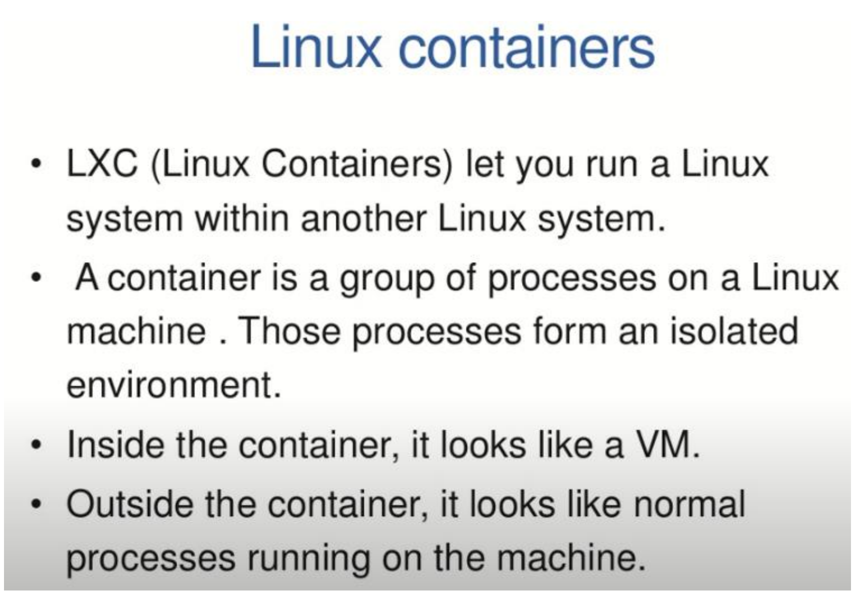
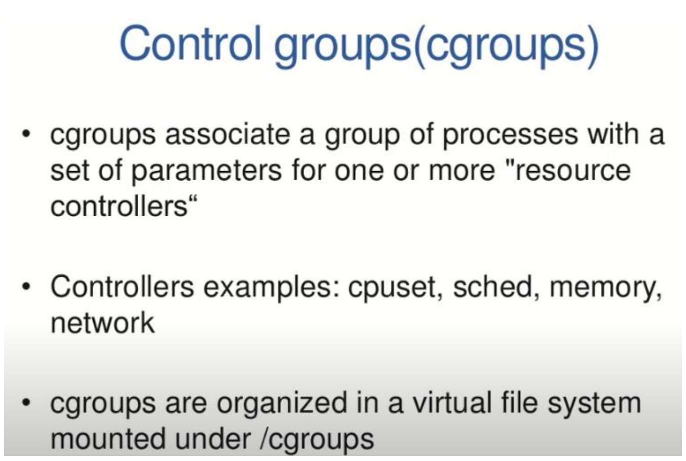


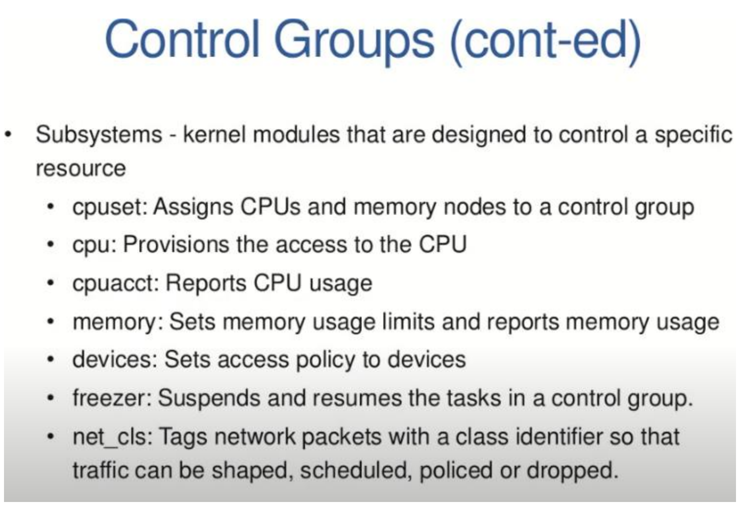
Linux containers

- LXC (Linux Containers) let you run a Linux system within another Linux system.
- A container is a group of processes on a Linux machine . Those processes form an isolated environment.
- Inside the container, it looks like a VM.
- Outside the container, it looks like normal processes running on the machine.



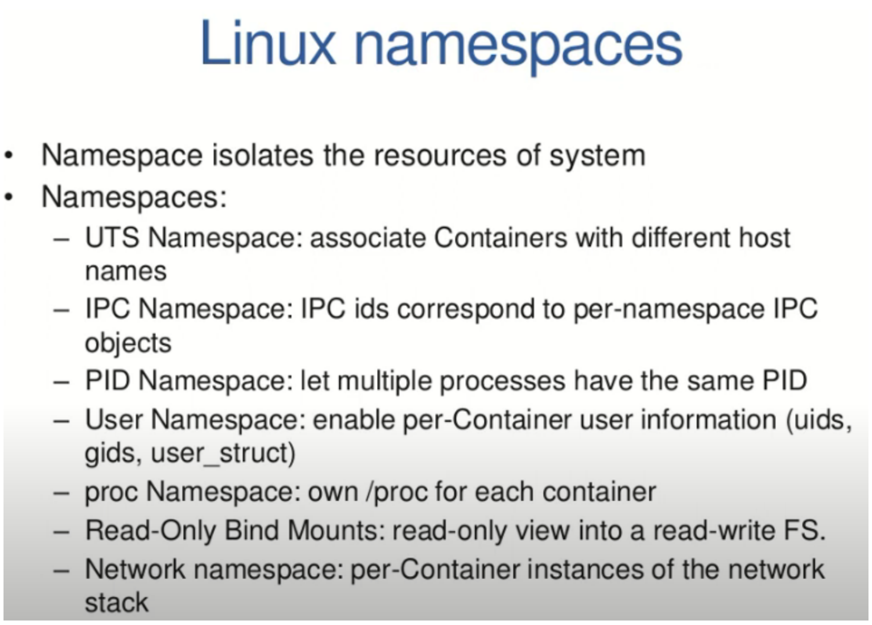
Control groups(cgroups)

- cgroups associate a group of processes with a set of parameters for one or more "resource controllers“
- Controllers examples: cpuset, sched, memory, network
- cgroups are organized in a virtual file system mounted under /cgroups



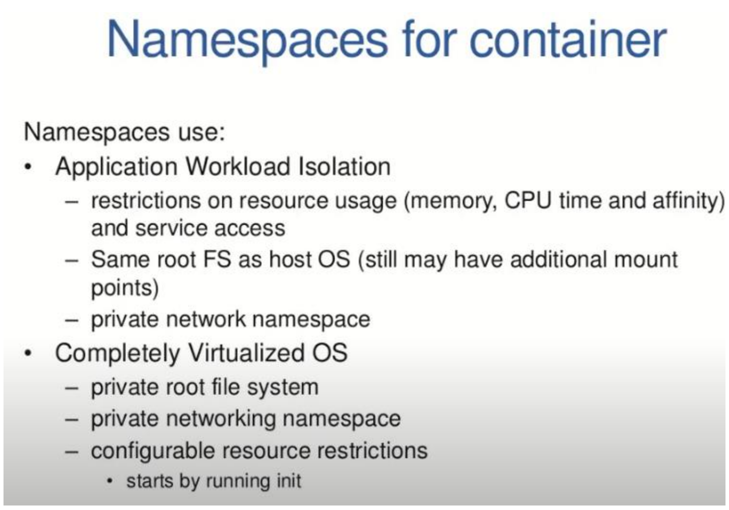
Control Groups (cont-ed)

- Subsystems - kernel modules that are designed to control a specific resource
  - cpuset: Assigns CPUs and memory nodes to a control group
  - cpu: Provisions the access to the CPU
  - cpuacct: Reports CPU usage
  - memory: Sets memory usage limits and reports memory usage
  - devices: Sets access policy to devices
  - freezer: Suspends and resumes the tasks in a control group.
  - net_cls: Tags network packets with a class identifier so that traffic can be shaped, scheduled, policed or dropped.



Linux namespaces

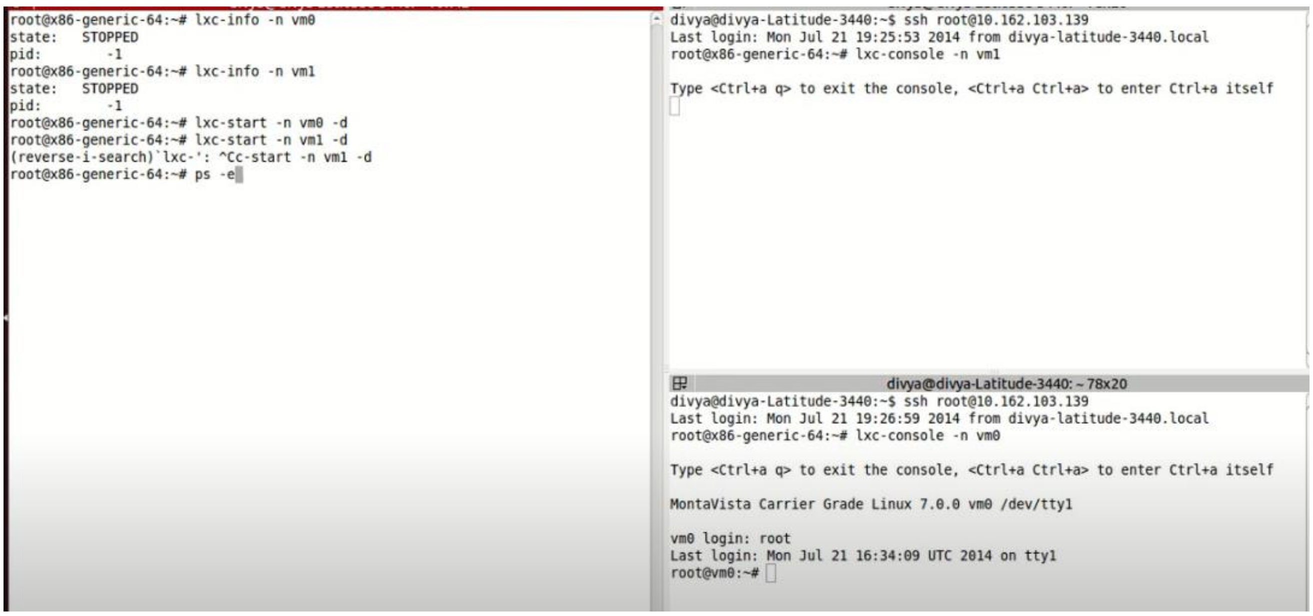
- Namespace isolates the resources of system
- Namespaces:
  - UTS Namespace: associate Containers with different host names
  - IPC Namespace: IPC ids correspond to per-namespace IPC objects
  - PID Namespace: let multiple processes have the same PID
  - User Namespace: enable per-Container user information (uids, gids, user_struct)
  - proc Namespace: own /proc for each container
  - Read-Only Bind Mounts: read-only view into a read-write FS.
  - Network namespace: per-Container instances of the network stack



Namespaces for container

Namespaces use:

- Application Workload Isolation
  - restrictions on resource usage (memory, CPU time and affinity) and service access
  - Same root FS as host OS (still may have additional mount points)
  - private network namespace
- Completely Virtualized OS
  - private root file system
  - private networking namespace
  - configurable resource restrictions
    - starts by running init



```
root@x86-generic-64:~# lxc-info -n vm0
state:   STOPPED
pid:        -1
root@x86-generic-64:~# lxc-info -n vm1
state:   STOPPED
pid:        -1
root@x86-generic-64:~# lxc-start -n vm0 -d
root@x86-generic-64:~# lxc-start -n vm1 -d
(reverse-i-search)`lxc-': ^Cc-start -n vm1 -d
root@x86-generic-64:~# ps -e
```

```
divya@divya-Latitude-3440:~$ ssh root@10.162.103.139
Last login: Mon Jul 21 19:25:53 2014 from divya-latitude-3440.local
root@x86-generic-64:~# lxc-console -n vm1

Type <Ctrl+a q> to exit the console, <Ctrl+a Ctrl+a> to enter Ctrl+a itself
```

divya@divya-Latitude-3440: ~ 78x20

```
divya@divya-Latitude-3440:~$ ssh root@10.162.103.139
Last login: Mon Jul 21 19:26:59 2014 from divya-latitude-3440.local
root@x86-generic-64:~# lxc-console -n vm0

Type <Ctrl+a q> to exit the console, <Ctrl+a Ctrl+a> to enter Ctrl+a itself

MontaVista Carrier Grade Linux 7.0.0 vm0 /dev/tty1

vm0 login: root
Last login: Mon Jul 21 16:34:09 UTC 2014 on tty1
root@vm0:~#
```

- Containers do not run ON docker. Containers are processes - they run on the Linux kernel. Containers are Linux.
- The docker daemon is one of the many user space tools/libraries that talks to the kernel to set up containers

Key concepts include:

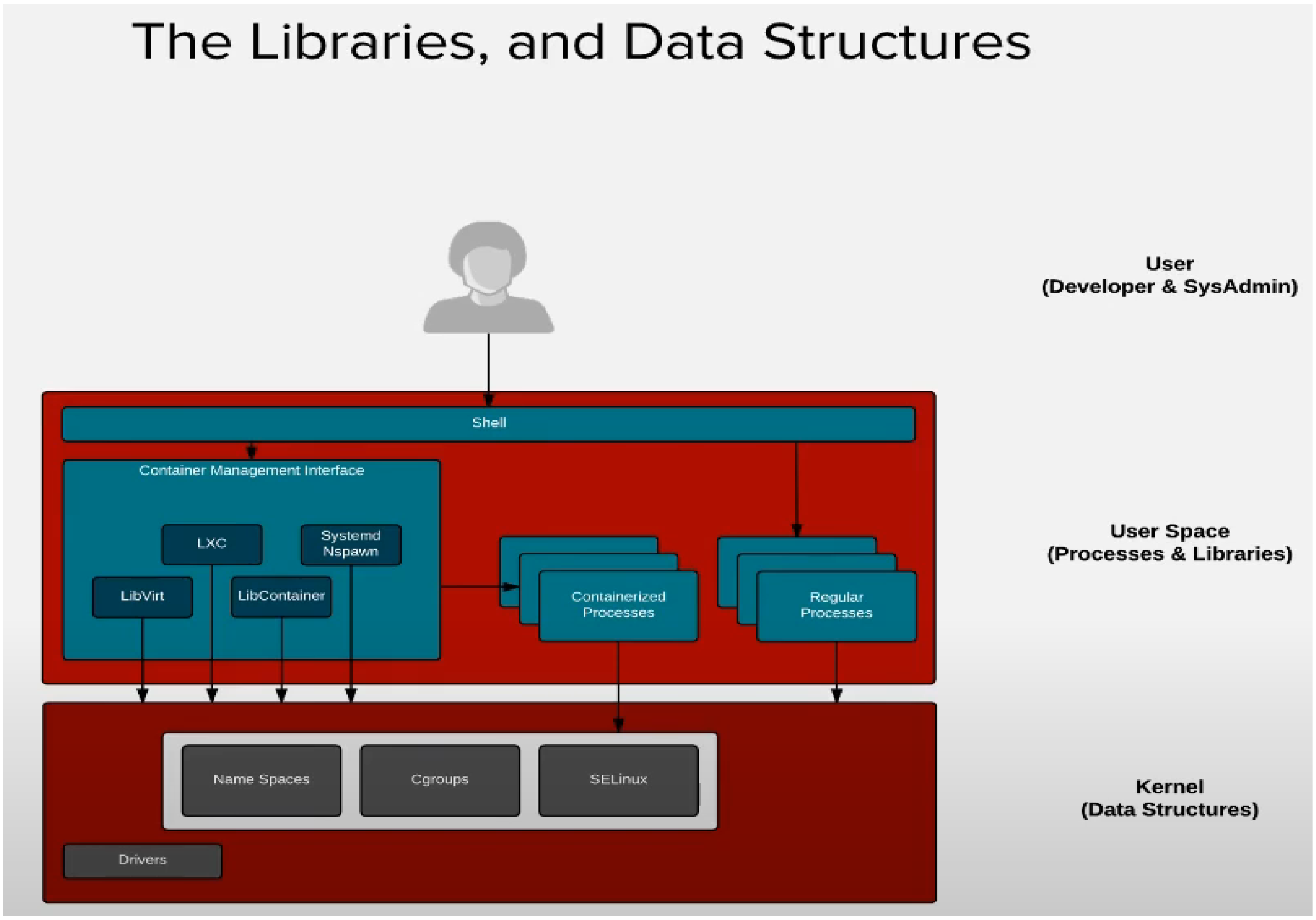
1.**Containerization Basics**: Containers are simply Linux processes utilizing kernel features like namespaces, cgroups, and SELinux for isolation and resource management. They function like regular processes but are created through specialized libraries.

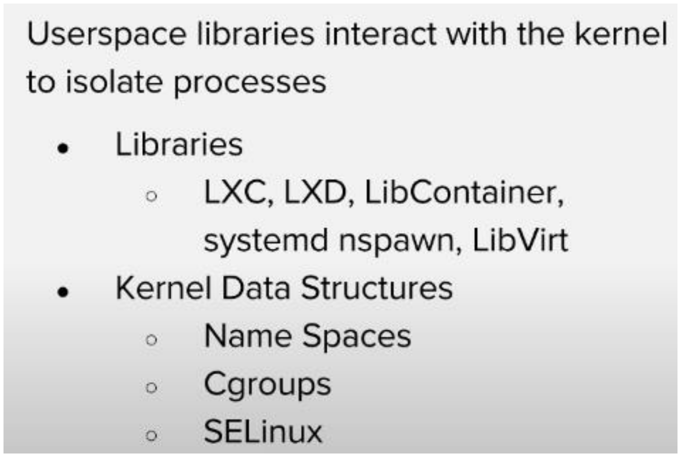
2.**Docker's Contributions**: Docker introduced a user-friendly daemon and a simple API for managing containers, along with the concept of container images standardized by the Open Container Initiative (OCI).

3.**Orchestration**: Tools like OpenShift and Kubernetes operate at a higher level, coordinating multiple nodes and managing workloads while still interacting with the Docker ecosystem.

4.**User Interaction**: Users can interface with the system through a load balancer or directly with the Docker daemon for troubleshooting.

5.**Community Landscape**: Understanding the relationships between various projects like Kubernetes, OpenShift, and OCI is crucial. These collaborations help create a standardized environment for container management.



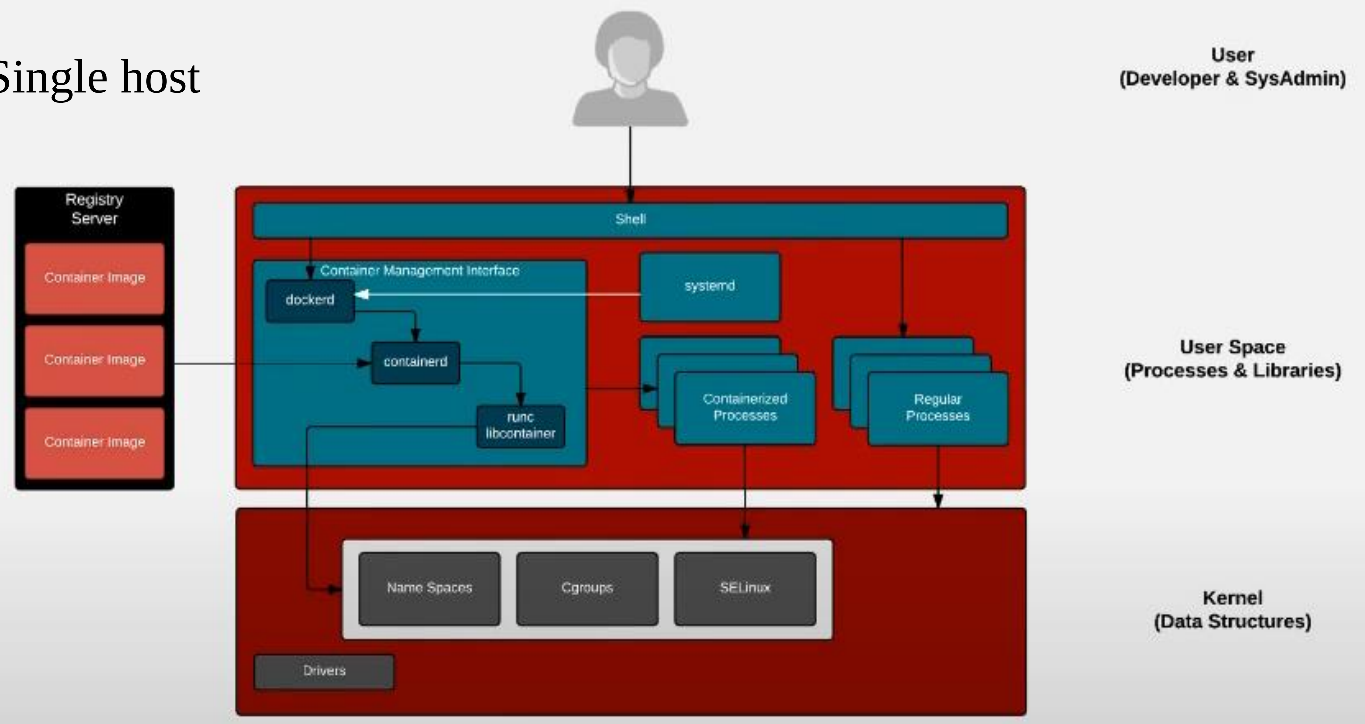


Userspace libraries interact with the kernel to isolate processes

- Libraries
  - LXC, LXD, LibContainer, systemd nspawn, LibVirt
- Kernel Data Structures
  - Name Spaces
  - Cgroups
  - SELinux

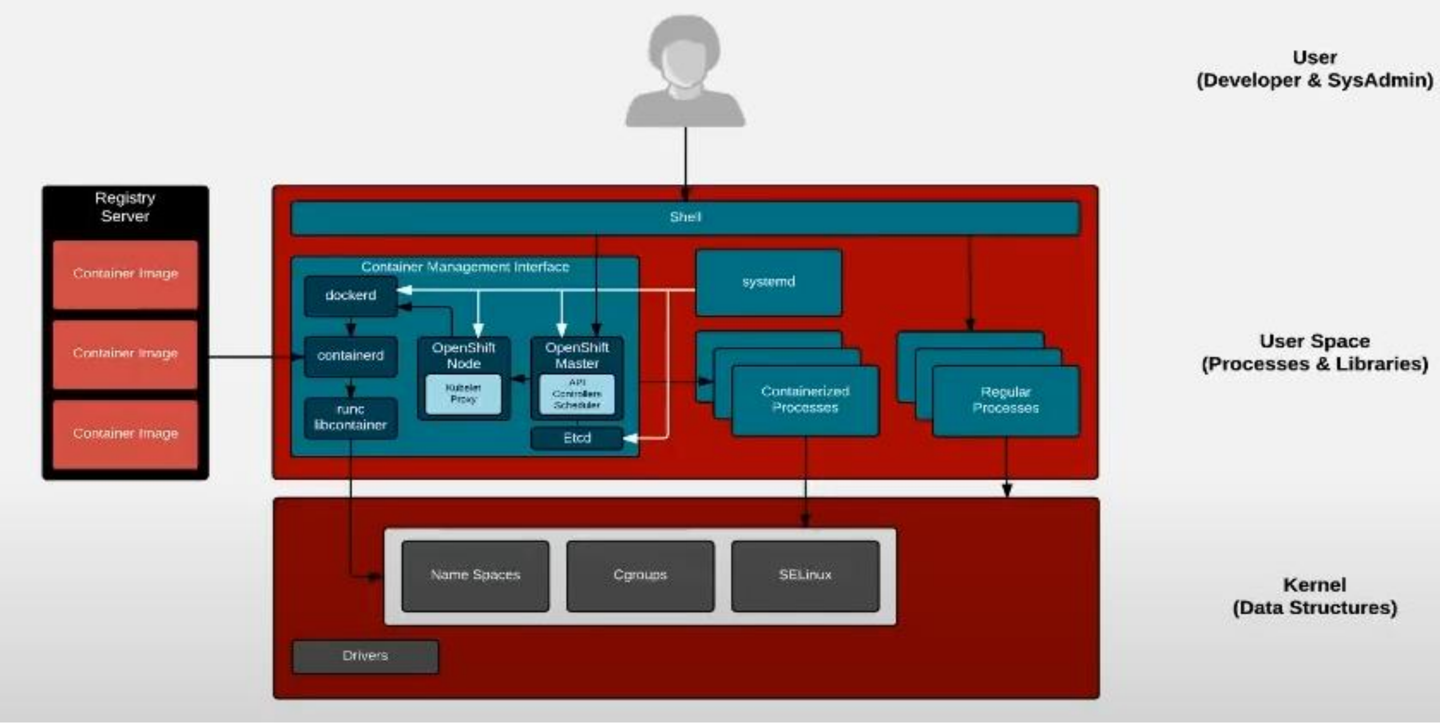
From traditional uses normal Linux we can create LXC which is separate jail kind
One lxc cant talk to other lxc
Here kernel has namespace(resources processess) and cgroup

Single host



Use can login into container using dockerd which it speaks with kernal as api

•**Docker Daemon**: The daemon simplifies the process of launching containers and supports the use of container images.
•**OCI Image Format**: Docker's introduction of container images led to the development of the Open Container Initiative (OCI) image format, providing a standardized, industry-neutral format for container images. This enables interoperability among various software and registries.
•**Architecture**: The Docker daemon initiates container processes through libraries that interact with the kernel, using features like namespaces and SELinux for isolation. Within each container, there is a virtual view of the Linux mounts, with the container image serving as the root filesystem.



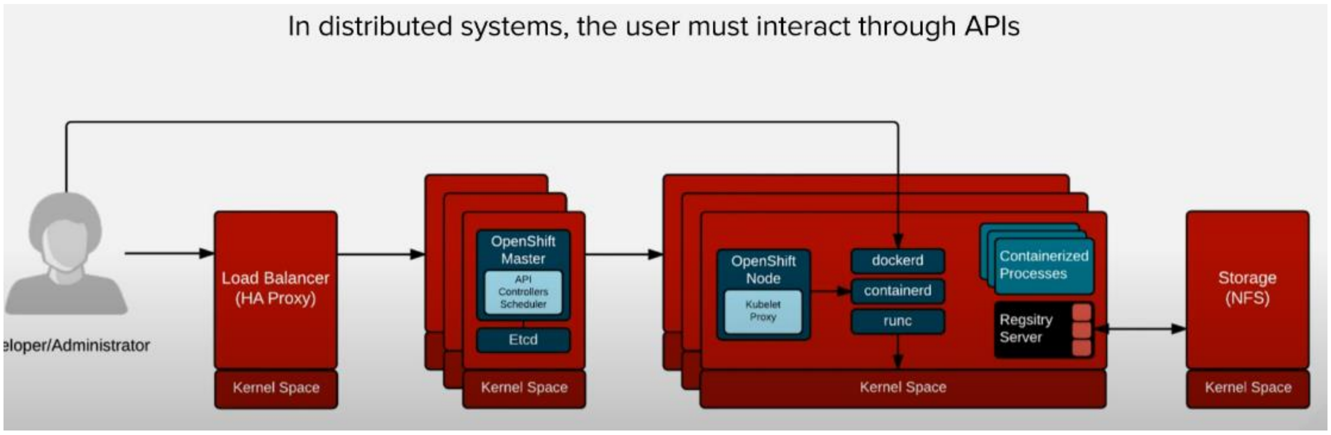
Key points include:

1.**Architecture Overview**: The orchestration involves several components:

- •**Docker Daemon**: Manages container processes.
- •**Containerd and RunC**: Handle the actual container execution by interacting with the Linux kernel.
- •**OpenShift Master and Nodes**: Users interact with the OpenShift Master using commands (oc or kubectl), which communicates with the OpenShift nodes.

2.**User Space Processes**: All components, including the Docker daemon, Containerd, RunC, and OpenShift processes, operate in user space. They do not reside in the kernel, emphasizing that these are standard daemons and processes without any magical functionalities.

3.**System Initialization**: The architecture illustrates that systemd starts the Docker daemon, OpenShift nodes, and the OpenShift master, which together manage the orchestration of containerized applications across the cluster.
Overall, this section outlines the multi-layered orchestration architecture that enables efficient container management and deployment using OpenShift and Kubernetes.



In the "Class Architecture" section, the focus is on the distributed architecture of OpenShift, which includes multiple masters and nodes. Key components include:

1.**Load Balancing**: A load balancer or proxy is used to distribute traffic to the OpenShift master's API, enhancing system reliability and performance.

2.**OpenShift Master**: This component maintains an etcd database that tracks the status of all nodes, ensuring it knows which nodes are operational and can reschedule workloads as necessary.

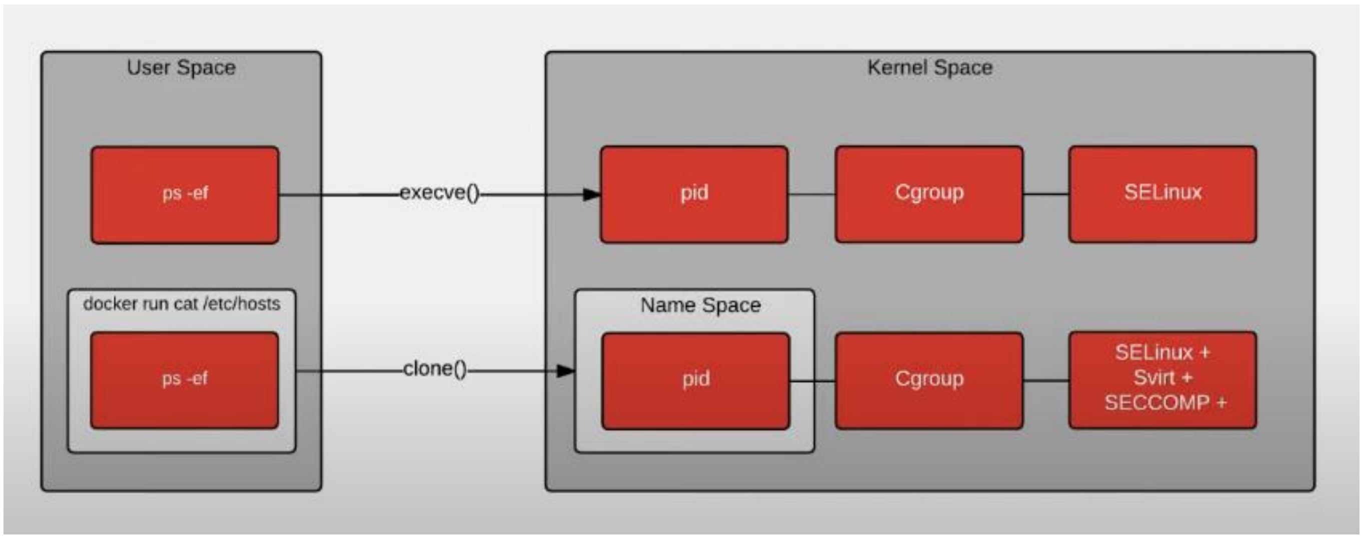
3.**Node Configuration**: Each node runs the Docker daemon, Containerd, and RunC to manage containerized processes. The architecture allows for the registry server to exist on some nodes, providing flexibility.

4.**Persistent Storage**: The registry requires persistent storage, typically using NFS, but can also support various other storage solutions.

5.**User Interaction**: Users can interact with the cluster via the load balancer or directly through the Docker daemon for troubleshooting. While direct connections are possible, they are generally not recommended for regular use.

Virtualization vs. Containerization
This section begins by comparing virtual machines (VMs) and containers. VMs bundle an entire operating system, application, and dependencies into a single disk image, making updates cumbersome due to tightly coupled components. In contrast, containers package only application dependencies with the host OS kernel remaining separate, allowing for greater agility in deployment and updates. Containers are lightweight and can be deployed across clusters more efficiently.

Operating System Interaction
The operating system has four main interactions involving user programs, libraries, system calls, and the kernel. Applications rely on system calls to interact with protected resources, and compatibility between libraries and system calls is crucial.



When comparing regular processes to containerized processes, the fundamental operation remains the same: both types of processes make system calls to access protected resources, such as RAM, disk, and file systems. System calls serve as a common language for processes to communicate with the kernel.

Key Differences

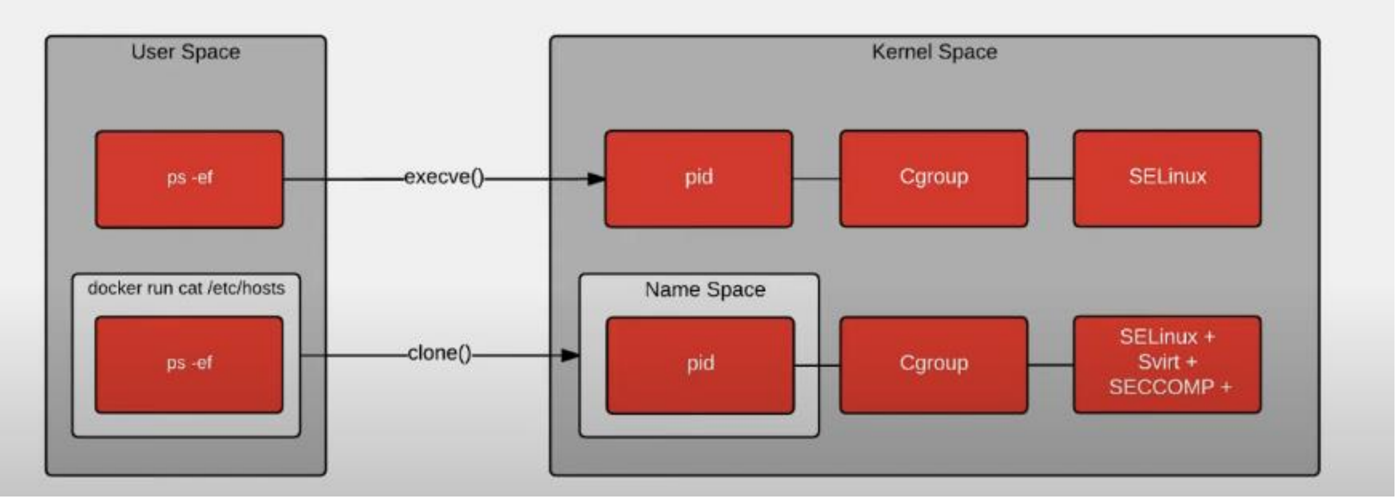
1.Environment Visibility:

- •**Regular Processes**: When you run a command like ps on a regular Linux system, you see all processes running on that system.

- •**Containerized Processes**: Inside a container, using the same ps command will only show the processes running within that container due to the use of process ID namespaces. This creates a more isolated environment, where processes are "jailed" and do not have visibility into the host system or other containers.

2.Isolation:

- •Containerized processes benefit from additional layers of isolation, which include namespaces for process IDs, network interfaces, and file systems. This helps in ensuring that the processes are secure and do not interfere with each other.



How Containerization Works

Containerization in Linux involves several key concepts and system calls that enable the isolation and management of processes. Here's how it works:

Process Creation

1.Starting a Process:

- Normal Linux processes are typically initiated using system calls like fork and exec. For instance, when you type a command in a shell (like bash), it uses exec to run that command.

2.Visibility:

- When you execute a command such as ps, the process can access various kernel data structures, allowing it to see all processes running on the system, unless restricted by controls like cgroups or SELinux.

Isolation Mechanisms

1.Namespaces:

- The introduction of the clone system call allows processes to operate within their own namespaces. This isolation means that processes inside a container can only see and interact with other processes within the same container, creating a separate execution environment.

2.Control Groups (cgroups):

- Cgroups are used to limit, prioritize, and isolate resource usage (like CPU, memory, etc.) for a set of processes, further enforcing boundaries.

3.SELinux:

- SELinux (Security-Enhanced Linux) can apply security policies to restrict how processes interact. Each process within a container is assigned a unique Multi-Category Security (MCS) label, allowing only those processes to access each other's resources.

4.Seccomp:

- Seccomp (Secure Computing Mode) allows for filtering system calls that a process can make. This means that only specific system calls necessary for the application's operation are permitted, enhancing security.

Practical Application

In a containerized environment, when a process runs a command like cat /etc/hosts or ps, it will only see the processes and resources within that container due to the enforced namespaces. All processes in the container share the same PID namespace, cgroup, and MCS label, ensuring they can only interact with each other.

Understanding Namespaces in Containerization

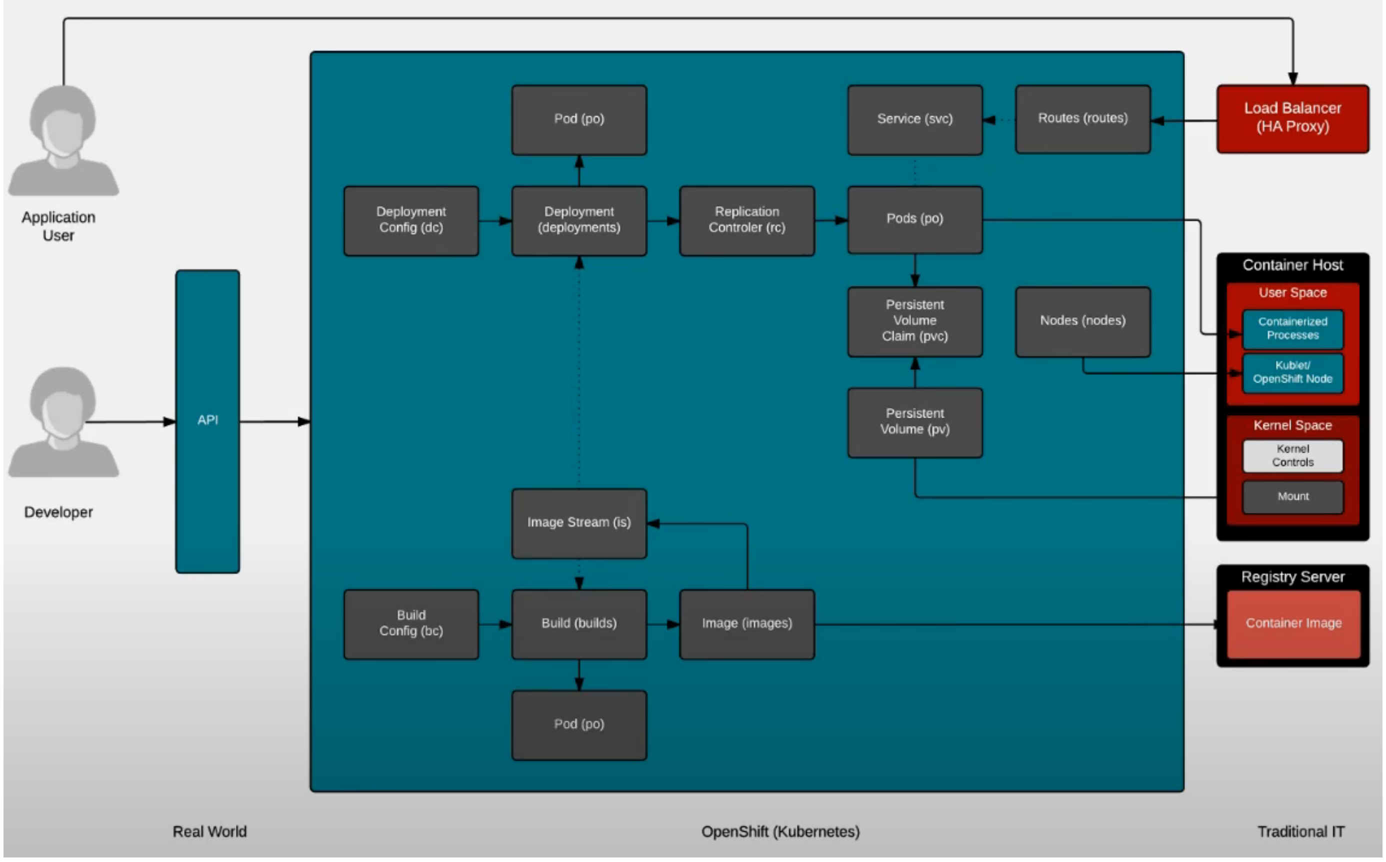
Namespaces are critical for isolating processes in containerized environments, allowing each container to have its own view of the system. Here's a deeper look into the various types of namespaces and their functions:

Types of Namespaces

1. **Process ID (PID) Namespace**:
    - Isolates process IDs, so processes within a container see only their own PIDs, not those of other containers or the host.
2. **User ID (UID) Namespace**:
    - Allows for different user IDs within containers, enhancing security by enabling non-root user execution within the container.
3. **Network Namespace**:
    - Provides isolated networking, giving each container its own network interfaces, IP addresses, and routing tables, preventing interference with other containers or the host.
4. **Mount Namespace**:
    - Each container can have its own filesystem view, only accessing files and directories that are explicitly mounted for it. This is essential for security and resource management.
5. **IPC Namespace**:
    - Isolates inter-process communication resources, so containers can communicate with their own processes without affecting others.
6. **UTS Namespace**:
    - Allows containers to have their own hostname and domain name, enabling isolation at the networking level.
7. **Time Namespace**:
    - Provides a separate notion of time, allowing containers to have their own time settings.

Creating and Using Namespaces

- **Clone System Call**:
    - When starting a container, the clone syscall is used to create these namespaces. This syscall allows for specifying which namespaces to use, effectively defining the isolated environment for the process.
- **Virtual View**:
    - Each namespace provides a "virtual view" of the system, enabling containers to operate independently and securely. Processes within a namespace cannot see or interact with resources in other namespaces unless explicitly shared.



The application definition in Kubernetes and OpenShift describes how applications should be built, deployed, and interact with various resources. It utilizes various objects that represent real-world components like container hosts, registries, and load balancers, making it easier to understand. Kubernetes can be complex initially, but by mapping familiar concepts to its objects, the learning curve can be mitigated. At the core of application definition are the **BuildConfig** and **DeploymentConfig** objects.

1.BuildConfig: This object defines how builds are initiated. When a build is triggered, it generates an image, which is then stored in a registry. The image stream represents these images and can trigger further automation, like cascading builds for dependencies.

2.DeploymentConfig: Once the images are ready, the DeploymentConfig handles the deployment of these images into pods, which run the applications. It allows for complex deployment patterns, enabling sophisticated upgrade processes, such as coordinating the update of a database and web server.

Additionally, the application definition can include persistent storage specifications and service configurations for internal and external access. This standardized object model allows for easy portability between environments, enabling consistent deployment from development to production.

Overall, this structured approach provides an elegant way to manage complex applications in a Kubernetes environment.