Understanding Docker

KURT LIDL

Kurt Lidl is a Principal Member of the Technical Staff at Oracle, working on the Oracle Public Cloud build team. He started using BSD UNIX with 4.2 BSD, and now contributes as a Committer on the FreeBSD Project. He lives in Potomac, Maryland, with his wife and two children. lidl@freebsd.org

Editor's note: A version of this article appeared in the July/August 2017 issue of the *FreeBSD Journal*.

Docker, from Docker Inc., is a popular containerization software system for building, deploying, and running Linux applications. Docker containers offer a relatively low overhead mechanism for running multiple Linux applications where the different applications are isolated from each other. Docker offers a high-level interface to configure, build, store, and fetch Docker images. This article contains a brief review of popular virtualization technologies, an example of Docker's facilities for building containers, and a brief discussion of Docker's future evolution.

Overview of Virtualization Techniques

There are many different virtualization techniques available across the operating systems in use today. The most basic virtualization is the concept of a software process. This is the traditional virtualization that UNIX and many other operating systems have provided to different processes from early on: each user process has an independent, protected-access memory map provided through the virtual memory system of the kernel. Other more comprehensive virtualization techniques—such as software, hypervisor-based, hypervisor-based with hardware acceleration, and containerized applications—will be reviewed.

Software Virtualization/Emulation

Software-based, complete machine emulators, such as QEMU and SIMH, can emulate practically any CPU and machine architecture on the hosting machine. These types of emulators are generally fairly slow but offer complete independence between the emulated hardware and the hosting machine. The emulation software provides an instruction-by-instruction emulation of the target machine and provides a software implementation of the hardware devices of the target machine. For example, disk drives on the target are often emulated with plain files on the hosting machine. Even machines that no longer have operating hardware, such as the Honeywell DPS8M, can be emulated. In this case, the emulation is of sufficient fidelity to allow the historically significant Multics operating system to run on the emulated machine with no software changes. Another significant example of this type of emulator was the Connectix Virtual PC software, which could emulate a complete x86 computer, hosted on a PowerPC-based Mac computer. The Connectix company was purchased by Microsoft, however, and the software is no longer available.

Hypervisor-Based Virtualization

At the opposite end of the virtualization spectrum are hypervisor-based implementations. A hypervisor-based virtualization generally runs at a significant percentage of the native speed of the hosting machine. Only a small set of hypervisor-mediated system functions execute in the hypervisor, and the rest of the user code runs in the virtualized machine at native speeds. This type of virtualization is considered fairly "heavyweight" in that each virtualized machine has its own copy of whatever operating system is being run (Figure 1). One area of performance issues with this scheme is that the virtualized operating system

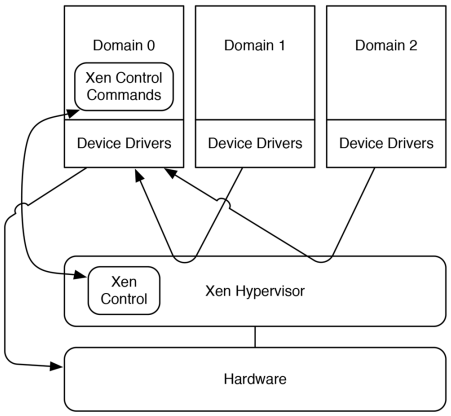

Figure 1: Xen architecture

also has to maintain its own set of memory protections for its own use. Hardware support for this type of operation, sometimes called "nested page tables," greatly enhances the operation of guest operating systems under the hypervisor.

There are many hypervisor-based virtualization platforms available, including:

- bhyve on FreeBSD
- KVM on Linux
- xhyve on Mac OS X
- Hyper-V on Microsoft Windows
- ESXi and vSphere from VMware
- Xen on multiple operating systems
- Several hardware architectures

Containerized Virtualization

Containers are lightweight virtualization schemes where processes have some sort of partitioning and isolation between different administrative groups on the same host. The different partitions all share a single kernel application binary interface (ABI) running against a single kernel instance. Often, but not always, each process running in a container can be seen on the hosting server. This type of virtualization is generally called "container computing" and offers a middle ground between the level of isolation from hypervisors and the "shared everything" from a standard UNIX environment.

Containerization is the fundamental idea behind the following facilities:

- Jail system on FreeBSD
- Control Groups on Linux
- Containers on Nexenta OS
- Containers on Solaris

Hybrid Virtualization Techniques

There are other hybrid virtualization techniques, such as running a combination of hypervisor virtual machines and then hosting various containerized applications on those virtual machines. This hybrid approach is how Docker is implemented on non-Linux machines such as the Mac OS version of Docker, which is built on top of the Mac OS xhyve virtualized machine. In a similar fashion, the Windows implementation of Docker uses the Hyper-V hypervisor to create a virtual machine running Linux, which is then used to execute the system calls from the Docker containers.

Linux Control Groups and Docker

The Linux kernel has a relatively new capability that makes Docker possible: Control Groups (aka "cgroups"). This is the fundamental technology that allows for the isolation of various user processes in one control group from affecting and directly interacting with a different control group.

In a traditional UNIX environment, there is a single hierarchy of user processes. The `init` process (pid 1) is the root of that hierarchy, and all processes can trace their ancestry back to that initial process. The cgroups facility in the Linux kernel allows for instantiating new hierarchies of processes that are contained entirely in the new hierarchy and can only interact with other processes in that hierarchy.

The cgroups facility can do more than just create new process hierarchies; it can set up resource limits (e.g., memory and network bandwidth) and attach these limits to the process hierarchies that are created. While management of the low-level cgroups mechanism via provided system utilities is possible, it is rather tedious. Docker provides a more convenient interface for controlling the cgroups mechanism at runtime, along with an easy-to-use system for building the static environments that will be executed later.

Docker uses cgroups, along with other Linux kernel facilities, such as iptables, for networking configuration and control and for a union file system (UnionFS) for isolating the container from the file systems of the hosting machine. There is also a mechanism available to allow explicit sharing of directories between the host machine and the Docker containers. The UnionFS that Docker implements is layered on top of a Docker storage driver. The storage drivers that are available depend on the particular Linux system that is running Docker and provide varying degrees of performance and stability.

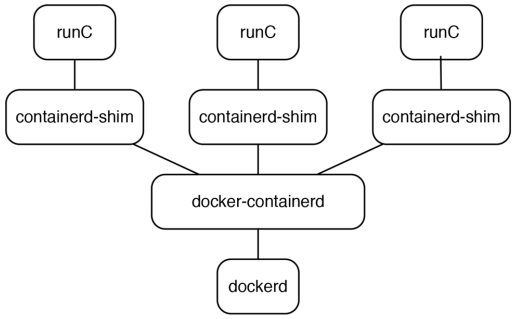

Figure 2: Docker process relationships

Docker Terminology and Software Architecture

An *image* is what Docker calls the containerized file system that has been created and loaded with the software layers that are needed for a particular application. When an image needs to run, a copy-on-write snapshot of the image is created, and that copy-on-write file system is called the *container*.

The process that starts a container is then placed into a new cgroup hierarchy. Any new processes spawned by the initial process in a Docker container will not be able to influence any processes outside of the running container, because all other processes will belong to different cgroups. This isolation prevents any interaction or interference between two or more Docker containers running on the same physical host.

A modern Docker installation typically has at least two long-running daemons, `dockerd` and `docker-containerd`. There is a single user command, `docker`, that takes multiple command keywords. This is similar to how many complex systems are controlled through a single dispatch command (e.g., `git`, `hg`, and `rndc`). The `docker` command communicates through a UNIX domain socket to the `dockerd` process. The `dockerd` process communicates with the `docker-containerd` process to specify the management of the containers on a system. There are other container software shims that are started for each container. The `runC` container runtime system initializes and starts the container, and then hands the file descriptors for `stdin/stdout/stderr` over to `containerd-shim`, which acts as a proxy of sorts between the running container and `docker-containerd` (Figure 2). This intermediate process is required so that a restart of the `dockerd` process, and therefore, the restart of `docker-containerd`, can allow the new `docker-containerd` daemon to reattach to the `containerd-shim` for each currently running container.

Docker Images Explained

A Docker image is a virtual file system, packaged as a series of layers. Each layer in the file system is stacked on top of the layers underneath it. The ultimate view of the file system is the union of the file systems that make up a Docker image. The layers in the image are built from the commands in a Dockerfile.

Dockerfile as a Recipe

The Dockerfile is a simple text file, holding one or more commands, and any comments that the user has placed in the file. When Docker builds an image, it runs each of the commands found in the file in the order they are encountered. In this manner, the `docker` build procedure is just like following a step-by-step recipe for preparing a meal. Each of the commands in the Dockerfile will generate a new layer in the resulting image. By convention, the Docker commands are written in uppercase to help differentiate them from user-specified commands. If any of the commands that are executed fail (that is, has a non-zero exit code), the building of the image stops immediately, and the image build is marked as a failure. For efficiency reasons, it is desirable to keep each layer in the Dockerfile as small as possible. This means that cleaning up after any commands that create large amounts of metadata, such as `yum update`, should be done as part of the same command that generated the metadata.

This example Dockerfile will create an image with six layers. Some of those layers, which are identical to the prior layer, will be automatically discarded during the build process.

```
An image for running Apache
FROM centos:7
MAINTAINER Ms. Nobody <nobody@example.com>
RUN yum -y --setopt=tsflags=nodocs update && \
    yum -y --setopt=tsflags=nodocs install httpd && \
    yum clean all
VOLUME ["/var/www/html", "/var/log/httpd"]
EXPOSE 80
CMD ["/usr/sbin/apachectl", "-DFOREGROUND"]
```

The first command, `FROM centos:7`, which creates the first layer in the image, specifies that the base image for CentOS 7 should be pulled from the central Docker repository into the local machine's cache of file-system layers. This layer is the bottom layer in the image. The `FROM` command must be the first command in a Dockerfile and initializes a new build.

The second command, `MAINTAINER ...`, sets a special label in the metadata for the image. This label is used to identify the creator of the image. There is also a `LABEL` command that could be used instead to set an arbitrary number of labels on an image. The labels can be used by the end user for any purpose.

The third command, `RUN yum -y update ...`, updates any out-of-date software packages that were included in the base image. The next part of the command, `yum -y install httpd`, installs the Apache `httpd` package. The final part of the command, `yum clean`, expunges all the package/repository metadata maintained by the `yum` package management system to minimize the size of the generated layer. For the same reason, the `yum` command, using the `nodocs` flag, is instructed to ignore any documentation during the upgrade and installation of packages. The arguments

to the RUN command are executed by a shell process, so the complexity of the generated layer in the constructed image can be quite elaborate.

The fourth command, VOLUME [...], marks a list of directories to be used as external mountpoints in the UnionFS file system. During container execution, these mountpoints will have external file systems mounted at these locations. The UnionFS layer should not attempt to capture that activity to the copy-on-write file system. This is one method for how a container can persist data outside of the copy-on-write image from which the container is executing.

The fifth command, EXPOSE 80, provides information that will be used when a container is started from this image. A port on the hosting machine can be mapped to the specified TCP port number of the container at runtime. Or, by specifying a different networking option at runtime, the port of the container can be made accessible to other containers running on the same host.

Finally, the sixth command, CMD ..., specifies the default command to be executed when a container is started from this image. In this case, it starts the Apache Web server in the foreground. When the Apache Web server process exits, the container will be automatically stopped. It is often useful to create a small wrapper script around a daemon that is started inside a Docker container in order to restart the daemon if it stops running. By automatically restarting the daemon, the Docker container can continue to run without needing to be restarted.

Now that the purpose for each of the lines is known, building an image is straightforward. Note that some of the output from the build process has been removed and lines wrapped to improve readability. The image is created by running the command docker build *directory*, where directory is the path to the directory holding the Dockerfile.

```
docker build -t centos-apache-testimage .
Sending build context to Docker daemon  2.048kB
Step 1/6 : FROM centos:7
 ---> 3bee3060bfc8
Step 2/6 : MAINTAINER Ms. Nobody <nobody@example.com>
 ---> Using cache
 ---> 7f88dbad6a42
Step 3/6 : RUN yum -y --setopt=tsflags=nodocs update &&
        yum -y --setopt=tsflags=nodocs install httpd &&
        yum clean all
 ---> Using cache
 ---> f50595808f75
Step 4/6 : VOLUME ["/var/www/html", "/var/log/httpd"]
 ---> Running in bce2b6331fc8
 ---> 51b4c07c8eba
Removing intermediate container bce2b6331fc8
Step 5/6 : EXPOSE 80
 ---> Running in 073e6fac8709
 ---> 5bf7cadf8102
Removing intermediate container 073e6fac8709
Step 6/6 : CMD /usr/sbin/apachectl -DFOREGROUND
 ---> Running in e5a44065f0d7
 ---> 4d119d3a4776
Removing intermediate container e5a44065f0d7
Successfully built 4d119d3a4776
Successfully tagged centos-apache-testimage:latest
```

Docker Image Inspection

It is instructive to look at some of the metadata about that image, via the docker inspect command. Not all the metadata is shown in this output, just some of the more interesting pieces.

```
docker inspect centos-apache-testimage:latest
[
   {
      "Id": "sha256:db9314a42feb [...]",
      "RepoTags": [
         "centos-apache-testimage:latest"
      ],
      "ContainerConfig": {
         "ExposedPorts": {
            "80/tcp": {}
         },
         "Env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:"
         ],
         "Cmd": [
            "/bin/sh",
            "-c",
            "#(nop) ",
            "CMD [\"/usr/sbin/apachectl\" \"-DFOREGROUND\"]"
         ],
         "Volumes": {
            "["/var/www/html",": {},
            ""/var/log/httpd"]": {}
         }
      },
      "DockerVersion": "17.06.0-ce",
      "Author": "Ms. Nobody <nobody@example.com>",
      "Architecture": "amd64",
      "Os": "linux",
      "Size": 275797466,
      "GraphDriver": {
         "Data": null,
         "Name": "aufs"
      },
      "RootFS": {
         "Type": "layers",
         "Layers": [
```

```
            "sha256:dc1e2dcd [...]",
            "sha256:41fc3fb9 [...]"
         ]
      }
   }
]
```

The `ContainerConfig` section has the complete environment specified for the processes in any containers that are started from this image. The `Architecture` and `Os` settings show that the containers support the Linux syscall interface, for the amd64 (aka x86 64) machine type. The Docker image for this article was created on a Macintosh computer, running macOS Sierra 10.12.5, but any containers will be executed with a Linux/amd64 runtime environment.

This image could be moved to any host capable of running Linux/amd64 Docker images. The portability of images is one of the principal advantages of Docker—ease of building and deploying across many different hosts without having to worry about shared library conflicts or corrupting configurations of already installed components. Docker supports the image registries where images may be stored and retrieved. A private Docker registry can be created that allows users to centrally store their customized images. Once the image is stored in the registry, a single command can retrieve the image to a host, and a second command can start a container from that image.

Running a Container

It is easy to create a running container from the example image:

```
docker run --rm -d -p 8080:80 \
   -v $(pwd)/htdocs:/var/www/html \
   -v $(pwd)/logs:/var/log/httpd \
   centos-apache-testimage:latest
```

This command starts the container, telling Docker to throw away the copy-on-write file system (`--rm`) when the container exits. The command runs the container in the background (`-d`) and port-maps localhost:8080 to the container's TCP port 80 (`-p 8080:80`). The command also performs volume mounts of `$(pwd)/htdocs` to the `DocumentRoot` of the Web server (`-v $(pwd)/htdocs:/var/www/html`), and mounts `$(pwd)/logs` to hold the log files from the Web server running in the container (`-v $(pwd)/logs:/var/log/httpd`). Finally, the name of the image to be used as the initial file system for the container is listed.

Looking at Running Containers

Once a container has been started, it will run until the initial process that started the container exits. The user who started the container, or the system administrator, can stop the container via the `docker stop` command. This sends a `SIGTERM` to the initial process in the container, and then a `SIGKILL` after a grace period, if the container has not stopped. This is very similar in intent to running the shutdown command on a UNIX host. The `docker kill` command just sends a `SIGKILL` to the root process in the container.

The `docker ps` command gives the list of running containers. The `docker rm` command can be used to remove the copy-on-write file system of a stopped container.

Looking at Images on the Machine

The `docker images` command will show the list of images currently available on the host.

The `docker rmi` command can be used to remove a reference to an image, freeing the storage associated with that image when the reference count drops to zero. Note that shared layers in the image may also be used by other images, so there often isn't a one-to-one correspondence between the amount of space listed as in use for the image and the amount of disk space used by the image. Many of the issues with double-counting of storage blocks that occur with file system snapshots are also evident with the `docker-containerd` storage of images.

Other Docker Commands

There are other Docker commands available that can be used for launching containers automatically and maintaining a set of containers that must run together to accomplish a given task. It is beyond the scope of this article to fully example the complete set of commands available inside of Docker. More advanced orchestration of multiple containers running across multiple hosts is possible via the `Docker Swarm` support in recent versions of Docker.

Comparison with FreeBSD Jails

Docker containers are similar to FreeBSD jails in terms of what virtualization is provided and how machine resources are shared. Both offer compartmentalized processes running against a single kernel image on the hosting machine (Figure 3). Docker offers an easy-to-use command line interface for creating, deploying, running, and updating images. Little setup and configuration are required on the hosting machine, other than the basic Docker Engine installation. FreeBSD jails have a considerably simpler interface to running and stopping jails. The base FreeBSD system offers essentially no high-level support for the building and installation of jails into a directory. There are several add-on FreeBSD ports (e.g., ezjail, qjail, and qjail4) that attempt to make jail usage less cumbersome, to varying degrees of success.

One significant advantage that Docker has over jails is the concept of spawning a per-instance copy-on-write file system for each container that is started. This is fundamental to the deployment and reusability of Docker images, whereas each FreeBSD jail typically runs in a persistent file system tree. Some of the add-on jail management systems use ZFS's snapshot and promote features to create a clone of a prototype file system

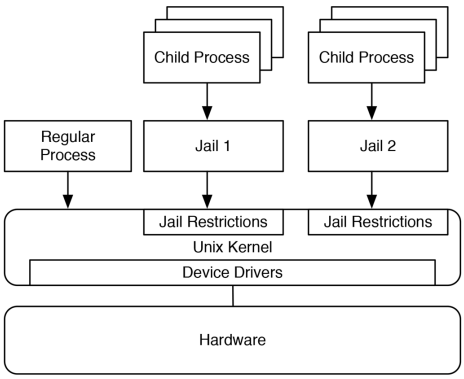

Figure 3: Jail architecture

for a newly instantiated jail, but that clone still persists the file hierarchy across multiple restarts of the jail.

With Docker containers, the Docker infrastructure takes care of mounting various directories when the container is started, whereas mounting of any directories, even including the crucial `devfs /dev mount`, must be handled explicitly for each FreeBSD jail.

For instance, running the example Docker container for Apache, there are several mountpoints active:

```
df
```

File system	1K-blocks	Used	Available	Use%	Mounted on
none	61890340	863500	57859916	2%	/
tmpfs	1023384	0	1023384	0%	/dev
tmpfs	1023384	0	1023384	0%	/sys/fs/cgroup
/dev/sda2	61890340	863500	57859916	2%	/etc/hosts
shm	65536	0	65536	0%	/dev/shm
osxfs	976426656	624064128	352106528	64%	/var/www/html
tmpfs	1023384	0	1023384	0%	/sys/firmware

The Docker system manufactured the required mounts automatically, with the exception of the `/var/www/html` mount, which was specified on the command line when the container was started.

The security benefits of Docker vs. jails are roughly equivalent. The security features of Docker are typically modified through command-line flags, while the security features for jails are either globally specified via sysctl settings or have per-jail configuration settings in the /etc/jail.conf file. Jails, when operated with the VIMAGE networking option, have a per-instance network stack. This implies that each jail could have different packet filtering in place. All the running containers on a Linux-based host share a single iptables-based packet filter configuration.

The control aspects of Docker vs. jails are quite different. Docker has many commands and options to allow almost all configurations to happen on the command line. FreeBSD jails rely heavily on the contents of the /etc/jail.conf file to specify which jails are to be run and how they are to be configured. Docker internalizes much of the configuration that is the metadata for a given Docker image. By attaching this metadata to the image, the deployment to a new host is significantly eased. FreeBSD jails have no such metadata directly attached to each jail.

In a related area, some of the FreeBSD ports for helping to manage bhyve virtualized host instances, such as iohyve, offer some of the same type of configuration help. These systems use ZFS properties to attach the metadata about a virtualized machine to a ZFS file system or ZFS zvol, which represents the file system for the virtualized machine. Several of the earliest versions of these management tools just used well-known names for the parameters that were to be controlled: hostname, number of CPUs, amount of memory, and so forth. None offered a generic, extensible tag:value configuration file that could be attached to a ZFS file system, although this might have changed since the earliest attempts at supporting virtual machine metadata in this manner.

Future Directions for Docker

The Docker system is undergoing fairly rapid evolution. There is a consortium of companies that have formed the Open Container Initiative (OCI). Significant members of the OCI include Amazon, AT&T, Cisco, Docker Inc., Facebook, Google, IBM, Microsoft, Oracle, Red Hat, and VMware. OCI is attempting to standardize both a container runtime system ("runtime-spec") and an image specification ("image-spec"). As a starting point for the standardization process, Docker Inc. donated their container specification, and their runtime system (runC) to OCI. Some of the members of OCI have vested interests in supporting more than just a Linux syscall ABI container, and the specifications are clear in the need to support multiple ABIs, as well as multiple operating systems hosting the container runtime. A recent development in the standardization process is Oracle's release of an open-source implementation of the `oci-runtime` called Railcar, which is written in Rust.

Docker on FreeBSD

Examining the current state of the Docker system, it seems that there are no insurmountable technical impediments to making the Docker system run natively on FreeBSD. The future of Docker and the support for different ABIs across containers implies that supporting a native FreeBSD kernel ABI for the containers would be possible. Obviously, this makes deployment using Docker less of a Linux/amd64 monoculture. Currently, Docker is effectively only running the Linux ABI on amd64 hardware. The Docker community, through the OCI, has already tentatively agreed to a multi-architecture system where both Linux and Windows will be supported as first-class ABI environments across multiple hardware platforms. This cross-system support is already available in a limited fashion for the Linux ABI on IBM's Z-System hardware, and nascent support for the arm64 architecture is available as well. It should be possible to extend this multi-ABI future to include FreeBSD.

Save the Date!

OSDI '18

13th USENIX Symposium on Operating Systems Design and Implementation

October 8–10, 2018 • Carlsbad, CA, USA

OSDI brings together professionals from academic and industrial backgrounds in what has become a premier forum for discussing the design, implementation, and implications of systems software. The OSDI Symposium emphasizes innovative research as well as quantified or insightful experiences in systems design and implementation.

The Call for Papers is now available.
Abstract registrations are due April 26, 2018.

Program Co-Chairs:
Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
and Geoff Voelker, *University of California, San Diego*

www.usenix.org/osdi18

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Save the Date!

2018 USENIX Annual Technical Conference

JULY 11–13, 2018 • BOSTON, MA, USA

USENIX ATC '18 will bring together leading systems researchers for cutting-edge systems research and unlimited opportunities to gain insight into a variety of must-know topics, including virtualization, system and network management and troubleshooting, cloud and edge computing, security, privacy, and trust, mobile and wireless, and more.

The Call for Papers is now available.
Paper submissions are due February 6, 2018.

Program Co-Chairs:
Haryadi Gunawi, *University of Chicago,* and Benjamin Reed, *Facebook*

Co-located with USENIX ATC '18

HotStorage '18: 10th USENIX Workshop on Hot Topics in Storage and File Systems
July 9–10, 2018
www.usenix.org/hotstorage18

HotCloud '18: 10th USENIX Workshop on Hot Topics in Cloud Computing
July 9, 2018
www.usenix.org/hotcloud18

HotEdge '18: USENIX Workshop on Hot Topics in Edge Computing
July 10, 2018
www.usenix.org/hotedge18

www.usenix.org/atc18

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION