What Are Namespaces and cgroups, and How Do They Work?

Jul 21, 2021 — by Scott van Kalken

This post is part of our series about container technology:

- *What Are Namespaces and cgroups, and How Do They Work? (this post)*
- *Building Smaller Container Images*

Recently, I have been investigating NGINX Unit, our open source multi-language application server. As part of my investigation, I noticed that Unit supports both namespaces and cgroups, which enables process isolation. In this blog, we'll look at these two major Linux technologies, which also underlie containers.

Containers and associated tools like Docker and Kubernetes have been around for some time now. They have helped to change how software is d

delivered in modern application environments. Containers make it possible to quickly deploy and run each piece of software in its own segregated environment, without the need to build individual virtual machines (VMs).

Most people probably give little thought to how containers work under the covers, but I think it's important to understand the underlying technologies – it helps to inform our decision-making processes. And personally, fully understanding how something works just makes me happy!

What Are Namespaces?

Namespaces have been part of the Linux kernel since about 2002, and over time more tooling and namespace types have been added. Real container support was added to the Linux kernel only in 2013, however. This is what made namespaces really useful and brought them to the masses.

But what are namespaces exactly? Here's a wordy definition from Wikipedia:

> *"Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources."*

In other words, 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. Mostly though, isolating services meets the architectural style of microservices as described by Martin Fowler.

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 running on a single server somewhere – but they are isolated from each other.

Types of Namespaces

Within the Linux kernel, there are different types of namespaces. Each namespace has its own unique properties:

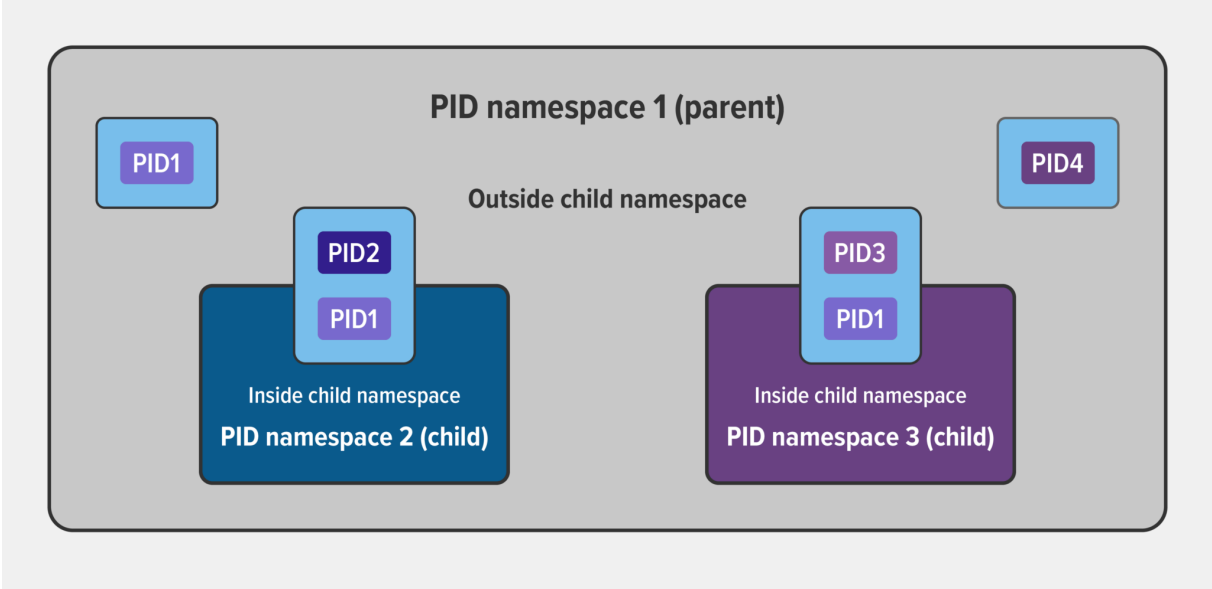
- A user namespace has its own set of user IDs and group IDs for assignment to processes. In particular, this means that a process can have `root` privilege within its user namespace without having it in other user namespaces.
- A process ID (PID) namespace assigns a set of PIDs to processes that are independent from the set of PIDs in other namespaces. The first process created in a new namespace has PID 1 and child processes are assigned subsequent PIDs. If a child process is created with its own PID namespace, it has PID 1 in that namespace as well as its PID in the parent process' namespace. See below for an example.
- A network namespace has an independent network stack: its own private routing table, set of IP addresses, socket listing, connection tracking table, firewall, and other network-related resources.
- A mount namespace has an independent list of mount points seen by the processes in the namespace. This means that you can mount and unmount filesystems in a mount namespace without affecting the host filesystem.
- An interprocess communication (IPC) namespace has its own IPC resources, for example POSIX message queues.
- A UNIX Time-Sharing (UTS) namespace allows a single system to appear to have different host and domain names to different processes.

An Example of Parent and Child PID Namespaces

In the diagram below, there are three PID namespaces – a parent namespace and two child namespaces. Within the parent namespace, there are four processes, named **PID1** through **PID4**. These are normal processes which can all see each other and share resources.

The child processes with **PID2** and **PID3** in the parent namespace also belong to their own PID namespaces in which their PID is 1. From within a child namespace, the **PID1** process cannot see anything outside. For example, **PID1** in both child namespaces cannot see **PID4** in the parent namespace.

This provides isolation between (in this case) processes within different namespaces.



Creating a Namespace

With all that theory under our belts, let's cement our understanding by actually creating a new namespace. The Linux `unshare` command is a good place to start. The manual page indicates that it does exactly what we want:

```
NAME

       unshare - run program in new name namespaces
```

I'm currently logged in as a regular user, `svk`, which has its own user ID, group, and so on, but not `root` privileges:

```
svk $ id

uid=1000(svk) gid=1000(svk) groups=1000(svk)
context=unconfined_u:unconfined_r:unconfined_t:s0-
s0:c0.c.1023
```

Now I run the following `unshare` command to create a new namespace with its own user and PID namespaces. I map the `root` user to the new namespace (in other words, I have `root` privilege within the new namespace), mount a new proc

filesystem, and fork my process (in this case, `bash`) in the newly created namespace.

```
svk $ unshare -user -pid -map-root-user -mount-proc -
fork bash
```

(For those of you familiar with containers, this accomplishes the same thing as issuing the `<runtime> exec -it </image> /bin/bash` command in a running container.)

The `ps -ef` command shows there are two processes running – `bash` and the `ps` command itself – and the `id` command confirms that I'm `root` in the new namespace (which is also indicated by the changed command prompt):

```
root # ps -ef

UID PID PPID C STIME TTY TIME CMD

root 1 0 0 14:46 pts/0 00:00:00 bash

root 15 1 0 14:46 pts/0 00:00:00 ps -ef

root # id

uid=0(root) gid=0(root) groups=0(root)
context=unconfined_u:unconfined_r:unconfined_t:s0-
s0:c0.c.1023
```

The crucial thing to notice is that I can see only the two processes in my namespace, not any other processes running on the system. I am completely isolated within my own namespace.

Looking at a Namespace from the Outside

Though I can't see other processes from within the namespace, with the `lsns` (list namespaces) command I can list all available namespaces and display

information about them, from the perspective of the parent namespace (outside the new namespace).

The output shows three namespaces – of types `user`, `mnt`, and `pid` – which correspond to the arguments on the `unshare` command I ran above. From this external perspective, each namespace is running as user `svk`, not `root`, whereas inside the namespace processes run as `root`, with access to all of the expected resources. (The output is broken across two lines for easier reading.)

```
root # lsns -output-all | head -1; lsns -output-all |
grep svk

NS TYPE PATH NPROCS PID PPID …

4026532690 user /proc/97964/ns/user 2 97964 97944 …

4026532691 mnt /proc/97964/ns/mnt 2 97964 97944 …

4026532692 pid /proc/97965/ns/pid 1 97965 97964 …
```

```
… COMMAND UID USER

… unshare -user -map-root-user -fork -pid -mount-proc
bash 1000 svk

… unshare -user -map-root-user -fork -pid -mount-proc
bash 1000 svk

… bash 1000 svk
```

Namespaces and Containers

Namespaces are one of the technologies that containers are built on, used to enforce segregation of resources. We've shown how to create namespaces manually, but container runtimes like Docker, rkt, and podman make things

easier by creating namespaces on your behalf. Similarly, the `isolation` application object in NGINX Unit creates namespaces and cgroups.

What Are cgroups?

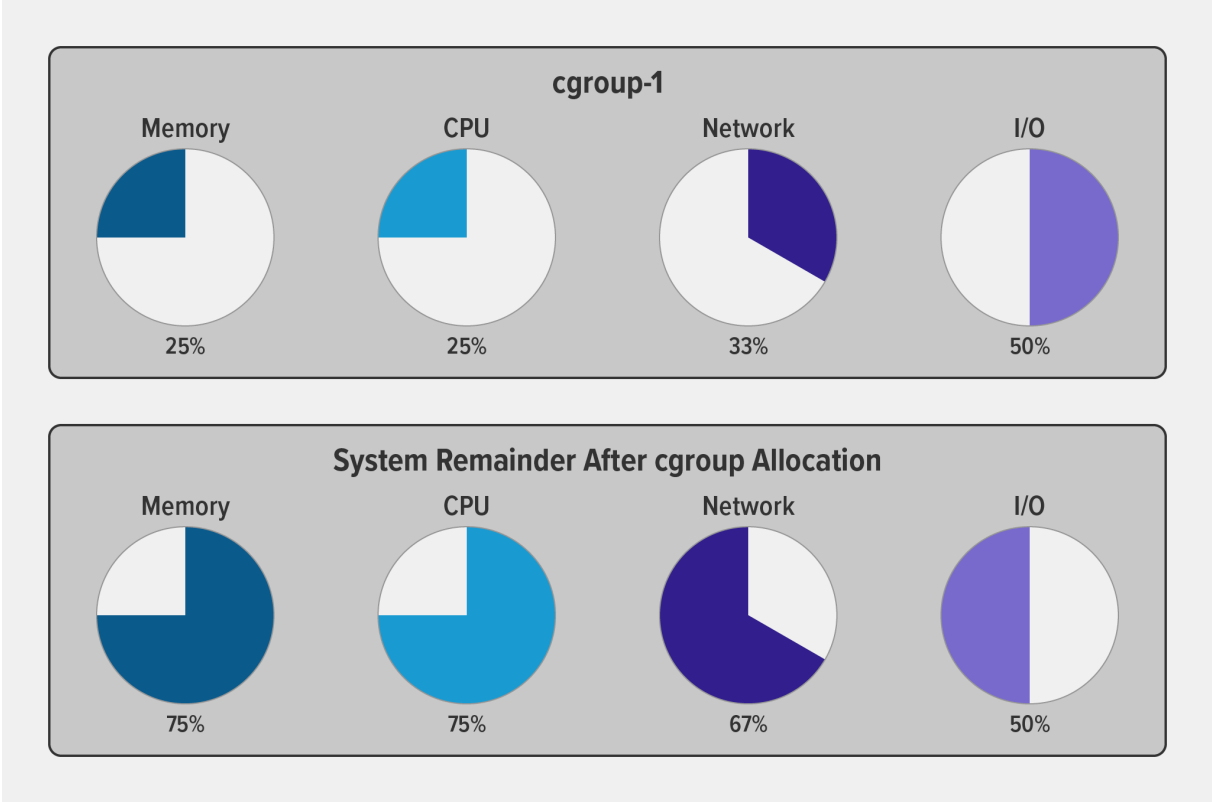
A control group (cgroup) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, and so on) of a collection of processes.

Cgroups provide the following features:

- **Resource limits** – You can configure a cgroup to limit how much of a particular resource (memory or CPU, for example) a process can use.
- **Prioritization** – You can control how much of a resource (CPU, disk, or network) a process can use compared to processes in another cgroup when there is resource contention.
- **Accounting** – Resource limits are monitored and reported at the cgroup level.
- **Control** – You can change the status (frozen, stopped, or restarted) of all processes in a cgroup with a single command.

So basically you use cgroups to control how much of a given key resource (CPU, memory, network, and disk I/O) can be accessed or used by a process or set of processes. Cgroups are a key component of containers because there are often multiple processes running in a container that you need to control together. In a Kubernetes environment, cgroups can be used to implement resource requests and limits and corresponding QoS classes at the pod level.

The following diagram illustrates how when you allocate a particular percentage of available system resources to a cgroup (in this case **cgroup-1**), the remaining percentage is available to other cgroups (and individual processes) on the system.



Cgroup Versions

According to Wikipedia, the first version of cgroups was merged into the Linux kernel mainline in late 2007 or early 2008, and "the documentation of cgroups-v2 first appeared in [the] Linux kernel … [in] 2016". Among the many changes in version 2, the big ones are a much simplified tree architecture, new features and interfaces in the cgroup hierarchy, and better accommodation of "rootless" containers (with non-zero UIDs).

My favorite new interface in v2 is for pressure stall information (PSI). It provides insight into per-process memory use and allocation in a much more granular way than was previously possible (this is beyond the scope of this blog, but is a very cool topic).

Creating a cgroup

The following command creates a v1 cgroup (you can tell by pathname format) called `foo` and sets the memory limit for it to 50,000,000 bytes (50 MB).

```
root # mkdir -p /sys/fs/cgroup/memory/foo
```

```
root # echo 50000000 >
/sys/fs/cgroup/memory/foo/memory.limit_in_bytes
```

Now I can assign a process to the cgroup, thus imposing the cgroup's memory limit on it. I've written a shell script called `test.sh`, which prints `cgroup testing tool` to the screen, and then waits doing nothing. For my purposes, it is a process that continues to run until I stop it.

I start `test.sh` in the background and its PID is reported as 2428. The script produces its output and then I assign the process to the cgroup by piping its PID into the cgroup file **/sys/fs/cgroup/memory/foo/cgroup.procs**.

```
root # ./test.sh &

[1] 2428

root # cgroup testing tool

root # echo 2428 >
/sys/fs/cgroup/memory/foo/cgroup.procs
```

To validate that my process is in fact subject to the memory limits that I defined for cgroup `foo`, I run the following `ps` command. The `-o cgroup` flag displays the cgroups to which the specified process (2428) belongs. The output confirms that its memory cgroup is `foo`.

```
root # ps -o cgroup 2428

CGROUP

12:pids:/user.slice/user-0.slice/\

session-13.scope,10:devices:/user.slice,6:memory:/foo,…
```

By default, the operating system terminates a process when it exceeds a resource limit defined by its cgroup.

Conclusion

Namespaces and cgroups are the building blocks for containers and modern applications. Having an understanding of how they work is important as we refactor applications to more modern architectures.

Namespaces provide isolation of system resources, and cgroups allow for fine-grained control and enforcement of limits for those resources.

Containers are not the only way that you can use namespaces and cgroups. Namespaces and cgroup interfaces are built into the Linux kernel, which means that other applications can use them to provide separation and resource constraints.

Learn more about NGINX Unit and download the source to try it for yourself.

← Previous: Our Roadmap for QUIC and HTTP/3 Support in NGINX

Next: Demoing NGINX at Sprint 2.0 – From Blast Off to Stable Orbit →