containers**.
- **containerd** is a **core container runtime** used by **Docker** to manage the lifecycle of containers.
- **containerd** is a **lightweight container runtime** responsible for **running, managing, and supervising containers on a host system**.
- A **container** is a **lightweight isolated environment** where the application runs.
    - ✓ It shares the **host operating system kernel**
    - ✓ But it **isolates processes, network, and resources**
- The container contains two main Linux mechanisms:
    1. **Namespaces** (Isolation)
    2. **Control Groups (Cgroups)** (Resource control)

1. Namespaces:

- **Namespaces** are a fundamental feature of the Linux kernel that provide isolation for system resources such as process IDs (PID), user IDs, and network interfaces.
- By creating a separate namespace, processes can be isolated, simplified, and controlled.
- This allows developers to run multiple instances of applications safely on a single host system without clashing or interfering with one another.

Types of Namespaces:

1. **PID (Process ID) :** Isolates the process ID, so processes inside a container have an independent set of PIDs and cannot see the host processes or other containers.
2. **NET (Network) namespace** – Provides each container with its own network stack with own IP addresses, routing tables , and network interfaces.
3. **MNT (Mount) namespace** – Isolates filesystem mount points. Each container can have its own view of the filesystem.
4. **IPC (inter-process communication**) **namespace** – isolates inter-process communication resources like semaphores and message queues. Processes in one namespace cannot access IPC resources of another namespace.
5. **UTS (UNIX Time-Sharing) namespace** – allows containers to have their own hostname and domain name.
6. **USER namespace** – maps user IDs inside the container to different user IDs on the host, enhancing security. That isolates user and group IDs (UIDs and GIDs) inside a container from the host system.

By using namespaces, Docker ensures that containers are separated from each other and from the host.

How Namespaces Work:

- When a new namespace is created, the Linux kernel essentially sets up a new environment where processes can run.
- For instance, a new PID namespace allows a process to start with PID 1, making it feel like the root process of an independent system.
- This leads to encapsulated environments that run their own service instances while keeping resources isolated from other processes and containers.

Advantages:

1. Namespaces separate processes so containers cannot see or affect other processes.
2. They allow each container to have its own system resources like process IDs and file systems.
3. Namespaces improve security by isolating containers from the host system.
4. They provide separate network environments for each container.
5. Namespaces allow multiple applications to run safely on the same system.
6. They make **system management easier** by keeping containers independent.
7. Namespaces help in **efficient use of system resources**.

2. Control Groups(Cgroups):

- Control Groups **(Cgroups)** are a Linux kernel mechanism that allocates and restricts hardware resources such as CPU, memory, disk I/O, and network bandwidth among groups of processes.
- In container environments, each container runs inside a cgroup, which ensures that it cannot consume more resources than the limit assigned to it.

Purpose of Cgroups:

Cgroups are used for the following purposes:

1. Resource Limiting:

They limit the amount of system resources a process or container can use.

Example:

- Limit a container to **1 CPU core**
- Limit memory usage to **512 MB**

2. Resource Monitoring:

Cgroups track how much CPU, memory, or I/O a process is using.

3. Resource Isolation:

They ensure that one container cannot affect the performance of another container.

4. Resource Prioritization:

Some processes can be given higher priority for resources compared to others.

Types of Resources Controlled by Cgroups:

1. CPU: Controls how much CPU time a process can use.

Example:

- Container A → 70% CPU

- Container B → 30% CPU

This prevents one container consuming all CPU power.

2. Memory: Controls RAM usage of processes.

Example:

- Limit a container to 1 GB RAM
- If it exceeds the limit, the system may terminate the process (OOM kill).

3. CPUSET: Specifies which CPU cores a process can run on.

Example:

- Container runs only on Core 1 and Core 2.

4. Block I/O: Controls disk read and write operations.

Example:

- Limit a container to 50 MB/s disk speed.

5. Devices: Controls access to hardware devices.

Example:

- Allow container access to USB device.
- Restrict access to GPU or disk devices.

6. Network Bandwidth: Limits network traffic usage for containers.

Example:

- Limit a container to 100 Mbps bandwidth.

How Cgroups Work:

Cgroups work by grouping processes together and applying resource limits to the group instead of individual processes.

Steps:

1. The system creates a control group.
2. Processes are assigned to that group.
3. Resource limits are applied to the group.
4. The Linux kernel enforces the limits automatically.

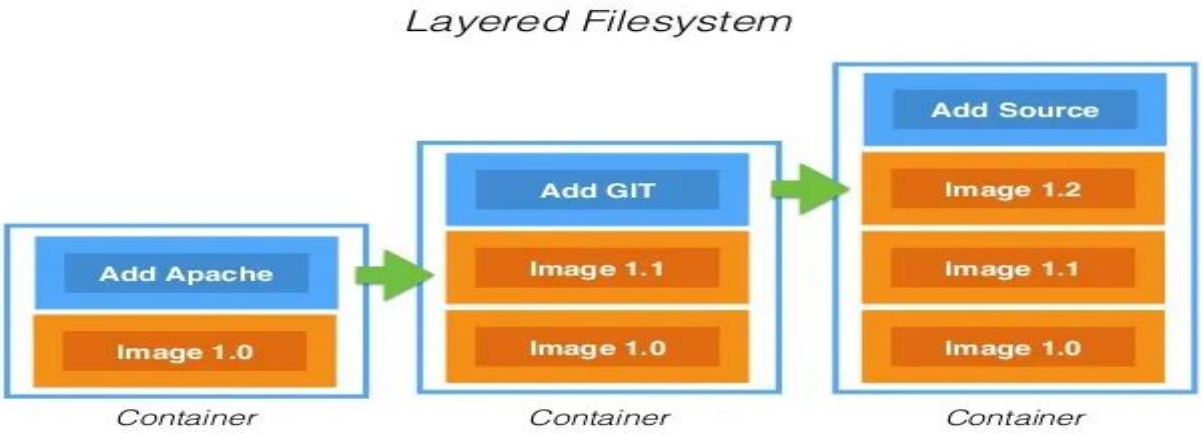
Advantages:

- ✓ Efficient resource management.
- ✓ Prevents system overload**.**
- ✓ Ensures fair resource sharing**.**
- ✓ Improves system stability**.**
- ✓ Enables container resource control**.**

Layered File System:

A **layered file system** (also called a **union file system**) in Docker is a way of organizing the files of a Docker image into multiple **stacked layers**, where each layer represents **changes** like adding, modifying, or deleting files.

- Each layer is immutable (cannot be changed once created).
- Layers are stacked on top of each other to form the final Docker image.
- Docker uses copy-on-write (COW) so that changes made in a container do not affect the underlying image layers.
- This system saves storage, speeds up builds, and allows multiple images to share common layers efficiently.



Docker's layered file system is a key part of modern software development and deployment.

- It organizes files in layers, making software development faster and more efficient.
- Each layer contains changes to the filesystem, so images can be built quickly.
- The underlying filesystem stays unchanged, ensuring consistency.
- Understanding this layered system is important for developers and DevOps, as containerization is widely used today.
- Docker's layers help in better storage management, faster image deployment, and easier updates.

Primary Terminologies:

1. Docker Image:

- A **Docker image** is a lightweight, self-contained package that has everything needed to run a software application such as code, runtime, libraries, and dependencies.

- Images are templates used to create Docker containers, which run as isolated processes on the host system.
- Docker images use a layered file system for efficiency and sharing common elements across images.
- Images are immutable, meaning they cannot be changed after creation, ensuring consistency across environments.

2. Layered File System (Union File System):

- It is Central to Docker's architecture.
- Stacks layers on top of each other; each layer represents changes like file additions, modifications, or deletions.
- Only changes from the previous layer are stored, saving space and avoiding duplicate data.
- Uses copy-on-write (COW) so container changes don't affect the base image.

3. Dockerfile:

- A **text file** with step-by-step instructions to build a Docker image.
- **Commands** include selecting a base image, installing packages, copying files, and configuring the environment.
- Each command creates a **new layer**, and the final image contains the application plus all dependencies.

Understanding Docker Layered File System:

- Each layer is a modification of the filesystem (add, change, or remove files).
- When a **container starts**, Docker merges all layers into a read/write filesystem, giving the container an isolated environment.

Step-by-Step Process:

1. **Image Creation**: Docker images are built from a Dockerfile. Each command creates a new layer.
2. **Layered Architecture**: Images are stacked layers — base layer = core filesystem; next layers = modifications.
3. **Efficient Storage**: Docker uses copy-on-write, storing only modified files as new layers. This saves disk space and speeds up deployment.

Example:

- Create a Docker image from a Dockerfile.
- Make changes (add files, modify configs).
- Docker creates layers for each change.

- Only changed files are stored in new layers, while the rest is shared efficiently.

1. Create a Dockerfile:

FROM ubuntu:22.04

RUN apt-get update && apt-get install -y curl

That Dockerfile is to make Docker create an image from all the updated Ubuntu image and then install Nginx based on it.

2. Build the Docker Image:

First open your console and then switch to the directory where the Dockerfile is located.

Then run the following command to build the Docker image:

docker build -t myimage:1.0 .

The CMD is the 1st instruction in the docker from build command, use the instructions of the dockerfile to build a docker image named **myimage:1.0.**

3. View Image Layers:

After building the image, we can view its layers using the docker history command:

docker images

Hence, this command shows the history of the image layers, every hanging the statements executed at every layer when an image was being built.

4. Inspect Image Layers:

One way to check the components for the image is by using the command docker inspect which can help to disclose the image layers.

For example:

docker inspect myimage:1.0

These ones give a detailed description of the Docker image that may include layers, settings, Metadata, and so on.

- **Create Dockerfile** - defines layers.
- **Build image** - each instruction becomes a layer.
- **View images** -docker images.
- **Check layers** - docker history.
- **Inspect details** - docker inspect.