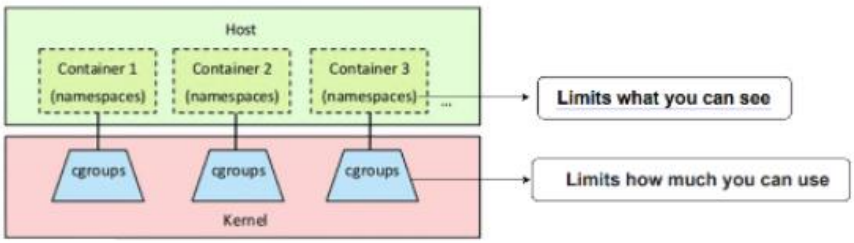
Topic-2 Basic

- Namespace
- CGroup
- Chroot

Namespace



- Namespaces are a **feature of the Linux kernel** that **partitions kernel resources** such that "**one set of processes sees one set of resources**" and "**another set of processes sees a different set of resources**".
- The feature works by having the same namespace for a group of resources and processes, but those namespaces refer to distinct resources.
- The key feature of namespaces is that they **isolate processes from each other**. On a server where you are running many different services, isolating each service and its associated processes from other services means that there is a smaller blast radius for changes, as well as a smaller footprint for security-related concerns.
- Namespaces provide processes with their own view of the system.
- There are six common types of namespaces in wide use today.

    a. Process isolation (PID namespace) : isolation of system process tree
    b. Network interfaces (net namespace) : isolation of host network stack
    c. Unix Timesharing System (uts namespace) : isolation of hostname
    d. User namespace : isolation of user IDs
    e. Mount (mnt namespace) : isolation of host filesystem mount points
    f. Interprocess communication (IPC) : isolation of ipc(shared segments, semaphores)
    g. cgroup namespace: isolation of the virtual cgroups filesystem of the host

Scenario,

Using containers during the development process gives the developer an isolated environment that looks and feels like a complete VM. It's not a VM, though – it's a process running on a server somewhere. If the developer starts two containers, there are two processes(due to process namespace) running on a single server somewhere – but they are isolated from each other.

Check available namespaces

$lsns

```
subarno@master:~$ lsns
        NS TYPE   NPROCS   PID USER    COMMAND
4026531834 time         3  1943 subarno /lib/systemd/systemd --user
4026531835 cgroup       3  1943 subarno /lib/systemd/systemd --user
4026531836 pid          3  1943 subarno /lib/systemd/systemd --user
4026531837 user         3  1943 subarno /lib/systemd/systemd --user
4026531838 uts          3  1943 subarno /lib/systemd/systemd --user
4026531839 ipc          3  1943 subarno /lib/systemd/systemd --user
4026531840 mnt          3  1943 subarno /lib/systemd/systemd --user
4026531992 net          3  1943 subarno /lib/systemd/systemd --user
```

$ lsns --help

$ lsns -t pid -o ns,pid,command

```
        NS   PID COMMAND
4026531836  1943 /lib/systemd/systemd --user
```

A. Process isolation (PID namespace):

Every time a computer with Linux boots up, it starts with just one process, with process identifier (PID) 1. This process is the **root of the process tree**, and it initiates the rest of the system by performing the appropriate maintenance work and starting the correct daemons/services. All the other processes start below this process in the tree. The PID namespace allows one to spin off a new tree, with its own PID 1 process. The process that does this remains in the parent namespace, in the original tree, but makes the child the root of its own process tree.

- With **PID namespace** isolation, processes in the child namespace have no way of knowing of the parent process's existence.
- However, processes in the **parent namespace** have a complete view of processes in the child namespace, as if they were any other process in the parent namespace.

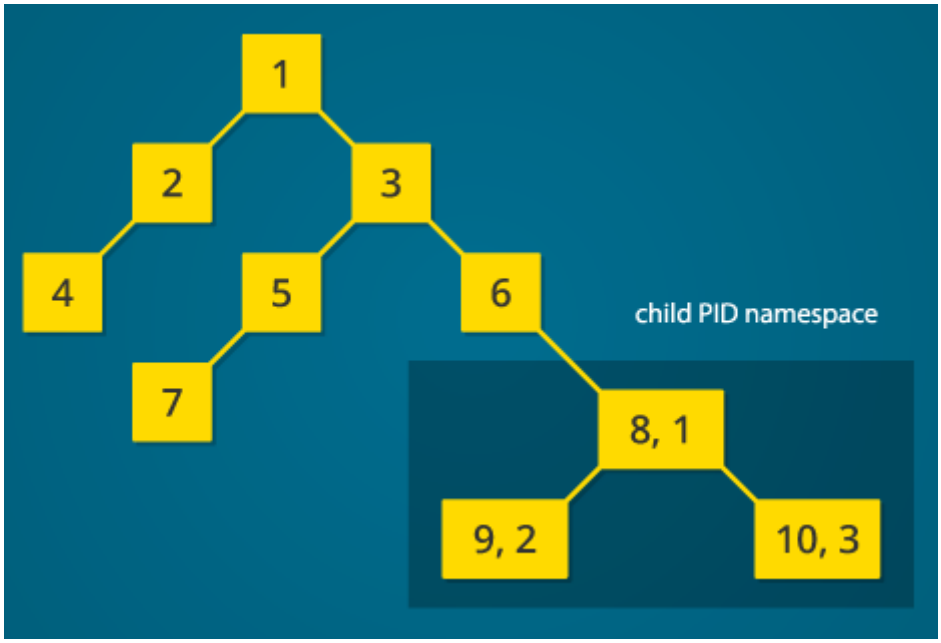


Fig. PID Namespace

Scenario,

Let's assume that you have Firefox and the Brave Browser open at the same time. You fire up your preferred search engine, such as DuckDuckGo, in each browser and search for cat pictures in one and dog pictures in the other. Both browsers are making a similar request to the same website. How does the computer ensure that the correct search results are returned to the correct browser? One of the main ways is by tracing the requests each PID makes and then returning the results to the requesting process.

B. Network Namespace (net namespace):

- Every computer that is connected to a network (such as the internet) requires an IP address. This is a unique number that allows computers to communicate effectively.
- A **network namespace** allows each of these processes to **see an entirely different set of networking interfaces**. Even the **loopback interface is different for each network namespace**.
- This doesn't automatically set up anything for you. You'll still need to **create virtual network devices and manage the connection between global network interfaces and child network interfaces**.
- Containerization software like **Docker already has this figured out and can manage networking** for you.

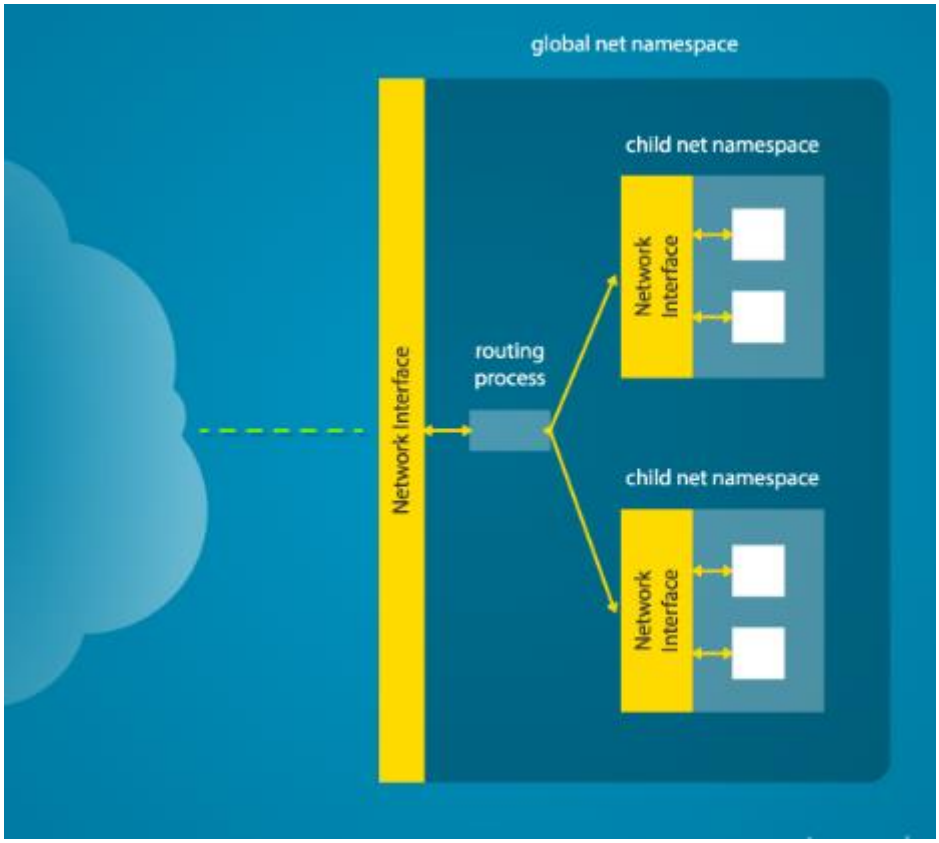


Fig. Network namespace

Scenario,

When a specific type of resource is accessed, say a web page, there is a particular port used for that communication. This is because a computer could host a web server, a game server, and perhaps an email server on the same host. The ports might be 80 or 443 for web traffic, 8888 for game traffic, and 25 for email. When I type https://<website address> into my browser, the computer translates it to send the traffic to the browser's IP xxx.xxx.xxx.xxx on port 443. The server on the other end then replies with the appropriate content via the source IP address.

Some technology stacks, as mentioned before, do not support multiple instances of the software running concurrently.

Unlike PID isolation, however, when software such as an email server is receiving a connection, it expects to be on a specific port. So even if you isolated the PID, the email server would only have a single instance running because port 25 is already in use. Network namespaces allow processes inside each namespace instance to have access to a new IP address along with the full range of ports. Thus, you could run multiple versions of an email server listening on port 25 without any software conflicts.

C. Unix Timesharing System (uts namespace)

- The UTS namespace isolates two specific identifiers of the system: **nodename** and **domainname**.
- It allows a single system to appear to have different host and domain names to different processes.
- When a process creates a new **UTS namespace**, the **hostname** and **domain** of the new **UTS namespace** are copied from the corresponding values in the **caller's UTS namespace.**
- Having multiple hostnames on a single physical host is a great help in large containerized environments.

Scenario,

Most communication to and from a host is done via the IP address and port number. However, it makes life a lot easier for us humans when we have some sort of name attached to a process. Searching through log files, for instance, is much easier when identifying a hostname. Not the least of which because, in a dynamic environment, IPs can change.

For example,

We create a **debian container** for example and we give it the name "**debian-host**" as its hostname, as we can see below:

```
root@debian:~# docker run --rm -it --name debian-host debian /bin/bash
root@vgb55e7d4s45:/$ hostname
vgb55e7d4s45
```

We gave the container a hostname from the host's **UTS namespace**, but **docker** created its own **UTS namespace** with a different hostname.

When we use the following parameter "**--uts=host**", we could **change the host's name from within the container**, because in that case the **container and the host share the same UTS namespace**:

```
docker run --rm -it --uts=host --name debian-host debian /bin/bash
```

D. User Namespace

- A **user namespace** has its own **set of user IDs and group IDs** for assignment to processes.
- The user namespace **allows a process to have root privileges within the namespace**, without giving it that access to processes outside of the namespace.
- Every computer system has some way of tracking which user owns which file(s). This allows the system to restrict access to sensitive system files. It also prevents people using the same computer from accessing each other’s files.

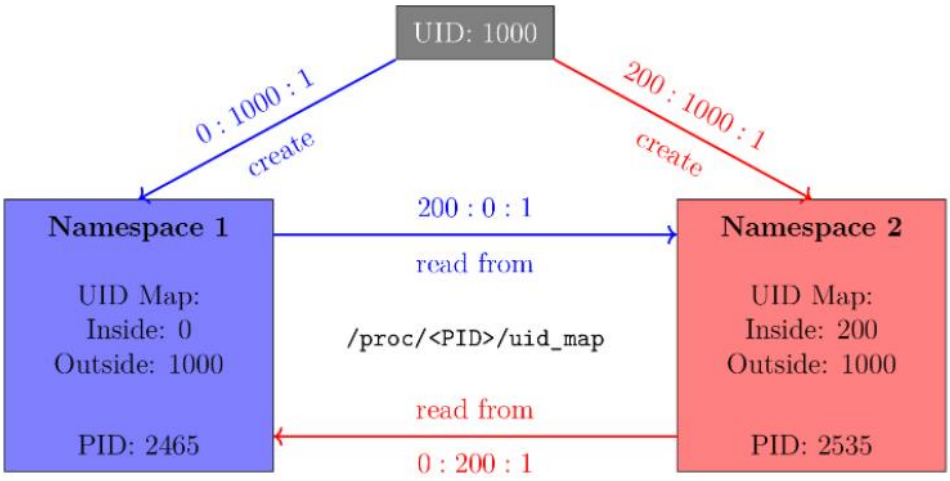


Fig. User Namespace

Depending on the namespace a process resides in, there exist differences in how the ID-outside-ns values are being interpreted. Consider the figure below illustrating this scenario. A user with UID 1000 creates two namespaces with different ID mappings. After that, the initial processes of both namespaces read the applied mappings of each other:

Consider the initial process writing to its own mapping file: To map the user identifier 1000 to its root user, a namespace has to use the same configuration as the one which namespace 1 receives above. This results in the **ID-outside-ns** value having to be interpreted as a UID of the parent namespace in case two processes reside in the same namespace.

However, if they are in different namespaces, as seen above, something else can be observed: **ID-outside-ns** is interpreted as being relative to the namespace the process that's being read from is being placed in. Namespace 1 reads **200:0:1** from namespace 2 – this means that the UID 0 in namespace 1 corresponds to the ID 200 of namespace 2 and both originate from the user identifier 1000. The same applies to namespace 2 the other way around.

It's possible to combine user namespaces with other namespace types. This enables users to create namespaces that would require **CAP_SYS_ADMIN** without being privileged. To accomplish this, one can use clone and a combination of CLONE_NEWUSER and other clone flags. The kernel processes the flag to create a new user namespace first and processes the remaining **CLONE_NEW*** flags inside the new user namespace.

Note (Security flaw):

Being privileged in a user namespace does not imply superuser access on the whole system. Nevertheless, the ability to perform actions an unprivileged user would not be able to execute without user namespaces also broadens the attack surface. For example, unprivileged users are able to execute certain mount operations that can be the target of kernel exploits. There may remain more potential security issues regarding user namespaces to be uncovered in the future. For example, it was previously discovered that the combination of user namespaces and the **CLONE_FS** flag can lead to a privilege escalation issue. More recently it was discovered that when the limit for the number **UID/GID mappings** a namespace can have was increased from 5 to 340, a security issue was introduced: When switching to different data structures to store the mappings in the kernel once the number of mappings exceeds five, this data is being

accessed in a wrong way. This results in processes of nested user namespace being able to access files being owned by other namespaces, e.g. **/etc/shadow** of the initial user namespace.

E. Mount (mnt namespace)

- The mount namespace is used to isolate mount points such that processes in different namespaces cannot view each other's files.
- The Mount namespace **virtually partitions the file system**.
- This is done at a kernel level, it's much more secure than changing the root directory with **chroot**.

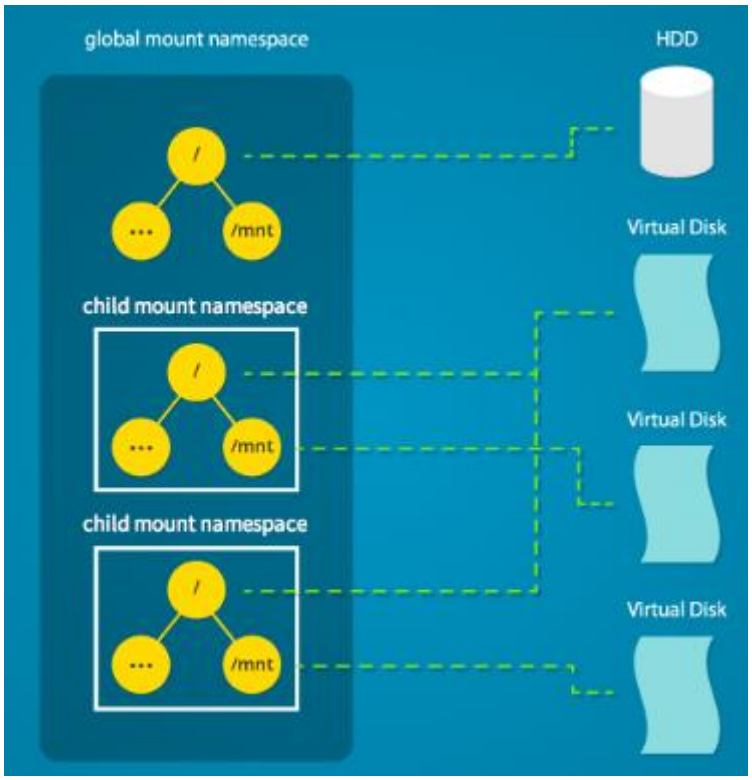


Fig. mnt namespace

Scenario,

Linux also maintains a data structure for all the mountpoints of the system. It includes information like what disk partitions are mounted, where they are mounted, whether they are **readonly**, et cetera. With Linux namespaces, one can have this data structure cloned, so that processes under different namespaces can change the mountpoints without affecting each other.

Creating separate mount namespace has an effect similar to doing a chroot(). chroot() is good, but it does not provide complete isolation, and its effects are restricted to the root mountpoint only. Creating a separate mount namespace allows each of these isolated processes to have a completely different view of the entire system's mountpoint structure from the original one. This allows you to have a different root for each isolated process, as well as other mountpoints that are specific to those processes.

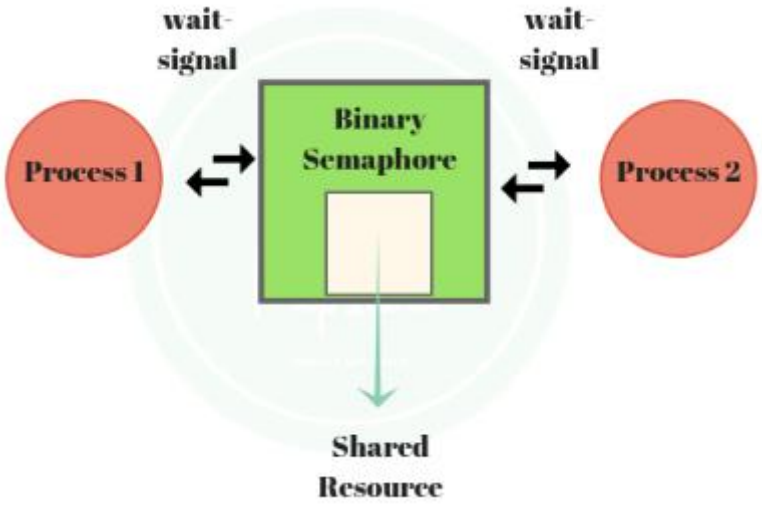
F. Interprocess communication (IPC)

- Isolating a process by the **IPC namespace** gives it its own inter process communication resources, for example, **System V IPC** and **POSIX messages**.
- This namespace controls whether or not processes can talk directly to one another.
- IPCs handle the communication between processes by using shared memory areas, message queues, and semaphores. The most common application for this type of management is possibly the use of **databases**.

Semaphore:

It is a variable or abstract data type **used to control access to a common resource by multiple processes in a concurrent system** such as a multiprogramming operating system.

Semaphores are used for communication between the active process of some applications such as Apache.



If you are facing the following error while restarting the Apache server.

- "Apache error: No space left on device: mod_rewrite: Parent could not create Rewrite Lock".
- "No space left on device: mod_rewrite: Parent could not create Rewrite Lock file /usr/local/apache/logs/rewrite_lock Configuration Failed"
- "[error] (28)No space left on device: Cannot create SSLMutex"
- "[emerg] (28)No space left on device: Couldn't create accept lock"

The error occurs if the **semaphores memory location is not free**. In most cases, the parent process dies without killing its child process properly.

If you want to check the status of the Semaphore, you can use the following command,

```
[root@server/]# ipcs -s

—— Semaphore Arrays ——-

key          semid      owner      perms      nsems
0x00000000      0       root       600        1
0x00000000  32769       apache     600        1
0x00000000  65538       apache     600        1
```

If the Apache causing any issues with the Semaphore, you need to kill the processes with user apache or nobody to start Apache properly.

########

CGroups(! Cgroup namespace)

- cgroups are a mechanism for **controlling system resources** (group of processes).
- When a cgroup is active, it can control(**limits/prioritizes**) the amount of **CPU**, **RAM**, **block I/O**, **network speed**, **disk space** and some other facets which a process may consume.
- This is a useful feature for containerized apps, but it doesn't do any kind of "**information isolation**" like namespaces would. The **cgroup namespace** is a separate thing, and only controls which cgroups a process can see, and does not assign it to a specific cgroup.
- By default, cgroups are created in the virtual filesystem **/sys/fs/cgroup**.

$ ls /sys/fs/cgroup/

```
cgroup.controllers      cpu.stat            memory.numa_stat
cgroup.max.depth        dev-hugepages.mount memory.pressure
cgroup.max.descendants  dev-mqueue.mount    memory.stat
cgroup.procs            init.scope          misc.capacity
cgroup.stat             io.cost.model       sys-fs-fuse-connections.mount
cgroup.subtree_control  io.cost.qos         sys-kernel-config.mount
cgroup.threads          io.pressure         sys-kernel-debug.mount
cpu.pressure            io.prio.class       sys-kernel-tracing.mount
cpuset.cpus.effective   io.stat             system.slice
cpuset.mems.effective   kubepods.slice      user.slice
```

- Creating a different cgroup namespace essentially moves the root directory of the cgroup.
- If the cgroup was, for example, **/sys/fs/cgroup/mycgroup**, a new namespace cgroup could use this as a root directory. The host might see **/sys/fs/cgroup/mycgroup/{group1,group2,group3}** but creating a new cgroup namespace would mean that the new namespace would only see **{group1,group2,group3}.**

Note (Security Flaw):

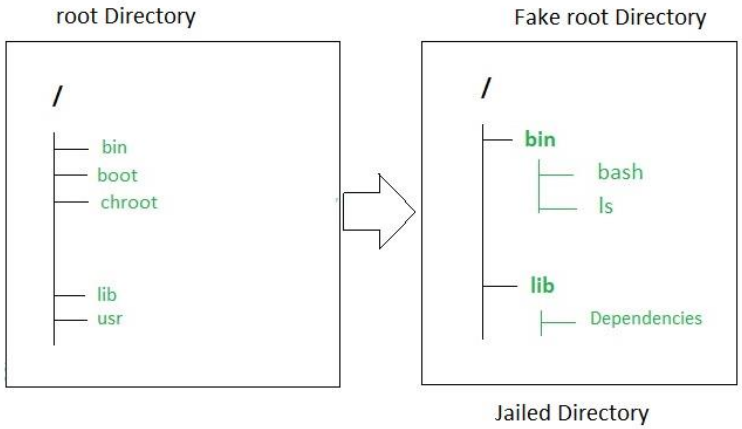
- In a traditional cgroup hierarchy, there is a possibility that a nested cgroup could gain access to its ancestor. This means that a process in **/sys/fs/cgroup/mycgroup/group1** has the potential to read and/or manipulate anything nested under **mycgroup**.

- Without the **isolation provided by namespaces**, the full cgroup path names would need to be replicated on a new host when migrating a container. Since pathing must be unique, **cgroup namespaces** help avoid conflicts on the new host system.

##################

chroot:

- **chroot** command in Linux/Unix system is used **to change the root directory**.
- Every process/command in Linux/Unix like systems has a **current working directory called root directory**.
- It **changes the root directory for currently running processes** as well as its **child processes**.
- Its effects are restricted to the **root mountpoint only**.
- A process/command that runs in such a modified **environment cannot access files outside the root directory** (like a **sanbox**).
- This modified environment is known as "**chroot jail**" or "**jailed directory**".
- Some root user and privileged process are allowed to use chroot command.



- "**chroot**" command can be very useful:
  - To **create a test environment**.
  - To **recover the system or password**.
  - To **reinstall the bootloader**.