Containers: from the Ground Up

Project Members:
Madison Tandberg, t98d739; Ryan Cummings, j37j718; Tyler Ross, x46r989; Ali Khaef, f15j638

Introduction

In this paper we will discuss the underlying technologies, advantages, and disadvantages of containers. To direct our discussion, we will focus on contemporary container software, using Docker as a reference implementation. We'll begin with a cursory introduction to containers, comparison to virtual machines, and related background information. Next, we'll discuss three fundamental, underlying technologies used to implement containers: control groups (cgroups), namespaces, and file systems. Finally, we'll explain and demonstrate how to implement a simple container on a Linux machine, using these three technologies.

Background

Containers isolate a process tree (i.e., a process and its children) from the rest of the system. This isolation creates the illusion of an independent system within the scope of the container. The most important aspect of this illusion is that containers do not emulate independent, hardware systems – unlike a virtual machine. As such, containers have significantly less overhead and can be run much more densely than virtual machines. Given a convenient, effective abstraction layer to manage the technical details of containerization, containers become an easy, lightweight alternative to traditional virtualization [1].

A natural first question when confronted with containers is "why not virtual machines?" There are some differences between containers and virtual machines. Most significantly, virtual machines generally provide some level of *hardware* emulation or isolation, allowing them to run fundamentally different operating systems from that of the hypervisor. In contrast, a Linux host cannot directly run a Windows container, due to their differing kernels.

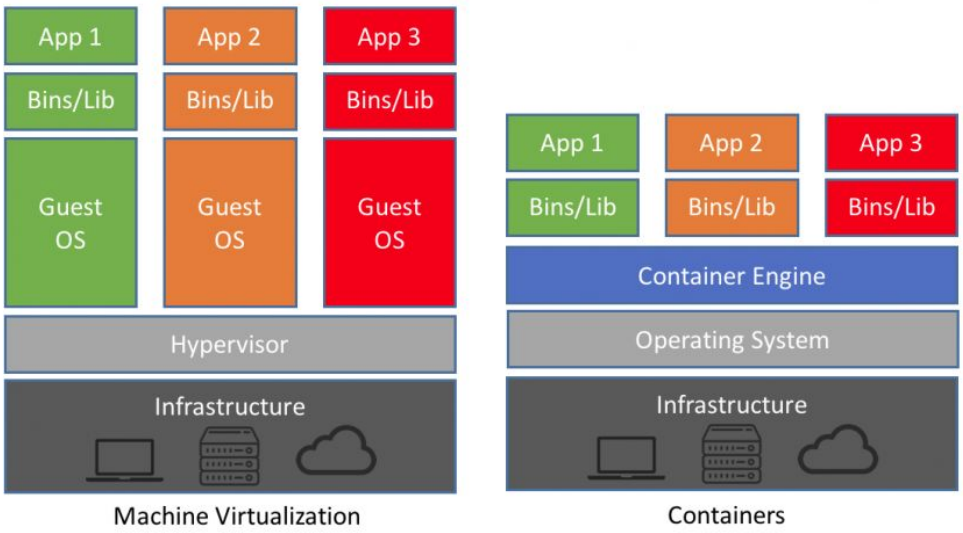

Figure 1: A general comparison of virtual machines (left) to containers (right) [2].

Let's now consider some of the pros and cons of containers, as compared to virtual machines. First, let's discuss the startup processes: when we use virtual machines, we need to boot an entire operating system. This takes a significant amount of time, when compared to the startup time of a container. Containers run on the kernel of the host operating system. An instance of a container is able to be implemented through simple system calls, and as such, they can be up and running in milliseconds. Next, containers use significantly less computer resources (such as disk space, CPU, and RAM) than virtual machines. The isolation techniques that containers employ that allow them to use less resources than virtual machines also make them more efficient [3, 4].

There are many projects that implement containers in a way that makes it easier for developers to use this virtualization. Docker, arguably the most successful implementation of containers, has seen rapid, widespread industry adoption. Throughout the course of this paper, we will use Docker to point to example implementation of the key container concepts.

Deep Dive into Container Isolation Technologies

A. Control Groups (cgroups)

The first important technical detail of a container is called a "cgroup"; this stands for control group. In fact, cgroups are a natural piece of the Linux architecture, even when containers are not -- officially -- being used. In general for operating systems, cgroups are a way to group processes based on the resources they utilize. These processes are organized in a hierarchical structure (the cgroups themselves) so that the operating system can distribute the required resources to processes in a systematic manner [5] This structure resembles a tree, where each cgroup is a leaf on the tree, and each process ID (PID) is able to be given resources in each cgroup. There can be multiple process hierarchies on a single operating system, which is how containers utilize cgroups to distribute and isolate resources to processes. While there are many different cgroups, some of the functionality they provide are: resource limiting, prioritization, accounting, and control [1].

As noted earlier, the functionality abstraction that enables cgroups (in Linux) is built directly into the Linux architecture. There is a root *cgroup* directory within the */sys/fs/* directory, that holds the specifics for each cgroup and its processes [5] . On the operating system that this class (CSCI 460, Fall 2020) is utilizing (that has had Docker installed), the root *cgroup* directory is shown below in Figure 2 [6].

```
[vagrant@vbox][/sys/fs/cgroup]$ ls -l
total 0
dr-xr-xr-x 5 root root  0 Nov 10 09:46 blkio
lrwxrwxrwx 1 root root 11 Nov  3 09:06 cpu -> cpu,cpuacct
lrwxrwxrwx 1 root root 11 Nov  3 09:06 cpuacct -> cpu,cpuacct
dr-xr-xr-x 5 root root  0 Nov 10 09:46 cpu,cpuacct
dr-xr-xr-x 3 root root  0 Nov 10 09:46 cpuset
dr-xr-xr-x 5 root root  0 Nov 10 09:46 devices
dr-xr-xr-x 3 root root  0 Nov 10 09:46 freezer
dr-xr-xr-x 3 root root  0 Nov 10 09:46 hugetlb
dr-xr-xr-x 5 root root  0 Nov 10 09:46 memory
lrwxrwxrwx 1 root root 16 Nov  3 09:06 net_cls -> net_cls,net_prio
dr-xr-xr-x 3 root root  0 Nov 10 09:46 net_cls,net_prio
lrwxrwxrwx 1 root root 16 Nov  3 09:06 net_prio -> net_cls,net_prio
dr-xr-xr-x 3 root root  0 Nov 10 09:46 perf_event
dr-xr-xr-x 5 root root  0 Nov 10 09:46 pids
dr-xr-xr-x 2 root root  0 Nov 10 09:40 rdma
dr-xr-xr-x 6 root root  0 Nov 10 09:46 systemd
dr-xr-xr-x 5 root root  0 Nov 10 09:40 unified
```

Figure 2: *cgroup* directory file structure

Containers (in the sense of a software application that allows processes to be "contained" from other processes) utilize this architecture to allocate resources. In particular, when a process is created using Docker, that process has its resources allocated and limited in this same directory that limits the other processes, as well. For example, let's look in the */sys/fs/cgroup/blkio/* directory (Figure 3):

```
[vagrant@vbox][/sys/fs/cgroup/blkio]$ ls -l
total 0
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.io_merged
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.io_merged_recursive
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.io_queued
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.io_queued_recursive
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.io_service_bytes
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.io_service_bytes_recursive
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.io_serviced
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.io_serviced_recursive
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.io_service_time
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.io_service_time_recursive
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.io_wait_time
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.io_wait_time_recursive
-rw-r--r--  1 root root 0 Nov 10 09:40 blkio.leaf_weight
-rw-r--r--  1 root root 0 Nov 10 09:40 blkio.leaf_weight_device
--w-------  1 root root 0 Nov 10 09:40 blkio.reset_stats
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.sectors
```

```
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.sectors_recursive
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.throttle.io_service_bytes
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.throttle.io_serviced
-rw-r--r--  1 root root 0 Nov 10 09:40 blkio.throttle.read_bps_device
-rw-r--r--  1 root root 0 Nov 10 09:40 blkio.throttle.read_iops_device
-rw-r--r--  1 root root 0 Nov 10 09:40 blkio.throttle.write_bps_device
-rw-r--r--  1 root root 0 Nov 10 09:40 blkio.throttle.write_iops_device
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.time
-r--r--r--  1 root root 0 Nov 10 09:40 blkio.time_recursive
-rw-r--r--  1 root root 0 Nov 10 09:40 blkio.weight
-rw-r--r--  1 root root 0 Nov 10 09:40 blkio.weight_device
-rw-r--r--  1 root root 0 Nov 10 09:40 cgroup.clone_children
-rw-r--r--  1 root root 0 Nov 10 09:40 cgroup.procs
-r--r--r--  1 root root 0 Nov 10 09:40 cgroup.sane_behavior
drwxr-xr-x  2 root root 0 Nov 10 09:47 docker
-rw-r--r--  1 root root 0 Nov 10 09:40 notify_on_release
-rw-r--r--  1 root root 0 Nov 10 09:40 release_agent
drwxr-xr-x 59 root root 0 Nov 10 09:40 system.slice
-rw-r--r--  1 root root 0 Nov 10 09:40 tasks
drwxr-xr-x  2 root root 0 Nov 10 09:40 user.slice
```

Figure 3: *blkio* directory file structure

As you can see within this directory, there is another directory called *docker*. This is where Docker stores data about the processes for all of the processes that have *blkio* privileges, using cgroups. This data can include PID's, limits of the processes, or any of the other data that is stored within the *blkio* cgroup directory for all Linux architectures. [See Figure 4 (below) to see that the files within the */sys/fs/cgroup/blkio/docker* directory are the same as within the */sys/fs/cgroup/blkio* directory.]

```
[vagrant@vbox][/sys/fs/cgroup/blkio/docker]$ ls -l
total 0
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.io_merged
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.io_merged_recursive
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.io_queued
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.io_queued_recursive
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.io_service_bytes
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.io_service_bytes_recursive
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.io_serviced
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.io_serviced_recursive
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.io_service_time
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.io_service_time_recursive
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.io_wait_time
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.io_wait_time_recursive
-rw-r--r-- 1 root root 0 Nov 10 11:38 blkio.leaf_weight
-rw-r--r-- 1 root root 0 Nov 10 11:38 blkio.leaf_weight_device
```

```
--w------- 1 root root 0 Nov 10 11:38 blkio.reset_stats
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.sectors
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.sectors_recursive
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.throttle.io_service_bytes
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.throttle.io_serviced
-rw-r--r-- 1 root root 0 Nov 10 11:38 blkio.throttle.read_bps_device
-rw-r--r-- 1 root root 0 Nov 10 11:38 blkio.throttle.read_iops_device
-rw-r--r-- 1 root root 0 Nov 10 11:38 blkio.throttle.write_bps_device
-rw-r--r-- 1 root root 0 Nov 10 11:38 blkio.throttle.write_iops_device
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.time
-r--r--r-- 1 root root 0 Nov 10 11:38 blkio.time_recursive
-rw-r--r-- 1 root root 0 Nov 10 11:38 blkio.weight
-rw-r--r-- 1 root root 0 Nov 10 11:38 blkio.weight_device
-rw-r--r-- 1 root root 0 Nov 10 11:38 cgroup.clone_children
-rw-r--r-- 1 root root 0 Nov 10 11:38 cgroup.procs
-rw-r--r-- 1 root root 0 Nov 10 11:38 notify_on_release
-rw-r--r-- 1 root root 0 Nov 10 11:38 tasks
```

Figure 4: Docker's blkio directory file structure

As you can see, Docker mounted it's *blkio* cgroup directly into the operating systems' existing *blkio* cgroup, but in a way that isolated Docker's *blkio* cgroup from the rest of the operating system that Docker is running on.

B. Namespaces

Namespaces are among the fundamental primitives used to create containers on a Linux operating system. From a high-level perspective, containers appear to function as virtual machines, separate and built on top of the host system. In this case, appearances are slightly deceiving [7]. Containers do not sit on a host system requiring a hypervisor, they are a series of isolated processes and system resources within the host system itself. Namespaces limit visibility of processes running in a container. Isolation is the key concept. Namespaces wrap global system resources in abstractions that make it appear to the processes within the namespaces that they have their own isolated instances of global resources [8]. In other words namespaces limit visibility of system resources so that processes in a namespace cannot see system resources allocated to other namespaces and processes.

Similar to scope in programming languages, namespaces provide security and convenience to the operating system. A PID that exists in one namespace cannot have a duplicate in the same namespace (i.e., if PID 23 already exists in a certain namespace, there can not be another PID 23 in that single namespace). However, the same PID can be used in *another* namespace, provided the set of namespaces the PID belongs to are mutually exclusive. All processes on the host system, not just the ones running in a container, run in namespaces. Processes can belong to multiple namespaces, and namespaces can have multiple processes. Processes that share a namespace have visibility to one another, but as the purpose of

namespaces is to provide a way to manage resource isolation, processes that don't share a namespace have no visibility to one another.

There are currently eight different namespaces used by the Linux Kernel: Cgroup, ICP, Network, Mount, PID, Time, User, and UTS. Additionally some system calls are available including clone() and unshare() [8]. A detailed discussion of each of these namespaces and system calls follows.

"Cgroup namespaces virtualize the view of a process's cgroups" [9]. The manual goes on to say "each cgroup namespace has its own set of cgroup root directories." This is the heart of the interaction between namespaces and cgroups. This namespace provides for cgroup directories preventing knowledge of host or ancestor cgroup file paths in the current process. IPC Namespaces are namespaces that isolate interprocess control resources. Likewise, Network Namespaces isolate network resources. Although networking is an important feature of containerization in industry, it is a research subject unto itself and we will not cover it in detail. "Mount namespaces provide isolation of the list of mount points seen by the processes in each namespace instance. Thus, the processes in each of the mount namespace instances will see distinct single-directory hierarchies" [10]. PID namespaces are also crucial. They isolate process IDs, meaning that if a PID exists in one namespace the same PID, but a different process, can exist in a different PID namespace. User Namespaces "isolate security-related identifiers and attributes, in particular, user IDs and group IDs" [11]. This is one of the important aspects of container security. UTS Namespaces isolate host and NIS (Network Information Systems) domain names. Lastly, Time Namespaces isolate date and time information. This namespace was added to the mainline Linux Kernel earlier this year. Although Namespaces have been in use for more than a decade, it is still an area that is under active development. The common thread with each of these namespace types is that they limit visibility system resources and help provide the illusion that all of the system resources are contained in the respective namespaces.

In addition to the eight namespaces, There are three system calls associated with namespaces. The first is *unshare()*. The *unshare()* syscall "unshares the indicated namespaces from the parent process and then executes the specified program. If a program is not given, then ``*${SHELL}*" is run (default: */bin/sh*)" [12]. In other words, it isolates a namespace from the host and runs either a specified program or a shell. *Clone()* is also used in conjunction with namespaces. *Clone()* creates a new process similar to *fork()* but with the optional ability to create a new namespace for the new process [13]. *Clone()* and its effects on namespaces is customizable through a series of flags. Finally, the *setns()* syscall "allows the calling process to join an existing namespace" [8]. Through these three calls, processes and namespaces can be created and managed providing processes isolation and control over visibility.

C. File Systems

As in more traditional computing paradigms, filesystems are of critical importance to the implementation of contemporary containers. The use of "union" filesystems, in-turn utilizing Copy-on-Write (COW) strategies, provides for more efficient distribution of container images,

more efficient use of storage resources and rapid deployment of new containers. We'll examine how Docker utilizes such filesystems to achieve each of these features, but we must first consider the underlying concepts: Copy-On-Write and "union" filesystems.

Copy-On-Write (COW) refers generally to a paradigm of resource management in which "copies" of a resource (file, storage block, memory pages, etc.) reference a single canonical copy *until* a modification is requested [14]. At which time the modification is either recorded separately (more correctly called Redirect-On-Write, though COW is often used to refer to either strategy) or the original resource is copied to a new storage area before modification [14].

Copy-On-Write (subsuming Redirect-On-Write for the purposes of this discussion), is used in many places including the Linux kernel's "same-page" memory page merging, the Microsoft SQL Server database and the ZFS file system – the "billion dollar filesystem" [14].

Of interest to the topic of containerization is the fact that COW directory trees (depending on implementation) can be "copied" entirely with negligible operations and without unnecessarily duplicating data [14]. Given this feature, a new container – whose base image may occupy several gigabytes on disk – can be created and started in trivial time and with minimal resource overhead.

Contemporary containerization platforms, like Docker, take advantage of Copy-On-Write in the form of "union" filesystems. Many union filesystems exist, including kernel-native implementations on Plan 9, FreeBSD and Linux. Windows has also recently introduced (to limited licenses/versions) the Unified Write Filter (UWF), which we'll neglect to discuss due to limited adoption. The most general idea of a union filesystem is that it "unions" (combines) multiple filesystems into a unified namespace. The exact semantics vary wildly with implementation, serving a wide range of use cases. Staying in the context of containerization on Linux, and particularly contemporary Docker, we'll discuss the Advanced Multi-Layered Unification Filesystem (AUFS; formerly AnotherUnionFS – a complete rewrite of the earlier UnionFS) and OverlayFS.

AUFS and OverlayFS implement a union filesystem using the idea of "layers", or "overlays". For this reason, Miklos Szeredi, the original developer of OverlayFS, argues "overlay filesystem" is a more appropriate, accurate classification for these filesystems. Conceptually, layers are merged (overlaid) with some ordinality such that later occurrences of a file mask earlier ones [15]. Thus, only the last instance of any file – as specified by path relative to the merged namespaces – is presented in the final, "merged" filesystem.

Linux implementations of union filesystems (in the more general sense than overlay filesystems is intended to encompass) date back to 1993 (the Inheriting File System (IFS) by Werner Almsberger). AUFS follows from UnionFS which was developed at Stony Brook University in 2003 [16]. AUFS began development as a fork of UnionFS, but was completely rewritten in 2006. AUFS is under active development and fully supports the latest mainline 5.x Linux kernel – NB: AUFS is *not* part of the mainline kernel! OverlayFS was developed independently, rather than as a fork, out of a perceived need for a union filesystem within the mainline Linux kernel. Miklos Szeredi submitted the first OverlayFS RFC patch to the Linux

kernel mailing list in 2010. OverlayFS was merged into version 3.18 of the mainline Linux kernel in 2014.

Implicit in the following discussion, given the context of containers, is a writable layer at the top of the overlay hierarchy – the underlying layers of which are read-only. Neither AUFS nor OverlayFS requires the presence of a writable layer, though most use cases will have one (and AUFS even supports multiple). This writable layer is visible in the following diagram, which also highlights some overlay filesystem concepts (e.g. “whiteout” files) which are discussed in following sections.

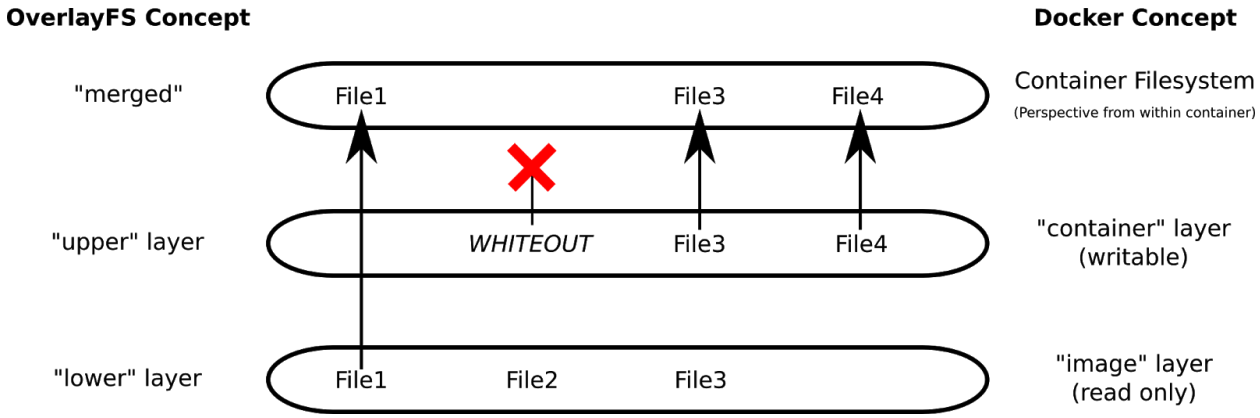

Figure: Example of two layers, a writable “upper” and read-only “lower”, unioned by OverlayFS in the context of a Docker container. The uppermost (“merged”) layer depicts the merged namespace presented – in this case, to a Docker container – by OverlayFS

AUFS, the Advanced Multi-Layered Unification Filesystem (AnotherUnionFS prior to version 2), improves on a number of problems in the older UnionFS – such as significant duplication of data structures – and provides many runtime configuration options (Valerie Aurora). It was ultimately rejected from the mainline Linux kernel due to its large, opaque codebase (20,000 "dense, unreadable, uncommented" [16] lines, compared to – at the time – 60,000 lines for the entire Linux VFS). Despite this shortcoming, AUFS improved significantly on prior work and saw practical application in several projects. Most notably for our discussion, AUFS was the default overlay filesystem used by Docker prior to the introduction of OverlayFS.

At a high level, AUFS is implemented as a middleware between the VFS and underlying, "unioned" file systems. The merged file system namespace exposed by AUFS is accessible via the standard VFS interface with operations interpreted and directed by the middleware. AUFS uses a number of intermediate data structures which generally take the form of a top-level filesystem object (such as a dentry) containing an array of references to the lower filesystems of the union. Whiteouts (see “File Operation Behaviors”) are implemented by hard-linking to a single whiteout inode in the respective directory [16].

OverlayFS differs most significantly from AUFS in that it merges precisely two filesystems (more generally, two directory trees – the underlying file systems need not be

distinct): an "upper" and a "lower" (which may itself be an OverlayFS) [17]. It is notable that OverlayFS has evolved into "overlay" in newer kernels – which relaxes some constraints (such as the single "lower" layer). These more recent advancements, and their nuances, are beyond the scope of this discussion. Suffice it to say, roughly equivalent constructions are possible with either "generation" of OverlayFS.

OverlayFS operates in a similar, intermediary fashion to many other union/overlay filesystems, but with some notable distinctions. Foremost, OverlayFS uses a combination of character devices (with device number 0/0) and the "trusted" class of extended attributes to indicate whiteouts – avoiding the whiteout-file (discussed later) and hard-linking problems which plagued earlier union filesystems [17]. Newer versions of OverlayFS also make more efficient use of inodes and other kernel data structures – though older versions still improved significantly on prior work such as AUFS.

File operation behavior overlaps considerably between AUFS and OverlayFS, as they mostly adhere to the same general paradigm. Following is breakdown of union filesystems' behavior – with some consideration for OverlayFS and AUFS specifically – under each of several common filesystem operations.

When a file read occurs, the layers are scanned from the "top" (generally the writable layer) down. The first occurrence of the file in this traversal is read from the underlying storage of that respective layer. It is notable that the file is read directly from the layer where it is found. Thus, there is very little read performance overhead.

When a file modification (in-place/partial write, append, etc.) occurs, if a copy does not exist in the top-level, writable layer, a "copy-up" occurs. The layer hierarchy is traversed to find the first occurrence of the file which is then copied to the writable layer and modified. In the case of a pure write (as opposed to a modification/append), a file – which implicitly supersedes any lower-layer file – is created in the writable layer.

When a file delete occurs, a "whiteout" file is created in the writable layer. This special file supersedes lower layers and effectively "hides" the file from users of the filesystem. This is necessary to workaround the generally read-only lower layers, and avoid traversing the layers in search of a deleted file. The implementation of this whiteout varies significantly (see prior discussion of AUFS and OverlayFS implementation details).

Renaming of files and directories is where overlay file systems depart from intuition and convention. As, by construction, the lower layers are immutable (excepting some novel (often inadvisable) use cases), renaming a file or folder which exists in a lower layer is a non-trivial operation. In general, rename operations on AUFS and OverlayFS will produce an "EXDEV" error (indicating a file cannot be renamed across filesystem boundaries) [17]. Applications are expected to handle this error by implementing a "copy and unlink" strategy.

In the case of such a "copy and unlink" operation, a copy-up occurs – creating file(s)/folder(s) of the desired name in the writable layer – and the original path(s) are "unlinked" (deleted) – creating a whiteout which masks paths of the original name at lower layers.

D. Overlay Filesystems in Docker [18]

Docker makes extensive use of overlay filesystems, to great effect. Foremost, the use of layers allows Docker to optimize the distribution of container images. Consider a system running three containers which share some underlying layer – for example, a particular version of Ubuntu. Such shared layers need only be downloaded to the system once, saving bandwidth.

Furthermore, containers using such base layers can be distributed and stored as only their differences to the underlying layer. This again saves on bandwidth and reduces the end-user's storage needs – as the underlying layer(s) are stored only once. This is in sharp contrast to more traditional virtualized "appliances" which are generally distributed as entire virtual machine disk images on the order of gigabytes in size. When using appliance virtual machines, such deduplication-savings are not possible (excepting expensive, complex storage infrastructure which can perform block- and/or file-level deduplication).

Finally, because of their readonly use, integrity checking (i.e. hashing of an archived layer) and modern, robust distribution norms (e.g. signed archives/hashes) can be used to detect and prevent the use of corrupt, or potentially malicious, layers.

Demonstration

Now that we understand the main technical components of a container, we want to provide some intuition on how they fit together. We will do this through a small demonstration of containerizing a process, in the same (or, at least similar) way that Docker containerizes processes.

The first step in containerizing a process is to prepare a filesystem for the new environment. For this demonstration, we'll be using a premade Alpine container image provided by Docker.

```
[root@CSCI460-DevBox][~]# #Download latest alpine docker image:
[root@CSCI460-DevBox][~]# docker pull alpine
[root@CSCI460-DevBox][~]# #Export the alpine container image to a .tar file:
[root@CSCI460-DevBox][~]# docker create --name alpine_temp alpine
[root@CSCI460-DevBox][~]# docker export alpine_temp -o alpine.tar
[root@CSCI460-DevBox][~]# docker rm alpine_temp
[root@CSCI460-DevBox][~]#
[root@CSCI460-DevBox][~]# #Setup a trivial (for demo concision) "root filesystem"
[root@CSCI460-DevBox][~]# #for the container
[root@CSCI460-DevBox][~]# mkdir containerroot
[root@CSCI460-DevBox][~]#
[root@CSCI460-DevBox][~]# #Load the container's filesystem with the alpine image
[root@CSCI460-DevBox][~]# tar xf alpine.tar --ignore-command-error -C containerroot/
[root@CSCI460-DevBox][~]#
[root@CSCI460-DevBox][~]# #Touch an arbitrary file, to provide context later
[root@CSCI460-DevBox][~]# touch containerroot/IN_CONTAINER_FILESYSTEM
[root@CSCI460-DevBox][~]#
```

The next step is to create the container by executing a process in a new, isolated namespace using the unshare syscall.

```
[root@CSCI460-DevBox][~]# # Start the "ash" shell inside a chroot into the
[root@CSCI460-DevBox][~]# "containerroot" filesystem, and “unshare” the
[root@CSCI460-DevBox][~]# # resulting process from the parent namespace
[root@CSCI460-DevBox][~]# unshare --mount --uts --ipc --net --pid --fork --user
--map-root-user chroot $PWD/containerroot ash
/ #
/ # ls
IN_CONTAINER_FILESYSTEM  lib                      root                     tmp
bin                      media                    run                      usr
dev                      mnt                      sbin                     var
etc                      opt                      srv
home                     proc                     sys
/ # ps -a
PID   USER     TIME  COMMAND
    1 root      0:00 ash
    6 root      0:00 ps -a
/ #
```

At this point, we have an instance of the “ash” shell (located in containerroot/bin/ash), which is not standardly available on the Ubuntu host system. We also note that ps reports ash, the process we containerized, has PID 1, thanks to the illusion created by namespaces. If we use pidof from outside the container, we observe the actual, global PID to be 30185.

```
[root@CSCI460-DevBox][~]# pidof ash
30185
[root@CSCI460-DevBox][~]#
```

Before we consider resource management using control groups, we quickly reestablish some key mountpoints (/proc, /sys, /tmp) to finish isolating the process in the scope of the new namespace.

```
/ # # Cleanup the kernel mountpoints, to finish isolating the container
/ # mount -t proc none /proc
/ # mount -t sysfs none /sys
/ # mount -t tmpfs none /tmp
/ #
```

Finally, to demonstrate management using kernel control groups, we’ll create a new freezer cgroup (used to suspend a group of processes), assign our container to it, and “freeze”

(suspend) it while observing a running process. We begin, on the host side, by creating the cgroup and assigning the container to it.

```
[root@CSCI460-DevBox][~]# # Make a freezer cgroup
[root@CSCI460-DevBox][~]# mkdir /sys/fs/cgroup/freezer/handmadecontainer
[root@CSCI460-DevBox][~]#
[root@CSCI460-DevBox][~]# # Put the container (running ash) in the new cgroup
[root@CSCI460-DevBox][~]# echo $(pidof ash) >>
/sys/fs/cgroup/freezer/handmadecontainer/cgroup.procs
[root@CSCI460-DevBox][~]#
```

We then start a long-running, easily observable process in the container (in this case, a looping timestamp). While observing this process, we "freeze" (suspend) the cgroup – noting the output has ceased – and "thaw" (resume) the cgroup – now noting that the output has resumed. For clarity of the demonstration, we've included accompanying timestamps to the freeze and thaw operations on the host side (figure #).

```
/ # # Start a long-running, observable child process
/ # while :; do date "+%H:%M:%S"; sleep 1; done
20:34:16
20:34:17
20:34:27
20:34:28
20:34:29
^C
/ #
```

Figure: Container perspective during cgroup-based "freeze," and subsequent "thaw."

```
[root@CSCI460-DevBox][~]# #Show timestamp, to correlate container freeze/unfreeze
[root@CSCI460-DevBox][~]# date "+%H:%M:%S"
13:34:17
[root@CSCI460-DevBox][~]# #Freeze the container using the "handmadecontainer" cgroup
[root@CSCI460-DevBox][~]# echo "FROZEN" >
/sys/fs/cgroup/freezer/handmadecontainer/freezer.state
[root@CSCI460-DevBox][~]#
[root@CSCI460-DevBox][~]# #Show timestamp, to correlate container freeze/unfreeze
[root@CSCI460-DevBox][~]# date "+%H:%M:%S"
13:34:27
[root@CSCI460-DevBox][~]# #UNfreeze the container
[root@CSCI460-DevBox][~]# echo "THAWED" >
/sys/fs/cgroup/freezer/handmadecontainer/freezer.state
[root@CSCI460-DevBox][~]#
```

Figure: Host perspective while "freezing", and then "thawing", a cgroup.

Discussion

One of the main positive attributes of Containers, in the software world, is the fact that software is able to be tested and run in an efficient manner, using containers. This is in contrast to the past use of virtual machines, which as we have already seen, require exponentially more resources than Containers. This particular positive effect of containers is mostly felt in the DevOps world. As software developers are able to run their processes in an isolated manner, this will lead to the future adoption of Containers in additional technical (and potentially non-technical) fields. For example, we predict that the health care system may adopt the use of Containers to isolate processes that manipulate patient data. By running these processes in an isolated container, there may be added security to the patient. This is just one non-technical area that Containers could have a lasting impact on. However, for these non-technical areas to be able to utilize the benefits of containers, there will need to be more development in making containers easier for people to use. While this is one of Docker's main selling points, there will likely be a barrier to adopting this technology to non-technical people until there is a graphical user interface that is easy to use. Through the widespread adoption of this technology, containers will be able to influence new sectors of technology. Additionally, we think that containers could be a direction of more robust isolation in mobile development in the future.

Conclusion and Lessons Learned

In this project, we presented an overview of Containers, with respect to their difference with other isolation technology used today, dove into technical details about the three main technologies that enable Containers in a Linux setting, and implemented a simple container, using Docker as a point of reference. Throughout this process, it became evident that Containers are extremely powerful. So much so, that Linux developers purposefully modified the Linux Architecture to better allow for containerization in the mid 2000's, and are still today updating the system to make this more convenient. Finally, it also became evident that working with these kernel interfaces is hard! This point explains why Docker, and other Containerization software, is widely used in industry, and why we predict their use will become even more widespread in the years to come. These softwares take the tricky commands we implemented in the Demonstration section, and provide powerful, convenient abstractions that do it automatically for the user. This lets the creation of containers be more systematic (i.e., less error prone), faster, and easier to navigate. We also learned more about the role of the kernel in distributing system resources and isolating processes.

References

[1] Stallings, W. (2018). Operating Systems: Internals and Design Principles (9th ed.). Pearson. 978-0134670959

[2] Chamberlain, D. (2018, March 16). Containers vs. Virtual Machines (VMs): What's the Difference? Retrieved November 14, 2020, from https://blog.netapp.com/blogs/containers-vs-vms/

[3] Nagy, G. (2015, May 19). Operating System Containers vs. Application Containers. Retrieved November 14, 2020, from https://blog.risingstack.com/operating-system-containers-vs-application-containers/

[4] Pezet, D. (2020, September 4). Virtual Machines vs Containers - Which is right for you? Retrieved November 14, 2020, from https://www.youtube.com/watch?v=XCWWPpfdbsM

[5] cgroups(7), Linux Programmer's Manual

[6] Gulati, S. (2019, January 3). How Docker uses cgroups to set resource limits? shekhargulati.com. Retrieved November 8, 2020, from https://shekhargulati.com/2019/01/03/how-docker-uses-cgroups-to-set-resource-limits/

[7] Petazzoni, J. (2015, August 19). Anatomy of a Container: Namespaces, cgroups & Some Filesystem Magic - LinuxCon. Slideshare. https://fr.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon

[8] namespaces(7), The Linux Programmer's Manual

[9] cgroup_namespaces(7), The Linux Programmer's Manual

[10] mount_namespaces(7), The Linux Programmer's Manual

[11] user_namespaces(7), The Linux Programmer's Manual

[12] unshare(1), The Linux Programmer's Manual

[13] clone(2), The Linux Programmer's Manual

[14] Jude, A., & Lucas, M. W. (2015). FreeBSD Mastery: ZFS. Tilted Windmill Press. 9781642350005

[15] Aurora, Valerie (2009-03-18). "Unioning file systems: Architecture, features, and design choices". LWN.net. Retrieved 2020-11-07.

[16] Aurora, Valerie (2009-04-07). "Unioning file systems: Implementations, part 2". LWN.net. Retrieved 2020-11-07.

[17] filesystems/overlayfs, The Linux Kernel Documentation

[18] Manage data in Docker. Retrieved November 7, 2020, from https://docs.docker.com/storage/