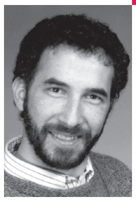
PETER BAER GALVIN

Solaris 10 containers



Peter Baer Galvin is the chief technologist for Corporate Technologies, Inc., a systems integrator and VAR, and was the systems manager for Brown University's Computer Science Department. He is currently contributing editor for *SysAdmin Magazine*, where he manages the Solaris Corner, and is co-author of the *Operating Systems Concepts* and *Applied Operating Systems Concepts* textbooks.

pbg@petergalvin.info

THE CONCEPT IS SIMPLE: ALLOW multiple copies of Solaris to run within one physical system. Indeed, creating and using Solaris zones is simple for experienced system administrators. But then why are there so many questions surrounding this new Solaris feature? Just what is a container? How do you upgrade it? How can you limit its resource use? Does it work like VMware? And so on. In this article I describe the theory of Solaris 10 containers, and the facts behind creating, managing, and using them.

Overview

Solaris 10 containers are a new feature of Solaris. A container is a virtualized copy of Solaris 10, running within a Solaris 10 system. While this is similar in concept to VMware, for example, it is more of a distant relative than a blood brother. Perhaps its closest relative is BSD jails. Containers have a different purpose from VMware (and other operating system virtualization software, such as Xen). Those tools create a virtual layer on which multiple operating systems can run. Containers run within Solaris 10, act only as Solaris 10 environments, and only run Solaris applications.

Before I delve further into containers, a clarification is needed. Solaris 10 has the concepts of "zones" and "containers." Simply, a container is a zone with resource management (including fair-share scheduling) added. Mostly the terms are used interchangeably, but where needed I will point out differences.

Given that limited Solaris 10 view of virtualization, of what use are containers? Consider the following set of features that containers add to Solaris 10:

- Up to 8192 containers can exist on the same system. The "global zone" is what would normally be called "the operating system." All other containers are referred to as "non-global."
- Containers can share resources with the global zone, including binaries and libraries, to reduce disk-space requirements. An average container takes 60MB of disk space or less. A container sharing files with the global zone is known as a "sparse container" and is created via a sparse install.
- Because binaries are shared (by default), and Solaris optimizes memory use by sharing objects in memory when possible, an average container uses approximately 60MB of memory when booted.

- By default a package (Sun's concept of an installable application) installs in the global zone and all nonglobal zones. Likewise, a patch will by default install in all containers that contain the packages to which the patch refers.
- As mentioned above, a container is a resource-managed zone. The first release of Solaris 10, on which this article is based, includes CPU use as a manageable resource. (Note that there are now three streams of Solaris release: the standard commercial release; the "Express" updates that arrive every month or so, and for all intents are beta releases available to anyone interested; and the Open Solaris community release, which is a periodic snapshot of the internal build of Solaris based on the Open Solaris release, and is the least tested of these. This latter release is the most recent of the builds, but also the most likely to have problems.)

As with other OS-virtualizing technologies, containers are secure from each other, allowing only network access between them. They also have their own network configurations, having their own IP addresses and allowing variations in network membership (subnets and network masks, for instance).

There are also some limits that come with containers (most of these are likely to be removed in future releases of Solaris):

- A container cannot be an NFS server.
- A container cannot be moved to another system (i.e., imported or exported).
- A container must have a pre-set network address (i.e., it cannot be DHCP-configured).
- The only container-manageable resource as of the first release of Solaris is CPU shares. A container could, for example, use all the system's virtual memory.
- Due to the security restriction that a nonglobal container be securely separate from other containers, some features of Solaris 10 do not work in a container. The most limiting is DTrace, the extraordinary Solaris 10 debugging/analysis tool.
- A container cannot run Linux binaries natively (the initial container marketing from Sun to the contrary notwithstanding). Likewise not currently supported, a container cannot have its own firewall configuration. Solaris 10 uses ipfilters as its firewall technology; ipfilters can be configured in the global zone only.
- By definition, a container runs the same operating system release as the global zone. Even kernel patches installed on the system affect all containers. Only application patches or non-kernel operating system patches can vary between containers.

So what we are left with is a very lightweight, easy to manage, but in some ways limited application segregation facility. The same application could be run in multiple containers, use the same network ports, and be unaware of any of its brothers. A container can crash and reboot without affecting any other containers or the global zone. An application can run amok in a container, eating all available CPU but leaving the other containers with their guaranteed fair-share scheduling slices. A user with root access inside of a container may control that container, but cannot escape from the container to read or modify the global zone or any other containers. In fact, a container looks so much like a traditional system that it is a common mistake to think you are using the global zone when you are "contained." An easy navigation solution is found in the command zonename. It displays the name of the current zone. I suggest you have that output displayed as part of your prompt so that you always track the zone you are logging in to.

Creation

The creation of a container is a two-step process. The zonecfg command defines the configuration of a container. It can be used interactively or it can read a configuration file, such as:

```
create -b
set zonepath=/opt/zones/zone00
set autoboot=false
add inherit-pkg-dir
set dir=/lib
end
add inherit-pkg-dir
set dir=/platform
end
add inherit-pkg-dir
set dir=/sbin
end
add inherit-pkg-dir
set dir=/usr
end
add inherit-pkg-dir
set dir=/opt/sfw
end
add net
set address=131.106.62.10
set physical=bcme0
end
add rctl
set name=zone.cpu-shares
add value (priv=privileged,limit=1,action=none)
end
```

In the above example, I create zone00. It will not automatically boot when the system boots. It will be a sparse install, inheriting the binaries, libraries, and other static components from the global zone, for all of the inherit-pkg-dir directories added. It will have the given network address, using the Ethernet port known as bcme0. Finally, it will have a fair-share scheduler share of 1. (See below for more information on the fair-share scheduler.) The command zonecfg –z zone00 –f config-file will read the configuration (from the file named "config-file"). Either a sparse install or a full container install is possible. If there are no inherit-pkg-dir entries, all packages from the global zone are installed in full. Such a container can take 3GB or more of storage but is then less dependent on the global zone. Typically, sparse installation is used.

Several options are available with zonecfg. An important one is fs, which mounts file systems within the container. Not only can file systems from other hosts be mounted via NFS, but loopback mounts can be used to mount directories from the global zone with the container. For example, to mount /usr/local read only as /opt/local in zone00, interactively:

```
# zonecfg –z zone00
zonecfg:zone00> add fs
zonecfg:zone00:fs> set dir=/usr/local
zonecfg:zone00:fs> set special=/opt/local
zonecfg:zone00:fs> set type=lofs
zonecfg:zone00:fs> add options [ro,nodevices]
zonecfg:zone00:fs> end
```

When the zone is rebooted, the mount will be in place.

Next, zoneadm –z zone00 install will verify that the configuration information is complete, and if it is will perform the installation. The installation for the most part consists of package installations of all of the packages in the inherit-pkg-dir directories, but only those parts of the packages that are nonstatic (configuration files, for example). The directory under which the container will be created must exist and must have file mode 700 set. Typically, a container installation takes a few minutes.

Once the container is created, zoneadm –z zone00 boot will boot the container. The first boot will cause a sysidconfig to execute, which by default will prompt for the time zone, root password, name services information, and so on. Rather than answer those questions interactively, a configuration file can do the trick. For example:

```
name_service=DNS
{
domain_name=petergalvin.info
name_server=131.106.56.1
search=arp.com
}
network_interface=PRIMARY
{
hostname=zone00.petergalvin.info
}
timezone=US/Eastern
terminal=vt100
system_locale=C
timeserver=localhost
root_password=bwFOdwea2yBmc
security_policy=NONE
```

Placing this information in the sysidcfg file in the /etc directory of the container (/opt/zones/zone00/root/etc/sysidcfg) before the first boot provides most of the sysifconfig answers. Now the container can be booted. To connect to the container console, as root, you can use zlogin –C zone00. Here you can watch the boot output. Unfortunately, there is still an NFSv4 question to answer for sysidconfig, but once that is done the container is up and running. The first boot also invokes the new Solaris 10 service management facility, which analyzes the container and adds services as prescribed by the installed daemons and startup scripts. Future boots of the container only take a few seconds.

Management

Once a container has been installed and booted, it is fairly self-sufficient. The zoneadm list command will show the status of one or all containers.

I find a script to execute a command against every container is helpful. For example:

```
#!/bin/bash
for z in zone00 zone01 zone02 zone03 zone04
zone05 zone06 zone07 zone08 zone09 zone10
zone11 zone12 zone13 zone14 zone15 zone16
zone17 zone18 zone19 zone20;
do
zoneadm -z $z boot
zlogin -z $z hostname;
done
```

Note that the zlogin command, when run as root in the global zone, can execute a command in a specified zone without a password being given.

Another administrative convenience comes from the direct access the global zone has to the other containers' root file systems. Root in the global zone can copy files into the directory tree of any of the containers via the container's root directory.

By default, any package added to the global zone is added to every container on the system. Likewise, any patch will be added to every container that contains the files to which that patch applies. It is a bit surprising to watch pkgadd boot each installed but nonrunning container, install the packages, and then return the container to its previous state. But that is an example of just how well integrated into Solaris 10 containers are. There are options to the pkgadd and patchadd to override this behavior.

Fair-Share Scheduler

As previously mentioned, the fair-share scheduler can be used in conjunction with zones to make them into containers. A container then may be limited in its share of available CPU scheduling slices. Fair share is a complex scheduling algorithm in which an arbitrary number of shares are assigned to entities within the computer. A given entity (in this case a container, but it could be a collection of processes called a project) then is guaranteed its fair share of the CPU—that is, its share count divided by the total. If some CPU is unused, then anyone needing CPU may use it, but if there is more demand than CPU, all entities are given their share. There is even the notion of scheduling memory, in that an entity that used more than its fair share may get less than its share for a while when some other entity needs CPU.

The system on which a container runs must have the fair-share scheduler enabled. First, the scheduler is loaded and set to be the default via dispadmin –d FSS. The fair-share scheduler is now the default on the system (even surviving reboots), and all new processes are put into that scheduling class. Now all the current processes in the timesharing class (the previous default) can be moved into fair share by priocntl –s –c FSS –i class TS. Finally, I'll give the global zone some shares (in this case, five) so it cannot be starved by the containers: prctl –n zone.cpu-shares –v 5 –r –i zone global. To check the share status of a container (in this case, the global zone), use prctl –n zone.cpu-shares –i zone global. Take note that the prctl command share setting does not survive reboots, so this command should be added to an rc script to make it permanent.

Use

Containers are new, and therefore best practices are still in their infancy. Given the power of containers and the low overhead, I would certainly expect that most Solaris 10 systems will have containers configured. An obvious configuration is that only root users access the global zone, and all other users live in one or more containers. Another likely scenario is that each application be installed in a nonglobal container, or each into its own container, if it only communicates with the other applications on that system via networking. If a set of applications uses shared memory, then they obviously need to be in the same container. Much will depend on the support by software creators and vendors of containers. Early indications are good that vendors will support their applications being installed inside containers.

Some of the design decisions surrounding containers will limit how they are used. For example, it does not make sense to install a development, QA, and production container for a given application on the same system. Development might want to be using a different operating system release than the one currently in use in production, for example. Or QA may need to test a kernel patch. It would be reasonable to create a container for each developer on a development server, however, to keep rogue code from crashing that shared system or hogging the CPU. For those that host applications for others (say, application service providers), containers are a boon for managing and securing the multiple users or companies that use the applications on their servers.

Conclusion

The Solaris 10 container facility is a different take on virtualized operating system environments commonly found on other systems. Only one Solaris release may run on the system, and even kernel patches affect all containers. Beyond that, containers offer a cornucopia of features and functions. They are lightweight, so they may be arbitrarily created and deleted. They are secure, protecting themselves from other containers and protecting the global zone from all containers. And they can be resource managed (although only CPU resources at this point) to avoid a container from starving others. Further, containers are integrated into other aspects of Solaris 10, such as package and patch management. Containers are a welcome addition to Solaris 10 and allow for improved utilization of machines, as well as more security and manageability.