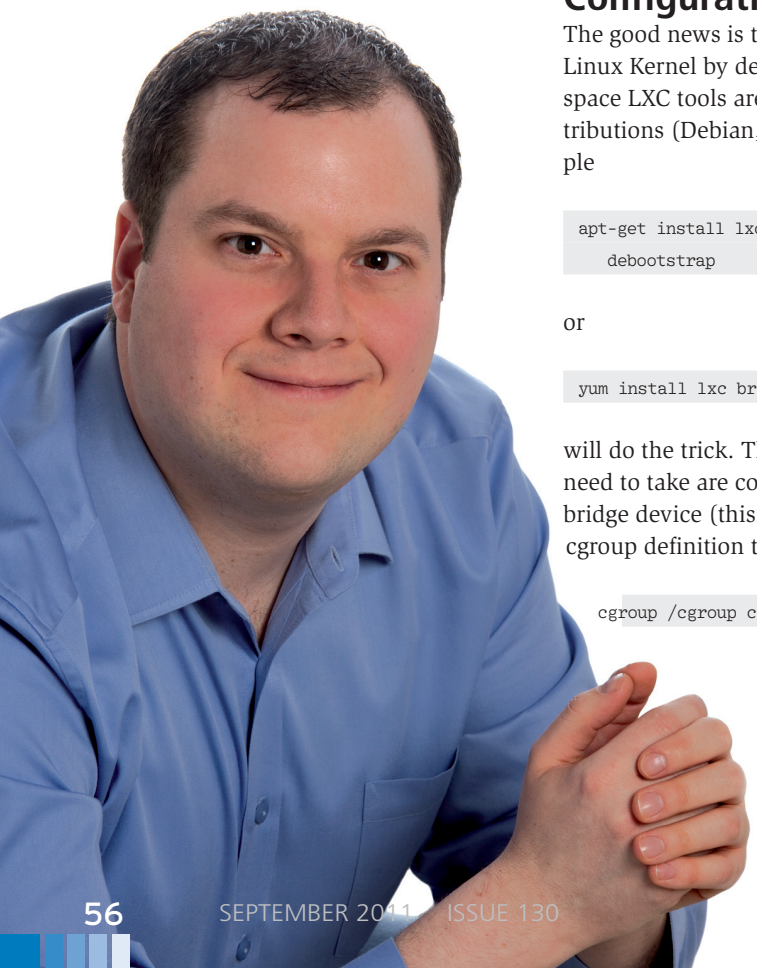
Fleet and flexible cgroups and Linux Containers

Virtual Host

The big virtualization tools like KVM and Xen can't compete on a small scale with resource-spare cgroups and Linux Containers. *By Kurt Seifried*

Normally when I talk about virtualization in Linux, I go straight to system virtualization using tools like KVM or Xen, to name two. But some interesting options aren't quite as well known. One such example is cgroups [1] and Linux Containers [2] (LXC), which sits on top of cgroups. In a nutshell, LXC uses cgroups to create a restricted view of the host operating system. Within the LXC guest environment, you can only see what the admin allows you to see of the host system; you can have a separate process space, for example and also create a separate filesystem for the guest.

So why would you want to use a technology like LXC instead of a full system virtualization platform like KVM or Xen? LXC has several advantages: For one thing, it has virtually no overhead, and it provides a degree of flexibility because of its ability to share resources between different LXC guests. (I know, these seem counterintuitive because the goal is to segregate them from each other,) Also, LXC supports not only virtualizing a running instance of an operating system (more on this later) but also individual applications, for which devoting an entire virtual machine is overkill. To see a good example of this, you can read more about what Google is doing with ChromeOS [3].



Installation and Basic Configuration

The good news is that cgroups is in the Linux Kernel by default, and the user-space LXC tools are included in most distributions (Debian, Fedora, etc.). A simple

```
apt-get install lxc bridge-utils ➥
   debootstrap
```

or

```
yum install lxc bridge-utils
```

will do the trick. The next steps you will need to take are configuring a network bridge device (this is optional), adding a cgroup definition to fstab,

```
cgroup /cgroup cgroup defaults 0 0
```

and then creating the directory and mounting it

```
mkdir -p /cgroup
mount /cgroup
```

with the preceding two commands.

Creating an LXC Container

Now comes the tricky part: you'll need to create an LXC container to host the application or the operating system. It is important to note that when running an operating system under LXC, you won't be running a separate kernel and so on, because the guest running inside of LXC will essentially be a separate "view" of the entire operating system running on the system (this can be as separate or as shared as you want). The easiest way to create an environment within an LXC instance is to use a tool such as `debootstrap`, which also includes specific LXC templates for setting up instances.

You will need to correct the template file `/usr/lib/lxc/templates/lxc-debian` so it installs squeeze (Debian GNU/Linux 6) instead of lenny (Debian GNU/Linux 5). This setp will also require you to change the dhcp package name, and you'll also need to add the lxc tools (and libcap2 so lxc-setcap works) to the image so that it can be started correctly. The abridged patch file is:

```
-dhcp-client,\
+isc-dhcp-client,\
+lxc,\
+libcap2,\
-lenny $cache/partial-$arch ➥
   http://ftp.debian.org/debian
+squeeze $cache/partial-$arch ➥
   http://ftp.debian.org/debian
```

KURT SEIFRIED

Kurt Seifried is an Information Security Consultant specializing in Linux and networks since 1996. He often wonders how it is that technology works on a large scale but often fails on a small scale.

Now you'll be able to create quickly a bare-bones instance of Debian for LXC to use. Simply run the commands:

```
mkdir -p /var/lib/lxc/vm0
/usr/lib/lxc/templates/lxc-debian ➥
   -p /var/lib/lxc/vm0/
```

To execute it, run:

```
lxc-start -n vm0
```

By default, the root password is `root`. Then, you can configure the system, add additional packages, and modify as needed. Alternatively, before starting the LXC hosted guest, you can use `chroot` to enter the filesystem of the guest you created. The advantage of this is that you'll have access to the host networking, and you can easily change root's password, install additional packages, and generally configure the system as if it were running and you were logged in to it:

```
chroot /var/lib/lxc/vm0/ /bin/bash
```

Creating a container for an application is a similar process; however, you will most likely not want to replicate an entirely separate system. By mounting directories, such as `/lib` inside the guest as `/lib` with read-only access, for example,

```
lxc.mount.entry = /lib ➥
  /var/lib/lxc/vm0/rootfs/lib none ➥
  ro,bind 0 0
```

you can prevent a guest from making changes and avoid having to update more than one system.

LXC Network Configuration

Network configuration is one of the trickier parts of LXC configuration because you have several options. LXC supports five kinds of network interface: `empty`, `veth`, `vlan`, `macvlan`, and `phys`. The `empty` type simply sets up a loopback interface, `veth` will attach an interface within the guest to a bridge interface hosted on the system, and `vlan` will attach an interface to one on the host. The `macvlan` interface is more interesting: It binds the interface to one on the host with a virtual MAC address assigned to it, allowing virtually any type of network activity (multiple IP addresses bound to the interface, spoofing, sniffing, etc.) to take place. Generally speaking, you'll probably want to use `veth`, which offers a degree of control and can prevent abuse to some degree. For more flexibility, `macvlan` allows virtually any network operation you would normally have access to. This topic is covered in detail in the `lxc.conf` man page, which also contains some excellent examples.

Improving LXC Performance

Say you want to set up virtual web hosting for a few dozen clients and have decided to use LXC. Unlike KVM or Xen, you can easily share directory structures in a read-only configuration, meaning you can deploy a template system (with standard `/lib`, `/usr`, etc.) used by all the running instances (which would, of course, have their own `/etc`, `/var`, and other directories that they actually need to write to). On its own, this setup will work quite well, especially if Linux caches all the files into memory, but you can also use `tmpfs` to mount the template system into memory, meaning, the majority of standard read-based disk I/O on client systems, will never touch a drive, speeding things up considerably. The only disk I/O that will actually occur is clients serving files, writing logfiles, etc.

Other LXC Tips and Tricks

LXC also offers some other tricks that are especially useful for developers and security folks such as myself. The `lxc-freeze` and `lxc-unfreeze` commands can be used to freeze the system and then resume operations. Combined with tools like `diff`, this method can be used to compare system states – for example, before and after installing a program – or to test a binary-only program. Another thing that comes to mind is the possibility of shipping applications within an LXC container – the only required external dependency being the system kernel. With the `lxc-execute` command, a user can start the program and even have access to their local X session (so things will "just work," which is nice). And for confining interactive shell accounts, LXC works much better than chroot.

Also, if you want to use LXC as a user, you will need to run `lxc-setuid` and `lxc-setcap`, both on the host and within any guest systems running LXC. Otherwise, things like `lxc-init` will fail because they do not have the needed privileges to mount the `/proc` filesystem, for example.

cgoups

But what about cgroups, the underpinning of LXC? cgroups can be used not only to restrict the view of the system but can be also used to enforce quotas on resource usage such as CPU, memory, and disk I/O. Traditionally, disk quotas (how much disk you can use) have always been supported, but with no restrictions on how you use it. A program that causes large amounts of small random writes can severely affect system performance by causing the disk drives to thrash, and on a shared system, especially one with paying customers, this will not do. Interacting with cgroups is a lot like using the `proc` psuedo-filesystem on Linux. cgroups creates a virtual filesystem, and from this, you can read settings (`cat /cgroups/`*`some-setting`*) and set settings (`echo "foo" > /cgroups/`*`some-setting`*). For a complete list of settings and options, refer to the documentation (it's extensive).

One Last Trick

Like most people, I get nervous about giving out shell access to users. But with LXC, you can create a container for a user, then simply create a shell script to launch it when the user logs in and set that shell as their login shell (remember to add it to `/etc/shells`):

```
#!/bin/bash
/usr/bin/lxc-execute -n vm0 /bin/bash
```

When the user logs in, they get dropped into an LXC environment that can easily be restricted but can also give them access to `setuid` programs and so forth safely. ■■■

INFO

[1] cgroups: *http://www.kernel.org/doc/Documentation/cgroups/*

[2] Linux Containers: *http://lxc.sourceforge.net/*

[3] System hardening: *https://sites.google.com/a/chromium.org/dev/chromium-os/chromiumos-design-docs/system-hardening*