SYSADMIN

System administration technologies brought to you from the coalface of Linux.

Jonathan Roberts dropped out of an MA in Theology to work with Linux. A Fedora advocate and systems administrator, we hear his calming tones whenever we're stuck with something hard.

When new products are being developed, one of the most important choices to be made is which technologies you're going to build on top of. Obviously, the right tool for the job is the correct answer, but how you decide which tool is the right one is less obvious.

As a sysadmin, my goal is to ensure that our technologies are the most secure, the most stable and the most familiar. All of these characteristics will reduce the chances of me being woken early in the morning, and when I am inevitably woken in the morning, I'll at least have the knowledge and experience to have half a chance of fixing whatever went wrong.

For developers, however, operational requirements often seem restrictive. Enterprise distributions come with technology that's more than three years old and requires the developers to write far more lines of code. This means there'll be more bugs and they may well take much longer to get the product to market.

During the two years I've worked as a system administrator, I've seen this conflict crop up several times, but never seen any obvious solutions that make both parties happy. I agree with the 'DevOps' crowd, who argue that part of the solution is to 'better align incentives', making operations more responsible for delivering software and developers more responsible for maintaining it.

We also need improvements from distributions, providing more reliable means for getting recent, but stable, software packages on to well tested platforms. The Fedora.next wheeze may be going some way to address this, as do PPAs in Ubuntu. Hopefully we'll see this mentality begin to translate to Debian and RHEL in the coming years, too.

Linux containers

Enterprise-grade virtualisation on a real kernel.

While Linux containers have been around for a while, they've recently been gaining more recognition as a lightweight alternative to traditional virtualisation products like KVM or VMWare. With the arrival of LXC, Docker, and the next generation of distributions, we're all likely to see a lot more of them over the coming decade.

As with all virtualisation, the idea of containers is to make it easy to run multiple applications on a single host, all the while ensuring each remains separate. This enables the administrator to carefully manage the resources assigned to each application and to ensure that they can't interfere with each other.

What makes containers different to traditional products is that they don't do any hardware emulation. Instead, the applications in question all run directly on top of the host kernel, just like any other process. Separation between the running containers is achieved through the careful use of a number of Linux kernel features.

Control Groups (cgroups) are the first of these features, and are probably the best known. They provide a means for administrators to group processes, and all their future children, into hierarchical groups. Various subsystems can then be used to strictly manage the processes and the resources they interact with.

Control groups

If you have systemd installed, you can quickly inspect what cgroup your processes are running in with the **ps** command:

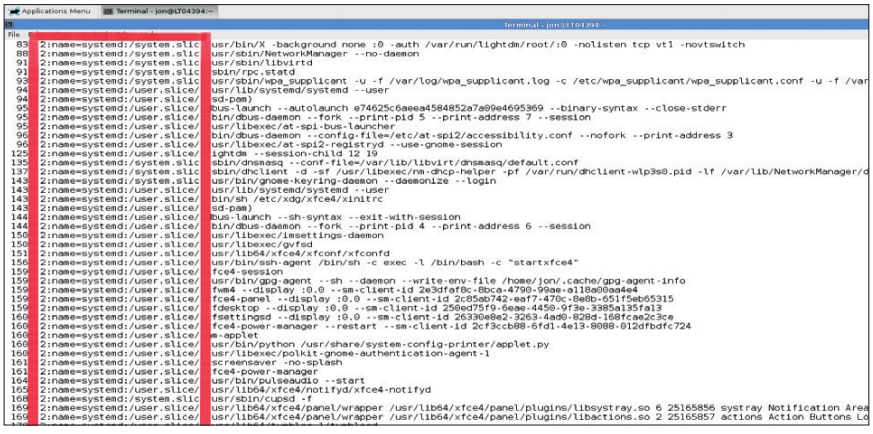
```
ps -aeo pid,cgroup,command
```

Running this, you should see that all processes are running in cgroups that exist in a hierarchy below the systemd cgroup. You could use systemd unit files to manage the resources assigned to a service (indeed, if you're using systemd, this is probably the best way to use cgroups), as described in the last issue, but you can also interact with cgroups directly, too.

There are a collection of tools available in the **libcgroup-tools** package, including **cgcreate**, for example. You can use this tool to create a new cgroup as follows:

```
cgcreate -g memory,cpu:mysql
```

This will create a new cgroup called **mysql** which has been tied to the **memory** and **cpu** subsystems. You can then take advantage



The highlighted area shows the cgroup in which the different processes are running. As you can see, all are either in the systemd defaults of **systemd:/user.slice** and **systemd:/system.slice**.

of a command, such as **cgset**, or interact directly with the virtual filesystem exposed by cgroups, to manipulate the resource limits of this newly created group:

```
cgset -r swappiness=xxx /sys/fs/cgroups/memory/mysql
```

This command will set the swappiness parameter of all processes running in the **mysql** cgroup to **xxx**. To add a process to the cgroup, all you need to do is echo its PID to the **tasks** file in the cgroup's filesystem or use the **cgclassify** command.

Namespace isolation

Namespace isolation is the other key technology that makes containers possible on Linux. Each namespace wraps a particular system resource, and makes processes running inside that namespace believe they have their own instance of that resource. There are six namespaces in Linux:

- **mount** Isolates the filesystems visible to a group of processes, similar to the chroot command.
- **UTS** Isolates host and domain names so that each namespace can have its own.
- **IPC** Isolates System V and POSIX message queue interprocess communication channels.
- **PID** Lets processes in different PID namespaces have the same PID. This is useful in containers, as it lets each container have its own init (PID 1) and allows for easy migration between systems.
- **network** Enables each network namespace to have its own view of the network stack, including network devices, IP addresses, routing tables etc.
- **user** Allows a process to have a different UID and GID inside a namespace to what it has outside.

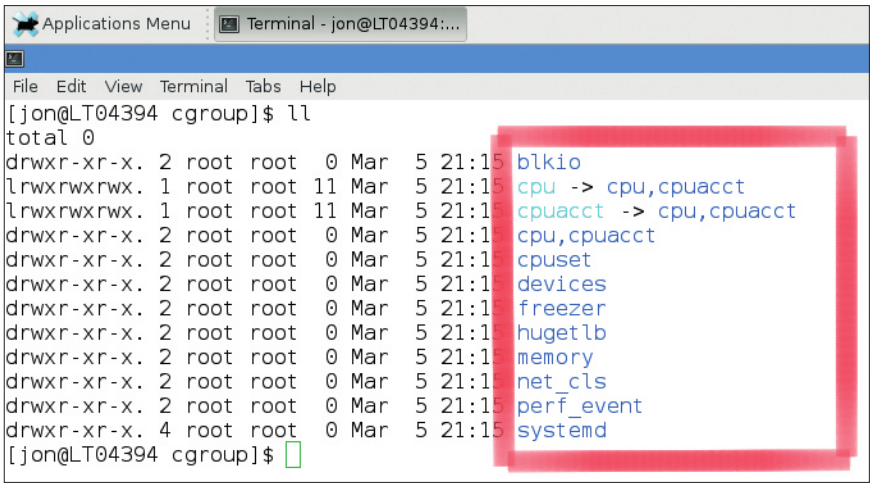
A quick way to experiment with namespaces yourself is to use the **unshare** command. This will run a particular program, removing its connection to a particular namespace of its parent:

```
sudo unshare -u /bin/bash
```

This will create a new bash process that doesn't share its parent UTS namespace. If you now set the hostname to **foo**, you'll then be able to look, in another shell on the same system, and see that the hostname in the root (original) namespace hasn't changed.

Linux containers

Now that you have an idea of what the underlying technologies do, let's take a look at Linux Containers (LXC), a userspace interface that brings them together. To install the LXC userspace tools, you need to install the **lxc** package on Ubuntu and Fedora, but in the case of the latter, you should also install **lxc-templates** and **lxc-extras** for a better experience.



The output of this long listing in the /sys/fs/cgroup directory shows all the different subsystems that are available for managing processes with cgroups on a default Fedora 20 installation.

Once that's done, creating a new container, depending on your requirements, can be simple. In the **/usr/share/lxc/templates** directory, you'll find a collection of scripts that will create some default containers, including Debian, Fedora and Ubuntu system containers, and sshd, BusyBox and Alpine application containers. To put one of these to use, all you need to do is run **lxc-create**:

```
lxc-create -n linux-voice -t /usr/share/lxc/templates/busybox --dir /home/jon/containers/linux-voice
```

- **-n** sets the name of the container.
- **-t** says which template you want to use.
- **--dir** says where you want the rootfs for the new container to be created.

This command creates a directory in **/var/lib/lxc** with the name set by the **-n** flag. The contents of this directory are populated by the script specified with the **-t** flag. If you look at, say, the BusyBox template, you'll see that this script sets up a filesystem hierarchy, copies appropriate binaries and installs important pieces of configuration with **heredoc** statements.

Inside the created directory, you'll also find that a config file has been created. This defines which system resources are to be isolated and controlled by the container. The **man lxc.conf** command goes in to detail on what options can be put in this file, but a few key examples will be helpful:

- **lxc.cgroup.cpu.shares = 1234** Sets the share of CPU that the container has.
- **lxc.utsname = linux-voice** Sets the hostname of the container.
- **lxc.mount.entry = /lib /home/jon/containers/busybox/lib** Specifies directories on the host filesystem that should be mounted in the container.

This configuration file means you can apply the existing templates in quite flexible ways, but if you really want to create a custom container, you're going to have to set to work creating your own template script.

> **"Creating a Linux container is paradoxically easier than creating an application container."**

As the LXC man page says, creating a system container is paradoxically easier than creating an application container. In the latter case, you have to start by figuring out which resources you want to isolate from the rest of the system, and then figure out how to populate the appropriate parts of the file system etc. In the former case, you simply isolate everything – much simpler.

Once you've created your container with **lxc-create** and modified the config file as you see fit, you can start it with the **lxc-start** command, use **lxc-console** to get a console in it, and shut it down with **lxc-shutdown**.

While cgroups and namespaces have reached a degree of maturity in Linux, the user experience still has some room for improvement. If you found the **lxc-** commands tricky to use, you might want to install **libvirt-sandbox**, which will provide a set of scripts and extensions for using LXC through the familiar **libvirt** tools.