Systems Notebook
What's in That Container?

CORY LUENINGHOENER

Cory Lueninghoener makes big scientific computers do big scientific things, mainly looking at automation, scalability, and large-scale system design. If you don't see him hanging out with the LISA and SREcon crowd, he's probably out exploring the mountains of northern New Mexico. cluening@gmail.com

Have you ever opened your refrigerator to get a tasty snack and caught sight of that one container in the back, the one that is unmarked but you know has been there since sometime last June? And as you close the door, you kind of wonder if it just moved a little? Have you ever felt the same way about Linux containers running on your servers? What exactly is in there? And how did they get there in the first place?

Containers on Linux have been the new hotness for some time now, which I suppose makes them pretty hot indeed. But despite the ubiquity of containers today, a lot of us still only interact with them by running `docker run`. Not one to take a whale for its word, I think it's worth looking more deeply at what's really going on. While there's only room to scratch the surface, over the next several pages I'm going to take a look at what Linux containers are made of, how to create a super-simple container using a few command line tools, and how to use those same tools to understand what Docker is doing under the covers.

But first, some caveats. Mentioning "Linux containers" can cause strong reactions among some people, so I want to state upfront this isn't going to be a column about the security implications of containers, how various operating systems provided the same functionality earlier (or better), what the exact definition of a "container" is, or anything else like that. This is just a quick look at (spoiler!) how Linux namespaces are used to provide container functionality. There are some simplifications in here for the sake of brevity and clarity, so forgive me if I leave out your favorite details about Linux containers. If you want a real deep dive into everything here, take a look at the references at the end of this column.

It's All Part of the Process

Back in the age of dinosaurs, when operating systems textbooks were written, you may recall learning that a *process* is the embodiment of a program running on a UNIX-like system. It contains the program code itself, as well as its active memory, a pointer to what instruction is currently running, and various other bookkeeping data structures. A booted system starts out with a single process running, process ID (PID) 1, and all other processes on the system can trace their lineage through a series of `fork()` and `exec()` system calls back to that initial process. All running processes are given a unique PID number, and, by default, all processes exist in a global shared *namespace* that lets them see information about all other processes currently running on the system. On a UNIX-like system, much of the information about running processes is presented to users in the `/proc` file system. There, the information is organized by directories named after processes' numeric IDs.

What if, instead of process information existing in a global namespace, processes could have their own independent views of what `/proc` looked like? In this scenario, after a process forks, the parent process would be told that its child got an incrementally higher PID, while the child process would be told that it is PID 1: the first process on a fresh system. Everything else would be shared between these processes—the kernel, file systems, users—but the new process would be in a new process namespace and have a new, empty view of what other processes exist on the system.

Can You Guess My Namespace?

This world exists, and it has existed since 2007 when kernel version 2.6.24 introduced *PID namespaces*. Similar to how variable namespacing in a programming language can keep variables in one function hidden from variables in another function, namespaces in the Linux kernel can create private views of kernel data for different processes. When a process creates and joins a new PID namespace, the kernel tells it that it is PID 1 and the only process running on the system. All descendants of this new PID 1 will be put in the same PID namespace, and their view of the running system will be limited to the contents of their namespace.

There are two important things to note about this functionality: one is that PID namespaces are created in a hierarchy, much like the way that processes are created. This means that processes higher up in the namespace hierarchy can see all of the processes in PID namespaces that exist below them, while processes in leaf namespaces can only see processes that are members of their own PID namespace. The other is that multiple PID namespaces can exist at the same time, meaning a system can have many processes running on it that all believe they are PID 1.

But That's Not All

PID namespaces aren't the only namespaces that can be created, and they weren't even the first ones to be included in the Linux kernel. That distinction belongs to mount namespaces, which appeared in Linux 2.4.19 in 2002. Today there are eight namespaces available, and they all have the same goal: give processes a private view of certain system resources. Along with the PID namespace that we've already seen, this includes hostname and network information (UTS and Network namespaces), file systems (Mount namespace), system users (User namespace), and time, resource, and IPC objects (Time, Cgroup, and IPC namespaces).

In its simplest form, a Linux container is nothing more than processes in one or more private namespaces. But looking at the list of available namespaces, you can start to imagine how you could use them to turn simple processes into something that looks like an entirely new computer without relying on starting up a virtual machine.

Where Does It Come From?

Like many kernel internals related to processes, the bookkeeping that makes namespaces work is exposed to userspace in `/proc`. Any process running on a modern Linux kernel has a `/proc/[PID]/ns` directory associated with it, and the namespaces that that process belongs to are presented as symbolic links within that directory. For example, to look at the namespaces that your current shell belong to, you can do the following:

```
[root@localhost ~]ls -l /proc/$$/ns
total 0
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 cgroup ->
  'cgroup:[4026531835]'
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 ipc ->
  'ipc:[4026531839]'
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 mnt ->
  'mnt:[4026531840]'
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 net ->
  'net:[4026531992]'
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 pid ->
  'pid:[4026531836]'
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 pid_for_children ->
  'pid:[4026531836]'
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 user ->
  'user:[4026531837]'
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 uts ->
  'uts:[4026531838]'
```

The numbers that the links point to are unique identifiers for each of the namespaces on the system, and any processes that share those numbers also share that particular namespace.

A program can use the `clone()` system call function with appropriate flags to create a copy of itself in a new namespace, ready to be replaced with a call to `exec()`. Meanwhile, an existing process can use the `unshare()` call to create and join private, non-shared namespaces or `setns()` to join existing namespaces. Each of these functions accepts a set of flags that specify what new namespaces to create.

Alternatively, the `unshare` and `nsenter` command line tools, provided by the `util-linux` package, can be used to create processes that are in new namespaces or members of existing namespaces from the command line. These tools provide command-line options to control which namespaces are created or joined.

Let's Get Our Hands Dirty

Let's take a look at namespaces on a real system. Using the `unshare` command line tool, it is easy to create a new process with one or more namespaces that are unique from its parent. We'll start by creating a new shell that's in a new PID namespace, but shares its other namespaces with its parent. With that new shell created, we can look at what processes are visible both inside and outside this miniature "container" and how to add more processes to it.

You can follow along on your own system: all of these examples were run on a CentOS 8 virtual machine booted up using Vagrant and VirtualBox, and for clarity each line of the shell session has been prefixed with `[o]`, for outside of the new namespace, or `[i]`, for inside the new namespace.

First, we'll use `unshare` to create a new shell in a new PID namespace.

```
[o] [root@localhost ~]unshare --fork --pid --mount-proc
  /bin/bash
[i] [root@localhost ~]#
```

At this point, the unshare command has started a new copy of bash with a new PID namespace. Both the old shell and the new shell have the same prompt, so it's kind of anticlimactic. But recall that in the previous section we saw that the list of namespaces a process belongs to is exposed in /proc/<PID>/ns. If you compare the new shell's namespaces against the shell in the previous section, you can see that the new shell is indeed in a new PID namespace:

```
[i] [root@localhost ~]ls -l /proc/$$/ns
[i] total 0
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 cgroup ->
  'cgroup:[4026531835]'
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 ipc ->
  'ipc:[4026531839]'
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 mnt ->
  'mnt:[4026532155]'
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 net ->
  'net:[4026531992]'
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 pid ->
  'pid:[4026532156]'
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 pid_for_children
  -> 'pid:[4026532156]'
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 user ->
  'user:[4026531837]'
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 uts ->
  'uts:[4026531838]'
```

Looking closely, you'll notice that the new shell is also a member of a new Mount namespace. The unshare man page explains why in its PID namespace section:

"It also implies creating a new mount namespace since the /proc mount would otherwise mess up existing programs on the system."

So we gained two namespaces for the price of one.

Since this new shell is a member of a new PID namespace, the only processes it knows about are itself, which it sees as PID 1, and its descendants. We can see this by running the ps command:

```
[i] [root@localhost ~]ps -ef
[i] UID          PID    PPID  C STIME TTY          TIME
  CMD
[i] root           1       0  0 03:41 pts/0    00:00:00
  /  bin/bash
[i] root          18       1  0 03:44 pts/0    00:00:00
  ps -ef
```

Meanwhile, this same process is visible from the system's default namespaces with a different PID number. It just takes some sleuthing to find it. Starting up another login shell on the system, we can find the namespaced process by looking for the original unshare process and examining its only child. Here we find that the parent namespace identifies our namespaced shell as PID 33783:

```
[o] [root@localhost ~]ps -ef
[o] UID          PID    PPID  C STIME TTY          TIME
  CMD
    ...
[o] root       33782    5283  0 03:41 pts/0    00:00:00
  unshare --fork --pid --mount
[o] root       33783   33782  0 03:41 pts/0    00:00:00
  /bin/bash
    ...
```

After one process creates a new namespace, other processes can join it. Having found our new container process from outside of its PID namespace, we can also start a new shell within its new PID namespace. With the original namespaced bash process still running via unshare, we can use the nsenter command to join it by targeting its external process ID:

```
[o] [root@localhost ~]nsenter --all --target 33783 /bin/bash
[i] [root@localhost /]ps -ef
[i] UID          PID    PPID  C STIME TTY          TIME
  CMD
[i] root           1       0  0 03:41 pts/0    00:00:00
  /bin/bash
[i] root          19       0  0 03:47 pts/1    00:00:00
  /bin/bash
[i] root          34      19  0 03:47 pts/1    00:00:00
  ps -ef
```

Let's review what all we just did: starting with a fresh virtual machine, we created a new process in a new PID namespace, confirmed that it appeared as PID 1, and started another new process inside that same namespace. Now, let's take it a step further.

Let's Build a Simple Container

Let's get one step closer to a full Docker-style container by building a new operating system image and starting processes using it. CentOS includes the debootstrap package, which can be used to install a full Ubuntu system inside of a single directory tree on a CentOS system. We can use that tool to create an Ubuntu file-system image in /root/ubuntu-bionic, and then use unshare along with chroot to create a shell with new Mount and PID namespaces in use. Once that shell is running, it will look exactly like it is running on an Ubuntu system. This can all be done from within a clean CentOS 8 install in a virtual machine.

```
[o] [root@localhost ~]yum install epel-release
    <output trimmed>
[o] [root@localhost ~]yum install debootstrap
    <output trimmed>
[o] [root@localhost ~]mkdir ubuntu-bionic
[o] [root@localhost ~]debootstrap --arch=amd64 bionic
  /root/ubuntu-bionic/ http://mirrors.vcea.wsu.edu/ubuntu/
    <output trimmed>
[o] I: Base system installed successfully.
```

```
[o] [root@localhost ~]unshare --fork --pid --mount-proc
  --mount chroot /root/ubuntu-bionic /bin/bash
[i] root@localhost:/mount -t proc proc /proc
[i] root@localhost:/mount -t sysfs sysfs /sys
[i] root@localhost:/ps -ef
[i] UID          PID    PPID  C STIME TTY          TIME
  CMD
[i] root           1       0  0 03:54 ?        00:00:00
  /bin/bash
[i] root          13       1  0 03:54 ?        00:00:00
  ps -ef
[i] root@localhost:/head -2 /etc/os-release
[i] NAME="Ubuntu"
[i] VERSION="18.04 LTS (Bionic Beaver)"
```

This still doesn't fully replicate the full containerization provided by tools like Docker, but we've started to get close. In the last several sections, we've essentially run `docker build` (using `debootstrap`), `docker run` (using `unshare`), and `docker exec` (using `nsenter`). As homework, you can expand this work by combining this same set of commands with other namespaces, giving you the ability to change the hostname, assign private network interfaces, and more.

Now, let's try the same tricks with a real Docker container.

What Was That about Docker?

Way back in the first paragraph of this column, I asserted that many people's main interface to containers is `docker run`. Since then, we've learned that containers are just processes with unique namespace configurations that give them the ability to see different root file systems, different process trees, and the like. When Docker starts a container, it uses the exact same kernel mechanisms we just looked at to get the job done. That means that you can use these same tools to interact with Docker containers, but without the Docker commands. As an example, let's use `nsenter` to replicate the base functionality provided by `docker exec`. As with the earlier examples, everything here was done within a CentOS 8 virtual machine built using Vagrant and VirtualBox.

To start, we'll fire up a simple Docker container and start a sleep inside it so that the process is easy to find:

```
[o] [root@localhost ~]docker run -it ubuntu bash
[i] root@94802998616b:/sleep 300
```

We can find this same process from outside of the container, just like we did before:

```
[o] [root@localhost ~]ps -ef | grep sleep
[o] root       52039   51999  0 04:00 pts/0    00:00:00
  sleep 300
```

Using the `nsenter` command, we can start a new shell that joins all of the same processes that the `sleep` command is a member of:

```
[o] [root@localhost ~]nsenter --all --target 52039 /bin/bash
[o] root@94802998616b:/ps -ef
[o] UID          PID    PPID  C STIME TTY          TIME
  CMD
[o] root           1       0  0 03:59 pts/0    00:00:00
  bash
[o] root           8       1  0 04:00 pts/0    00:00:00
  sleep 300
[o] root           9       0  0 04:01 ?        00:00:00
  /bin/bash
[o] root          12       9  0 04:01 ?        00:00:00
  ps -ef
[o] root@94802998616b:/
```

The new shell is now a member of the Docker container, complete with the container's hostname (`94802998616b`) and the only four processes it knows about (two instances of `bash`, plus `sleep` and `ps` processes). We've just replicated the base functionality of `docker exec` with standard Linux utilities.

Conclusion

Building containers by hand is more of an interesting trick than something that's useful in production, but knowing what's going on underneath Docker, Buildah, Podman, and other container tools gives you greater insight into how to tune, debug, and work with those tools. By understanding the underlying technology and how to access it with lower-level tools, you have a better overall view of how your system works and how to keep it running optimally.

References

If you want to dig deeper into Linux namespaces, here are two great places to start:

Namespaces(7) Linux manual page: https://man7.org/linux/man-pages/man7/namespaces.7.html.

M. Kerrisk, "Namespaces in Operation" series, *The Linux Weekly News*, January 4, 2013: https://lwn.net/Articles/531114/.