Getting your hands dirty with Containers

Indian Institute of Technology - Bombay

By: Prashanth
prashanth@cse.iitb.ac.in

Guide: Prof. Purushottam Kulkarni

Contents

List of Figures

1. Introduction

The subject for discussion in context to this manual is **Containers**. But before we go there, we need to understand the initial motivation which created the need for such a system. To understand the initial motivation for the same, it in-turn requires us to understanding the basics of an Operating System.

Operating system (OS) is itself a branch of study that would take a very long for one to understand in detail. The study of OS is beyond the scope of this manual, but the manual provides a brief introduction to understand the setup of an OS with context with understanding containers. The current chapter discusses a quick introduction to Operating Systems followed by their drawbacks with respect to process control and isolation. The chapter concludes with the need to design a new system to overcome the discussed drawbacks.

The flow of the manual there after is as follows. After setting up the problem to be addressed in this chapter, it mentions about the initial designs proposed to the system, which is followed by the drawbacks in the initial proposals. Then it talks about the components that make up a container following which it discusses how these components interact with one other to provide the required system. Finally it looks at the different types of containers provided by various vendors and how they differ from one other. The manual also provides some hands-on with demonstrations followed by exercises. The manual concludes with a summary on the existing system and how things could work out in the future.

The purpose of this manual is to provide an overview of the existing container system. It is strongly recommended to skim cover to cover if you are a beginner to this arena. However, if you feel confident enough and only want to look at the specifics of a new few sections, feel free to skip to the required sections.

1.1 A quick overview of OS design

The earliest systems were mainframe [2] computers. These were large systems which performed bulk processing of data and lacked any form of a controlling entity. Programmers had to design applications according to the internal structure of the hardware. This lead to tedious programming and redundant logic in programming the interaction between applications and hardware. Hence there was need for an intermediate software in-place to make programming application easier, so as to make the programmers think more about the application logic than to design its interaction with the required hardware. This was one of the primary motivation for the development of an operating system.

There are many definitions for an operating system. But the most common definition is given by Silberschatz [3] who defines as,

"An operating system is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware"

To put it more in terms of design space, an operating system revolves around the idea of a system level software that manages requests of user programs to utilize the hardware resources available in a system, in an administered manner. With this in mind, operating systems were originally designed to manage and control the different resources present on a system. This design works well as long as the resource requirements for a given process are met by the system instantaneously.

1.2 Drawbacks in OS with respect to processes

Operating system was initially designed with a typical use case in mind, although it tries to generalize the design as much as possible overtime. There are several drawbacks of the current OS design and a few of them depends more on the actual use case of the system. The above described system provides an very effective design in most cases, but there are couple major drawbacks of the above existing design with respect to processes which have been discussed below.

1.2.1 Resource Control

Consider situations where the system isn't able to allocate the requested resources by a process at a given instant. Hence it defers the request and allocates these requested resources when these resources are released by other processes which were using it and allocates them at a later instant to the requesting process. This causes execution of a process to be delayed and this delay may not be always acceptable. There may even be situations where a malicious process executing on a system blocks all or majority of the resources on a system and there by not allowing other processes to execute.

To demonstrate the support of the above argument, consider a server running an apache demon attached to a PHP backend system to handle incoming HTTP requests. Now there might be SMTP mail server running on the same system which provides a mailing service. Now consider the situation where the apache server receives a flash crowd due to which most of the system resources are being utilized to handle the enormous traffic which ultimately crashes the server and there by crashing both the apache and mail server. In this case, the mail server crashed without it being the cause of the instability in the system. The above scenario illustrates the problem of resource control in a system where processes executing in a system have no control over the resource allocation in a system.

1.2.2 Resource Isolation

Now that we have established the problem of resource control, let's have look at the problem of resource isolation. Consider the case where two different processes managed by two different users. Due to the inherit simplistic design of the operating system, these processes would most likely be able to view the resources utilized by one another and might also be able to manipulate them (depending on the resource

isolation provided by the underlying OS). Thinking about it logically, these processes executed by different users ought to hide process specifics from one another. Hence the inherit design of an operating system provides a very week form of isolation.

1.3 Requirements of a New System

To overcome the two problems discussed above, there was a need to come up with new mechanisms by which processes have some control and isolate resources present in a system. This would ease the task of enforcing policies to develop robust applications with specific resource constraints while providing isolation between other processes executing on the same system.

While designing the system it was taken into account to allow the following requirements to be enabled for a per process or a group of processes basis, and here after we would refer to either of them as process groups. The following were the requirements of process groups in a system. They must be able specify the following to the OS, and the OS in return must be able to enforce them.

1. Account different resources used by process groups
2. Specify resource constrains
3. Constrains maybe be resource (CPU/ IO/ Memory etc.) specific
4. Processes running in the same process group must be kept in isolation from other processes on the same system and vice-versa.
5. Resources utilized by these process groups must be kept hidden from other processes on the system.
6. Each process group must be able to have its own group of users who have access to the processes.
7. Users executing on a process group in a shared machine must only be able to access and view processes and resources that have been assigned to them.
8. Processes-ID, User-ID, resource-ID in a process group must be reusable in a different process groups present on the same physical machine.
9. Should provide minimum overhead on enforcing the above mechanisms

Keeping the above mentioned requirements in mind, several vendors focused on implementing a solution that would address the above problem by satisfying the requirements. The following chapter discusses the initial attempts to the solution.

2. History

This chapter mainly deals with the initial solutions proposed to the above required system. To understand the same one needs to know the understand the basic definition of a container. In simple terms container can be defined as follows,

"Container is a process or set of processes grouped together along with its dependent resources into a single logical OS entity. It enables multiple isolated user-space instances on a host machine."

Container was the solution proposed to the above problem. It is to be remembered that a container is built as an extension to the existing operating system and not as an independent system. Containers are also referred to as OS-Virtualization as it virtualizes the system at an OS level. An in-depth discussion on the details of a container would be done in the later chapters. Right now we just trying to setup a basic outline of a container to set up the discussion for the current chapter for discussing the initial works.

2.1 The Myth

Often people have this misunderstanding that container is a recent technology and has only been around for while. This confusion is because containers are often treated as light weight alternatives to Virtual Machines [4], and the use of containers in the context of virtualization has been popularized only since about 2013. But all this assumption is a myth. The idea of containers dates back to the year 1999, where Resource Containers [5] were proposed to provide fine grained control over resources in an operating system which had nothing to do with virtualization.

Having said all this, it has to be noted that containers are often dealth with from a virtualization viewpoint and throughout this manual we would be focusing our discussion from a virtualization viewpoint.

2.2 Recognition of its use in virtualization

In the same year as resource containers were introduced, Alexander Tormasov proposed a new direction towards virtualized containers. This lead to the building containers which mentioned virtualized containers as

"A set of processes with namespace isolation, file system to share code/ram and isolation in resources "

He lead his team to building the same with adding bits and features to Linux kernel [6] version 2.2. In the year 2002, they released their initial version of virtualized containers known as SWsoft (now known as Virtuozzo, owned by Parallels). It was initially launched for Linux and later in 2004 released the same for windows. In the year 2005, SWsoft created the OpenVZ [7] project to release the core of Virtuozzo under GNU GPL.

While all this was going on, there was another parallel project started by Jacques Gélinas called the Linux V-Server [8] in the year 2003 which also aimed to develop virtualized containers. Similarly over the years several new vendors and projects were started at building more and more implementations to virtualized containers.

To get an idea of how these projects/vendors implemented this process grouping, lets take the example of V-server [8]. V-server implemented a token bucket filter to schedule process groups, a hierarchical token bucket to schedule network packets, memory control by using a watchdog demon to kill process groups that overuse, hides processes of other groups by making changes to the kernel data structures and using them same while displaying processes, chroot to bar process groups to a specific directory and many more such techniques. Similarly, many of these involved building there own vendor/project specific implementations which were patched to the kernel. Most of these implementations even added an significant amount of overhead in doing the same.

2.3 New features in Kernel

In 2006, Eric W. Biederman proposed "Multiple Instances of the Global Linux namespaces" to support process group isolation in Linux symposium. Initially 10 different namespaces were proposed, of which the first one already existed as mount namespace and was introduced in 2002. Over the years the namespaces proposed have been implemented one by one and added into the Linux kernel. Right now 6 of the 10 namespaces proposed, are present in the Linux kernel. More are excepted to be implemented in the future kernel versions.

In 2007, Google presented a new generic method to solve one of the initial problem of resource control with the cgroups [9] project. This allowed resources to be managed based on process groups in terms of accounting and control. In 2008, this was merged with the mainline kernel. More about cgroups and namespaces are mentioned in the coming chapters.

2.4 Standardization

As mentioned earlier, between the years 2007-11 many companies started providing their own patches to support their container management systems and had their own implementation. The virtualization world started realizing the true potential of containers but also noticed in a flaw in the design. As in the case of hypervisors which exists today in the mainline kernel to run virtual machines, multiple vendors provide

multiple patches to the kernel source to support them. They have their own implementations which are redundant code and each vendor's implementation may not be the most efficient one in all dimensions.

In 2011, all the container vendors/projects came together and agreed upon a common unification to avoid the redundancy in kernel patches which existed in hypervisors. They agreed to take support from cgroups and namespaces and provide their APIs on top of them to manage their containers. In 2013, first Linux kernel supporting this was released. Today most container technologies make use of these two features and build upon them to provide their own version of a container manager.

The next chapter discusses about the components that make up today's container and have been discussed in detail.

3. Nuts and Bolts of a Container

Container technologies support different existing operating systems like Linux, Windows, OS-X etc. But for the purposes of our discussion we try to focus on the Linux kernel as it the most widely used OS for deploying containers. Almost every system built is typically an amalgamation of several smaller components. Similarly, there are three crucial elements which constitute a Linux container, which are - cgroups, namespaces and disk images. The current chapter introduces to them, before moving on to actual design and contents of the present day containers.

3.1 Control Groups

The solution to process control and accounting was proposed by Google in 2007 which was originally called Generic Process Containers [9] and was later renamed to Control Groups (cgroups), to avoid confusion with the term Containers. A cgroup/subsystem refers to a resource controller for a certain type of CPU resource. Eg- Memory cgroup, Network cgroup etc. It is derives ideas and extends the process-tracking design used for cpusets system present in the Linux kernel. There are 12 different cgroups/subsystems, one for each resource type classified.

For the purpose of our discussion we will stick to subsystem as the terminology referring to individual resource control and cgroup to refer a cgroup node in hierarchy. The Linux kernel by default enables most subsystems hence even if you think you are using a system without using subsystems, technically all your processes are attached to the default (root) cgroup of each of the subsystem. Default (root) cgroups are described in the coming sections. The overheads introduced by cgroups are negligible.

3.1.1 Changes to Kernel

The changes to the kernel which are required to understand the system described here. These are data structures introduced to incorporate cgroups into the kernel. Data structures are shown in green characters in Fig:3.1.

- `container_subsys` is single resource controller and it provides callbacks in the event of a process group creation/modification/destruction.
- `container_subsys_state` represents the a node in a resource controller hierarchy on which constrains can be specified.

- `css_group` holds one `container_subsys_state` pointer for each registered subsystem. A `css_group` pointer is added to the `task_struct`.

Set of tasks with same process group membership are binded to the same `css_group`. The major changes to the kernel code to incorporate cgroups have been listed,

- Hook at the start of `fork()` to add reference count to `css_group`
- Hook at the end of `fork()` to invoke subsystem callbacks
- Hook in `exit()` to release reference count and call exit callbacks

3.1.2 Hierarchies

It is possible to have 12 different hierarchies, one for each subsystem, or a single hierarchy with all 12 subsystem attached, or any other combination in between. Every hierarchy has a parent cgroup(node). Each cgroup in a hierarchy derives a portion of resources from its parent node. And resources used by the child is also charged for the parent. Each process in the system has to be attached to a node in each hierarchy, this node may even be the default node. It starts from the root (default) cgroup in each hierarchy. The root node typically has no constrains specified and is bounded by the actual resources available in the system. Although a node could be potentially attached to multiple subsystems at once, our discussion deals with a node being attached to a single system at once, as multiple subsystems attached to a single hierarchy is uncommon in real world use currently.

Fig:3.1 illustrates a minimalistic outline of a cgroups hierarchy with 3 subsystems mounted in a system onto their own hierarchies. The three subsystem mounted are - memory, cpuset and blkio and are mounted at `/sys/fs/cgroups/`. Memory root cgroup of 8GB is divided into two cgroups M1 and M2 of 4GB each. cpuset root cgroup of 4CPUs is divided into two cgroups C1 and C2 of 3CPUs and 1CPU respectively. blkio root cgroup of is divided into two cgroups B1 and B2 of 1000 and 500 as relative weights respectively. Every process which attaches itself to the same set of subsystems are referred by a single `css_group` which in turn points to the cgroup node the process is attached to. In the Fig, processes 1,2 attach itself to the blue `css_group` and 3,4,5 to the red one. The `css_group` in turn has pointers to `container_subsys_state` that is one for each cgroup. Notice how the blue `css_group` points to the root cpuset cgroup there by assigning it all the CPUs in the system which is also a valid and default value to attach processes.

3.1.3 User Space APIs

Control groups are managed using pseudo file system. A root cgroups file system directory contains a list of directories which refer to different hierarchies present in the system. A hierarchy refers to a subsystems/ or a group of subsystems attached to a single hierarchy. `mount()` is used to attach a new hierarchy to a system, and it should indicate the subsystems to be bound to an hierarchy. Each hierarchy will have its own hierarchy of directories (nodes).

Every cgroup(node) in hierarchy contains files which specify control elements of the cgroup in the hierarchy. Only unused subsystems maybe be mounted. Initially all processes are in the root cgroup of all hierarchies mounted on the system. Creating a new directory `mkdir()` in cgroup directory creates the new child.

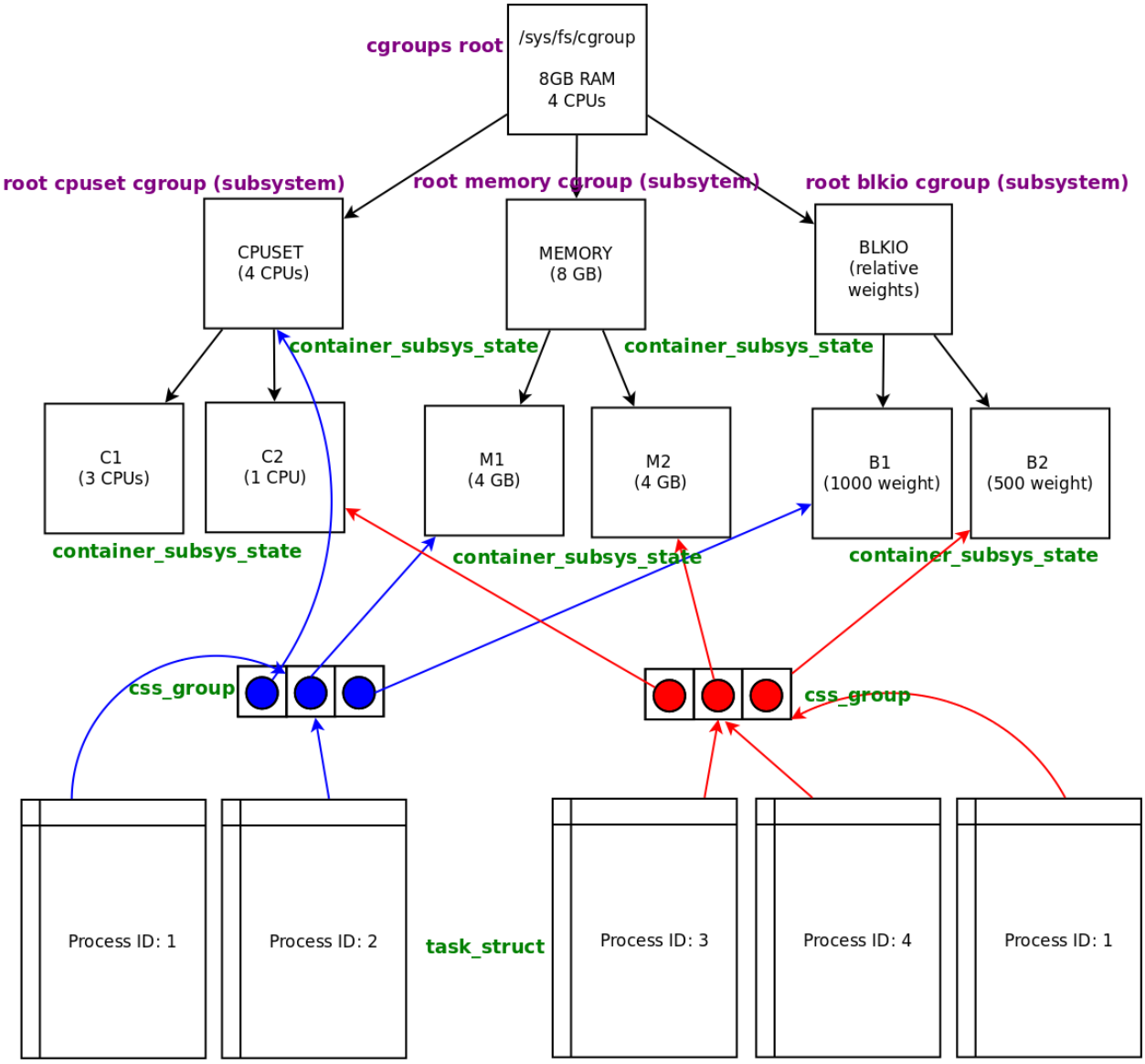

Figure 3.1: Control groups illustration using 3 controllers

Can be nested in any desired manner. Special control file – cgroup.procs is used to track all processes in the cgroup. `rmdir()` can be used to remove cgroup provided no processes are attached to it.

3.1.4 Different Subsystems Available

This section gives a brief intuition to the different subsystems available. Although an in-depth discussion of each on the same is beyond the scope of our discussion as each subsystem itself contains content enough to be discussed over a large chapter.

1. Memory

Memory subsystem use a common data structure and support library for tracking usage and imposing limits using the "resource counter". Resource controller is an existing Linux implementation for tracking resource usage. Memory cgroup subsystem allocates three `res_counters`. The three of them are described below.

i. Accounting: Accounting memory for each process group. Keeps track of pages used by each group. Pages can be classified into four types.

- Anonymous: Stack, heap etc.
- Active: Recently used pages
- Inactive: Pages read for eviction
- File: Reads/Writes/mmap from block devices

ii. Limits: Limits can be set on each cgroups. Limits are of two types - soft and hard. Soft limit is the limit up to which the system guarantees availability. Hard limit is the limit up to which the system tries to accommodate, but cannot guaranty this if system is under memory pressure. Limits can be set in terms of byte for,

- Physical memory
- Kernel memory
- Total memory (Physical + Swap)

iii. OOM: Out Of Memory killers are used to kill processes or trigger any other such event on reaching hard limit by a process group.

2. HugeTLB

HugeTLB restricts the huge pages usable by a process group and also enforces a controller limit during page fault. SIGBUS typically occurs when you try to access a file beyond your mapped region. Since HugeTLB doesn't support page reclaim, when a process group tries to overdo its limit, it send a SIGBUS to the process. HugeTLB subsystem internally it also makes use of one `res_counters`, similar to memory subsystem.

3. CPU

CPU subsystem manages how the scheduler shares CPU time among different processes groups. Access to resources can be scheduled using two schedulers - Completely Fair Scheduler (CFS) & Real-Time scheduler (RT). The stats related to a execution time and throttled time for a process group can be viewed by running `cat cpu.stat`. It allows us to specify the following parameters respect to a cgroup,

- Period: Specifies a period of time in microseconds, for how regularly a cgroupś access to CPU resources should be reallocated. Maximum is 100000 microseconds.

- Quota: Total amount of time in microseconds, for which all processes in cgroup can run in a given time period.
- Shares: Shares represent relative shares with respect to other cgroups in the system. Default value of CPU shares is 1024.

Quota and period parameters operate on a CPU basis. To allow a process to fully utilize four CPUs, for example, set `cpu.cfs_quota_us` to 400000 and `cpu.cfs_period_us` to 100000. Say you wish to enforce a 2:1 priority to two cgroup nodes running on a system. You leave the former cgroup at 1024 shares and set the latter one at 512 shares. This would give you the desired ratio.

4. CPUset

The CPUset subsystem pins individual logical CPUs and memory nodes to cgroups. Its support existed in the kernel even before introduction of cgroups. These are mandatory parameters, which indicates their values to be set before moving a process into them.

Say you have 4 logical CPUs and 4 memory nodes. You can specify mappings as `cpuset.cpus` = 0-2 (Indicates CPU nos 0,1,2 pinned) `cpuset.mems` = 0,3 (Indicates Memory nodes 0,3 pinned). Apart from the above mentioned parameters, there are several other parameter that can be configured. But the above two mentioned are of most interest.

5. Cpuacct

CPUacct as the name suggested, all it does is accounting; it doesn't exert control at all. It measures the total CPU time used by all processes in the group – per logical CPU/Total. Breaks into "user" and "system" time of the total CPU time used by a process group. It also reports a per CPU time consumed by each process group.

6. Block I/O

Block IO Keeps track of I/Os for each group using - Per block device, Reads vs Writes, Sync vs Async. It can sets throttle limits for each group using - Per block device, Reads or Write, Block Operations or Bytes. It can also set relative weights for each group.

7. Network class

The `net_cls` subsystem tags network packets with a class id that allows the Linux traffic controller (tc) to identify packets originating from a particular cgroup. `net_cls.classid` contains a single value that indicates a traffic control handle. The format for these handles is: 0xAAAABBBB, where AAAA is the major number in hexadecimal and BBBB is the minor number in hexadecimal. The traffic controller can be configured to assign different priorities to packets from different cgroups using simple command line commands.

8. Network Priority

The Network Priority (net_prio) subsystem provides a way to dynamically set the priority of network traffic per each network interface for processes within various cgroups. A network's priority is a number assigned to network traffic and used internally by the system and network devices. Network priority is used to differentiate packets that are sent, queued, or dropped.

Both Network class and Network Priority subsystems delegate their work to respective network interfaces, which leads to additional work at a lower layer. The traffic controller can be configured to assign different priorities to packets from different cgroups using simple command line commands for both the subsystems.

9. Device

Lets you control which devices can be accessed by a cgroup. By default a cgroup cannot access any device. Each entry has four fields: type, major, minor, and access. Access permissions are specified as Read(r), Write(w) and Modify(m). Modify allows to create device files that don't exist. It maintains an list of all devices with the above access permissions specified.

10. Perf Event

Collects various performance data for a process group. All cgroups in that hierarchy can be used to group processes and threads which can then be monitored with the perf tool, as opposed to monitoring each process or thread separately or per-CPU. Count hardware events such as instructions executed, cache-misses, suffered, or branches mis-predicted.

11. Freezer

Allows to freeze (suspend) / throw (resume) a process group. Processes in the cgroup are unaware of the freeze time. It is more frequently used for scenarios which require batch scheduling, migration etc.

12. Debug

Debug makes a number of internal details of individual groups, or of the cgroup system as a whole, visible via virtual files within the cgroup file system. Some data structures and the settings of some internal flags.

3.2 Namespaces

Now that we have understood how resource control and accounting support is provided by the Linux kernel to a group of processes. Lets move on to isolation. Linux Namespaces provides process groups

their own isolated system view. Namespaces exist for different types of resources as in the case of cgroups. Each process is in one namespace of each type. There were 10 namespaces proposed initially out of which 6 have been added to the current Linux kernel. These namespaces may or not have their own hierarchy similar to that of cgroups.

3.2.1 Changes to Kernel

Two new structs added and referred from the task_struct of every process.

1. `nsproxy` – which in-turn refer to 5 namespace structs
2. `cred` – user namespace pointer

Three Syscalls added/modified for supporting namespaces,

1. `unshare()` - Creates new namespace and attaches a process
2. `setns()` - Attaches a process to an exisiting namespace
3. `clone()` - is modified which in turn modifies fork() and exit()

6 New flags were added to clone() system call, one for each namespace. When any of the flag is set, the process gets created in new namespace of the corresponding flag. An inode is created for created for each namespace created. To view details of namespace of a process,

```
sudo ls -al /proc/<pid>/ns
```

3.2.2 Six existing namespaces

The following section describes the 6 namespaces existing currently. It is to be remembered that new namespaces are likely be added in the near future to support higher levels of isolations.

1. PID

PID namespace exists to isolate process views existing in a process group. Processes within a PID namespace only see processes in the same namespace. Each namespace has its own isolated numbering. PID numbers may overlap across different PID namespace. Each level of process group gives a different PID to a process. If PID-1 of a process group is killed the namespace is removed along with all its processes.

As seen from the Fig:3.2, PID namespaces follow a hierarchy similar to that of cgroups. The Fig, demonstrates how processes with PIDs 1,2,3 in the child namespace internally maps to 6,8,9 in the parent namespace.

2. Network

Each network namespace gets its own networking resources like - network interfaces (including loopback), routing table, IP table routes, sockets etc. You can move a network resource across network namespace.

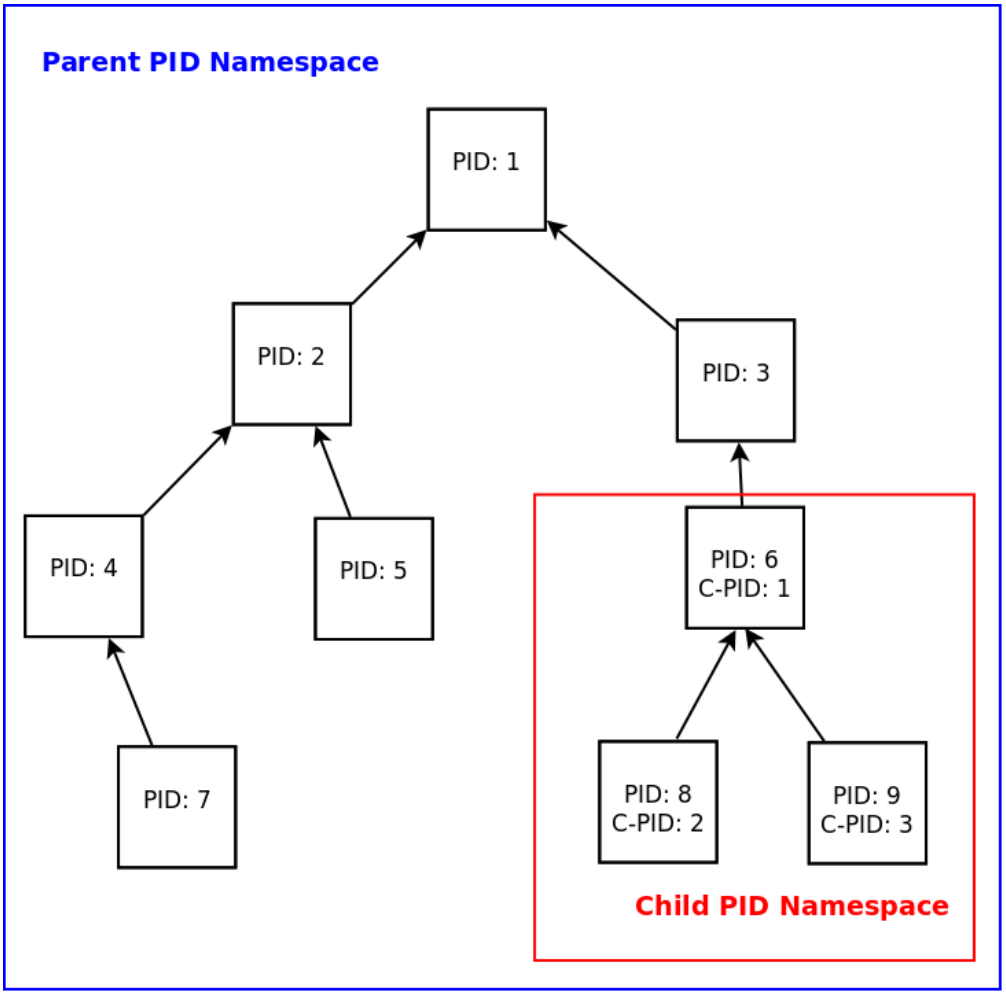

Figure 3.2: Example for PID namespace in Linux

Typically, a virtual network interface is used to network with host. Network namespace is the only namespace that can exist by itself without any processes attached to it. All other namespaces need to attach a process on creation.

3. Mount

Mount Namespaces was the first namespace to be introduced even before the proposal for namespaces were created. It can isolate the set of file system mount points seen by a process group. It creates environments that are similar to chroot jails, but are more secure. Can be set up in a master-slave relationship. They can have private/shared mounts. Mount namespace as a peculiar property compared to other namespaces and that is, on passing a new mount namespace flag, a new mount namespace is created by copying the existing mount points. Any new mount created in the new namespace is only local to the current namespace and is not visible in the parent namespace.

4. UTS

Lets process groups have its own host-name and NIS domain name. Isolates two system identifiers—node-name and domain-name. The names traditionally are set using the `sethostname()` and `setdomainname()` system calls. This can be useful to initialize and configure scripts that perform actions based on the two above mentioned parameters.

5. IPC

Allows a process groups to have its own - IPC Semaphores, IPC message queues and IPC shared memory. This is a very useful feature to make processes present in a namespace to only be able to communicate with other processes in the same namespace.

6. User

It is the newest namespace which was introduced in kernel version 3.8. Using user namespace you can be root inside a process group, but once you move out a process group, your just a normal unprivileged user. With the introduction of this new namespace an unprivileged process has root privileges in its own namespace and hence have access to functionality that was earlier only limited to root user. User namespace is relatively a new one, and not many container providers have still incorporated this feature.

3.3 Disk images

The final component that constitutes a container is a disk image. The disk image gives a ROOTFS, which contains the list of files and directory structure containing the root mount for a container. It typically has the structure of the root directory found in an Linux machine with lesser files. This disk image is usually smaller than the typical OS-disk image because it doesn't contain the kernel as it shares the kernel with the host machine.

A disk image could also contain only your application and make use of the host ROOTFS hence could even be as small as a few kilo Bytes. These disk images could also be designed in such a way that multiple containers could share a read-only disk image using COW (copy-on-write) techniques. The exact Implementation details of disk images in context of containers would be described in chapter 5.

3.4 Summary

Container makes use of the three above mentioned techniques and combine them to form the building structure. The exact Implementation details of containers would be described in the chapter 5.

4. Demonstration: cgroups and namespaces

It is rightly said that *"All study and no play makes Jack a dull boy"*. To avoid making Jack a dull boy, let's get started with some hands on. It is recommended to go through all the demos in this manual one after the other in sequence to get the real feel of the system, however feel free to try other options which you come across along the way. In this chapter we focus on short demos pertaining to cgroups and followed by namespaces. Later chapters we have a few more demos related to containers and disk images. Each demo section is typically followed by an exercise section, and its again strongly recommended to try out the same.

All demo's illustrated in the manual are run on an ubuntu 14.04 LTS distro running on top of Linux kernel version of 4.1.3. However it is not necessary for you to run the same kernel version. Anything **above kernel version 3.8 is recommended**, although most features(except for user namespaces) are supported by kernel versions 3.2 and up. Remember that cgroups and namespaces are enabled by the default kernel configs these days by the kernels. There might be cases where your kernel support for cgroups is turned off and if so recomplie your kernel and the whole cgroups module (including all the submodules) turned on. Enough said, let's get started.

4.1 Control Groups

This section gives a set of simple demonstration on memory subsystem to get you a feel of control groups. The current section first makes you mount a memory subsystem on a system.

4.1.1 Checking all the mounted cgroup subsystems

This demo helps you getting to know all the subsystems currently mounted onto your system. It gives you an overall picture of cgroup hierarchies existing on your system.

1. Navigate to the cgroups pseudo filesystem – it contains all the cgroup subsystems which are currently attached to thesystem,
   `cd /sys/fs/cgroup`
2. Login as root user,
   `sudo su`
3. List all the contents of directory,
   `ls`

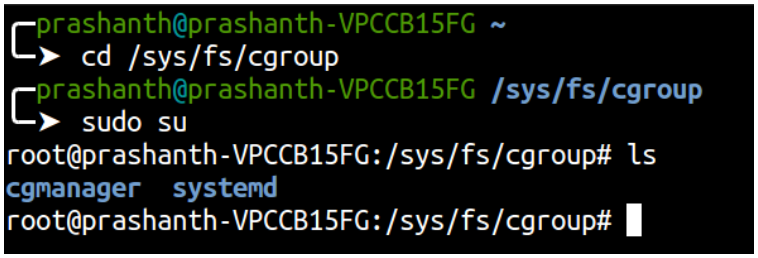

Figure 4.1: Checking current subsystems mounted

You show see a directories similar to as shown in Fig:4.1. These folders list the cgroups mounted in the system currently. Although directories like systemd & cgmanager aren't subsystems and have to more to with further options on cgroups, systemd is present my default and cgmanager (not relevent to you) is only on my machine. If you see additional directories, you need not worry as its a good thing, as they represent the cgroup subsystems already mounted on your system.

4.1.2 Mount/Unmount a New subsystem hierarchy

This section introduces you on how to attach a new hierarchy to your existing system with one or more cgroup subsystems. It allows demos how to unmount an existing subsystem if you want to. Fig:4.2 gives you a proper illustration of its working.

1. Create a new directory named memory (if it doesn't exist) – this is will be used to manage the memory subsystem,
   `mkdir memory`
2. Mount the memory cgroup (if it hasn't been mounted already). The options specified after -o represents the subsystems to be attached. You can specify multiple subsystems with comma separated values.
   `mount -t cgroup -o memory cgroup /sys/fs/cgroup/memory`
3. Change directory to memory
   `cd memory`
4. Now we are in the root memory cgroup, remember that a root cgroup is typically not configurable, it merely provides statistics and accounting information
5. List all the files in the current directory and see all the files present – each file corresponds to different attributes (maybe configurable) to the child cgroup
   `ls`
6. Try running a cat on most of the files and see what they display.
7. To unmount existing hierarchy, to the following
   ```
   cd ..
   umount memory
   rmdir memory
   ```

```
root@prashanth-VPCCB15FG:/sys/fs/cgroup# ls
cgmanager  systemd
root@prashanth-VPCCB15FG:/sys/fs/cgroup# mkdir memory
root@prashanth-VPCCB15FG:/sys/fs/cgroup# ls
cgmanager  memory  systemd
root@prashanth-VPCCB15FG:/sys/fs/cgroup# mount -t cgroup -o memory cgroup /sys/fs/cgroup/memory/
root@prashanth-VPCCB15FG:/sys/fs/cgroup# ls
cgmanager  memory  systemd
root@prashanth-VPCCB15FG:/sys/fs/cgroup# cd memory/
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory# ls
cgroup.clone_children           memory.kmem.tcp.failcnt             memory.soft_limit_in_bytes
cgroup.event_control            memory.kmem.tcp.limit_in_bytes      memory.stat
cgroup.procs                    memory.kmem.tcp.max_usage_in_bytes  memory.swappiness
cgroup.sane_behavior            memory.kmem.tcp.usage_in_bytes      memory.usage_in_bytes
lxc                             memory.kmem.usage_in_bytes          memory.use_hierarchy
memory.failcnt                  memory.limit_in_bytes               notify_on_release
memory.force_empty              memory.max_usage_in_bytes           release_agent
memory.kmem.failcnt             memory.move_charge_at_immigrate     tasks
memory.kmem.limit_in_bytes      memory.numa_stat                    user
memory.kmem.max_usage_in_bytes  memory.oom_control
memory.kmem.slabinfo            memory.pressure_level
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory# cd ..
root@prashanth-VPCCB15FG:/sys/fs/cgroup# umount memory
root@prashanth-VPCCB15FG:/sys/fs/cgroup# rmdir memory/
root@prashanth-VPCCB15FG:/sys/fs/cgroup# ls
cgmanager  systemd
root@prashanth-VPCCB15FG:/sys/fs/cgroup#
```

Figure 4.2: Mounting a new hierarchy

4.1.3 Creating a cgroup and adding process to it

Now that we have mounted a memory cgroup lets move on to creating a cgroup node and adding a process to it. Refer Fig:4.3

1. Create a new child cgroup of the root memory cgroup by making a new directory
   `mkdir memcg1`
2. Now navigate into this newly created cgroup, you could browse its contents which appear to be similar to its parent
   `cd memcg1`
3. procs file stores the tasks belonging to the current cgroup. Display the contents of this file, it initially is empty as no task exists
   `cat cgroup.procs`
4. Open a parallel terminal and start a new process and note its process id. (If you are creating a new firefox process, make sure it wasn't running earlier),
   `ctrl + shift + t`
   `firefox &`
5. Now come back to the original terminal and add the created process into the current cgroup (memcg1) as shown in
   `echo {{pid-of-process}} > cgroups.procs`

```
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory# mkdir memcg1
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory# cd memcg1
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory/memcg1# cat cgroup.procs
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory/memcg1# echo 29117 > cgroup.procs
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory/memcg1# cat cgroup.procs
29117
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory/memcg1# cat memory.stat
cache 4096
rss 421888
rss_huge 0
mapped_file 0
writeback 0
pgpgin 104
pgpgout 0
pgfault 103
pgmajfault 0
inactive_anon 0
active_anon 421888
inactive_file 4096
active_file 0
unevictable 0
hierarchical_memory_limit 9223372036854771712
total_cache 4096
total_rss 421888
total_rss_huge 0
total_mapped_file 0
total_writeback 0
total_pgpgin 104
total_pgpgout 0
total_pgfault 103
total_pgmajfault 0
total_inactive_anon 0
total_active_anon 421888
total_inactive_file 4096
total_active_file 0
total_unevictable 0
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory/memcg1#
```

Figure 4.3: Mounting a new hierarchy

6. Now display contents of tasks again and you will find the pids of all your added processes,
   `cat cgroups.procs`
7. Now you can view the memory statistics of the current cgroup by as shown in
   `cat memory.stat`

4.1.4 Setting constrains

1. Initially the memory limit is the parent cgroup node value, and this can be set using
   `echo 128M > memory.limit_in_bytes`
2. You could check various resource accounting information like current memory usage, maximum memory used, limit of on memory etc.
   `cat memory.usage_in_bytes`
   `cat memory.max_usage_in_bytes`
   `cat memory.limit_in_bytes`
3. One important parameter to track memory oom is failcnt, it lets us know how many times a cgroup has wanted to exceed its alloted limit
   `cat memory.failcnt`
4. Similarily memory.kmem.* and memory.kmem.tcp.* stats could be accounted/controlled
5. It is to be remembered that by default processes in memory that exceed usage limit maybe killed depending on the load of the system. To avoid this situation, you will need disable it as given below
   `echo 1 > memory.oom_control`

4.1.5 Exercise

Now that you know how to interface with cgroups as an exercise try to do the following,

CPU cgroup

1. Create a new CPU cgroup
2. Add a new CPU intensive process to it
3. Change its period, shares and quota
4. Note down the process behavior after doing the same
5. You can view that stats using cpu.stat
6. Repeat the same above steps a few times

Multiple cgroups on same hierarchy

1. Now that you are familiar with two cgroups, pick any two random cgroups (say- block i/o and net_prio) and try to combine to see cgroups to build a hierarchy
2. While mounting a new hierarchy using the mount command specify the cgroup you wish to use
3. Make use you disable the hierarchies that you wish to mount during kernel boot as you cannot mount multiple cgroups into multiple Hierarchies

```
root@prashanth-VPCCB15FG:/home/prashanth# ip netns add netns1
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 ping 127.0.0.1
connect: Network is unreachable
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 ip link set dev lo up
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.084 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.091 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.094 ms
^C
--- 127.0.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.084/0.089/0.094/0.011 ms
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 ping 192.168.200.1
connect: Network is unreachable
root@prashanth-VPCCB15FG:/home/prashanth#
```

Figure 4.4: Creating new namespace

4. Create a new cgroup in the newly created hierarchy and attach processes and vary parameters and note your observations

4.2 Namespaces

Now that we have gotten a feel of cgroups, lets move on to namespace. Namespaces are usually created by passing appropriate clone flags to the `clone()` system call. But network namespace provides us with simple command line interfaces to create new namespaces. We use network namespace for this reason, for the purpose of simplicity in demonstration, however it is advised to try the exercise section where you will create other namespaces using the `clone()` system call. This section gives a simple demonstration of network namespace.

4.2.1 Creating a new network namespace

This section demonstrates creating a new namespace and pinging its loopback interface.

1. Switch to root user
   `sudo su`
2. Create a new network namespace
   `ip netns add netns1`
3. List all the devices present in the newly created network namespace. We observe that the devices exposed in default network namespace isn't visible here. Only loop-back is present by default
   `ip netns exec netns1 ip link list`

4. Try pinging loop-back address or any other address. The network is unreachable as devices are either unavailable or not up (loopback)
   `ip netns exec netns1 ping 127.0.0.1`
5. To enable loop back,
   `ip netns exec netns1 ip link set dev lo up`
6. Now repeat step 4, and you are able to ping loop-back, but no other address as shown in
   `ip netns exec netns1 ping 192.168.200.1`

4.2.2 Creating virtual tunnel between default and new NS

Now that we have created a network namespace, let's try to create a tunnel between the newly created namespace and default namespace to route traffic between them.

1. Now we can create two virtual devices which forms a tunnel in the default namespace,
   `ip link add veth0 type veth peer name veth1`
2. Now we move one of the virtual devices to the newly created namespace
   `ip link set veth1 netns netns1`
3. We can look at IP tables and routing tables in the new namespace,
   `ip netns exec netns1 route`
   `ip netns exec netns1 iptables -L`
4. Now assign IPs to both the virtual devices
   `ifconfig veth0 192.168.100.1/24 up`
   `ip netns exec netns1 ifconfig veth1 192.168.100.2/24 up`
5. Now we can look at the devices on both the namespaces and see the difference,
   `ip link list`
   `ip netns exec netns1 ip link list`
6. We can now ping the virtual device on both namespaces, using the other as shown in Fig:4.5
   `ping 192.168.100.2`
   `ip netns exec netns1 ping 192.168.100.1`

Furthermore, a software bridge could be setup to provide Internet access to processes running on a network namespace.

4.2.3 Exercise

Network namespace is the only namespace that provides simple APIs to manage a namespace. All other namespaces make use of the system calls described in 3.2.1 along with their flags to create/manage processes between namespace.

Your exercise would be to create a new bash process in PID namespace and mount namespace, and run the top command to verify that you only view processes attached to your shell and that you

```
root@prashanth-VPCCB15FG:/home/prashanth# ip link add veth0 type veth peer name veth1
root@prashanth-VPCCB15FG:/home/prashanth# ip link set veth1 netns netns1
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@prashanth-VPCCB15FG:/home/prashanth# ifconfig veth0 192.168.100.1/24 up
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 ifconfig veth1 192.168.100.2/24 up
root@prashanth-VPCCB15FG:/home/prashanth# ping 192.168.100.2
PING 192.168.100.2 (192.168.100.2) 56(84) bytes of data.
64 bytes from 192.168.100.2: icmp_seq=1 ttl=64 time=0.167 ms
64 bytes from 192.168.100.2: icmp_seq=2 ttl=64 time=0.095 ms
^C
--- 192.168.100.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.095/0.131/0.167/0.036 ms
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 ping 192.168.100.1
PING 192.168.100.1 (192.168.100.1) 56(84) bytes of data.
64 bytes from 192.168.100.1: icmp_seq=1 ttl=64 time=0.114 ms
64 bytes from 192.168.100.1: icmp_seq=2 ttl=64 time=0.109 ms
^C
--- 192.168.100.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.109/0.111/0.114/0.010 ms
root@prashanth-VPCCB15FG:/home/prashanth#
```

Figure 4.5: Creating new namespace

have new process-IDs to your processes which initialize with 1. For reference use the following link www.lwn.net/Articles/531114/ . Implement the same using C, remember to use the system calls.

5. Virtualized Containers

Container makes use of the three mentioned components in the previous section i.e - cgroups, namespaces and disk images and combine them to form the building structure. Container is a set of processes grouped together along with its dependent resources into a single logical OS entity. It enables multiple isolated user-space instances on a host machine. This is achieved by sharing the OS kernel among the different containers running on the host machine and hence is also referred to as OS-Virtualization. It is to be remembered that by sharing the host kernel with the container, we are actually exposing the host system call table to the container.

Control groups are used to attain resource control i.e tracking resource usage and imposing quantitative limits. Typically all physical resources can be controlled using cgroups however, controlling network bandwidth alloted to a container is not currently supported by any of the container managements systems. There are few articles over the Internet which mention work around for this, but this has to be done from outside the container and at a kernel level.

Kernel namespaces are used to limit the visibility and reach of processes to only resources provisioned to its corresponding container. Typically, for every container a new mount namespace is initialized and on this new mount namespace a new ROOTFS is mounted on to the root. Of course this is a typical usage, but however there might be cases where mount points are shared with the host system or other containers and so on.

Containers are not a part of Linux kernel, but only cgroups and namespaces are. Containers are usually designed by a vendor own makes use of the above kernel features to implement the required basic functionality and adds their own proprietary features on top of it and release them as a commercial product/projects and such products/projects are referred to as Container Managers. Containers are classified into two types - system and application containers which have been described below.

5.1 System Containers

Containers whose virtual environment provides is similar to a native machine is often referred to as system or full containers. You can install, configure and run different applications, libraries, demons, etc., just as you would on any OS. It is more useful to someone like a cloud PAAS service provider. Such containers have been used by cloud providers for a while now. Examples of system containers are

- Linux containers (lxc)

- Parallels Virtuizzo
- Solaris Zones

5.1.1 Linux Containers (lxc)

lxc provides an userspace API to deploy containers using the underlying kernel features. It helps in easy creation and management of system containers. lxc provides wide range of APIs for a user to deploy containers few of which are - Go, ruby, python, Haskell. lxc is a free software and is licensed under GNU. lxc was first launched in the year 2008. Recently lxc has come up with its latest stable release of 2.0.

lxc uses cgroups to limit resource limits. A basic configuration file is generated at container creation time with default keys coming from the default.conf file. The configuration file can specify various cgroup parameters as options. These configurations are then mapped to their corresponding cgroup subsystem. For example setting the memory limit at 2GB would indicate creating a new cgroup in memory subsystem with a limit of 2GB and attaching the processes of the container to this cgroup. Similarly other resources of a container would be mapped to a new node in each subsystem.

Isolation is mapped by specifying `CLONE_NEWIPC, CLONE_NEWNET, CLONE_NEWPID, CLONE_NEWUSER, CLONE_NEWUTS, CLONE_NEWNS (mount)` flags to a `clone()` system call. This uses the previously described namespaces concept to keep the container in a new namespace. lxc is one of the few new container version which support user namespaces currently where an user can be mapped to root inside a container but is just a normal user outside it. Further more the a new ROOTFS is mounted using a base container image. The base container image is fetched from an online repository for the first time and is their after is fetched from the local cache. Each time a new copy of the ROOTFS is made on which a new container makes its changes. Running an existing container makes use of its previously used ROOTFS to execute.

5.2 Application Containers

Application containers on the other hand provide a more similar execution environment. They are used as an virtual environment for application development and testing. It allows minimalistic configurations with no demon processes. Application containers have gained popularity over the last few years. It is often used by devops to develop new applications, build, test and finally ship the application. It is often recommended by application container managers to deploy only one application per container and deploy multiple applications by creating one container for each application. Examples of application containers are,

- Docker
- Rocket

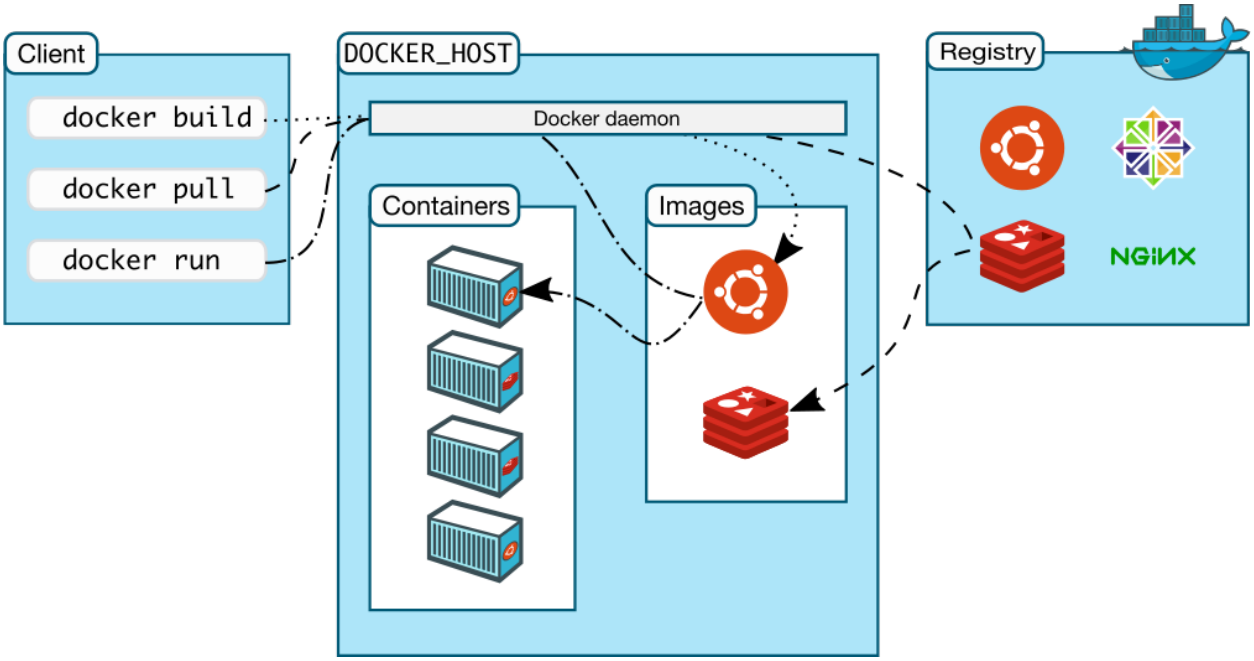

Figure 5.1: Docker Architecture, source:[1]

5.2.1 Docker

Docker is an application container manager. It is an open platform primarily used for developing, shipping and running applications. Application developers can write code locally and share their development stack with their team. When the application is ready, they push the same stack into a test environment where the application is further tested. From here, you can push the code into deployment. It can run on either a local host machine, physical or virtual machines in a data center, or on a cloud.

Docker similar to lxc uses cgroups to limit resource limits. A basic configuration file is generated at container creation time with default keys coming from the default.conf file. The configuration file can specify various cgroup parameters as options. The mapping is similarly to how resources of a container was mapped to a new node in each subsystem as described for lxc.

Isolation is mapped by specifying `CLONE_NEWIPC, CLONE_NEWNET, CLONE_NEWPID, CLONE_NEWUTS, CLONE_NEWNS (mount)` flags to a `clone()` system call. This uses the previously described namespaces concept to keep the container in a new namespace. Unlike lxc docker deosn't yet have the support for user namespace to run a container in a local root privilege. But this support is expected to come out soon in the newer docker versions. Disk images are more optimized by Docker and these optimizations are described in the upcomimg sections.

Docker Architecture

Components of Docker architecture are,

- Client: User interface to Docker using commands

- Images: Read-only template, used to create Docker containers
- Daemon: Building, running, and distributing Docker containers
- Registry(hub): public or private(paid) stores from which you upload or download images
- Container: Created from a Docker image. Can be run, started, stopped, moved, and deleted

The Docker client and host as illustrated in Fig:5.1 form a client-server architecture. They can both be on the same machine or even on different machines. Communication between client-host-registry happens using RESTful API.

Docker Images

Two key technologies behind Docker image and container management are stackable image layers and copy-on-write (CoW). They make use of a overlay FS like AUFS to combine layers into a single image. We would refer it to as COW. The main advantage of using COW mechanism is reduce storage space. This is where Docker leads over other container technologies. COW disk images have showed great potentials in lowering the amount of storage space required but this comes at a cost of minimal performance overhead.

Each Docker image references a list of read-only layers that represent filesystem differences. Images are read-only templates. Each images consists of a series of layers. These layers are content addressable since Docker version 1.10 and hence can be reused by other images which use the same.

There are again two kinds of images - one is a base image and another is an image created from a base image. Consider running a container from an ubuntu image which is directly fetched from the Docker registry, now this image is a base image. Now, given a base image the user can add layers over it like,

- Run a command
- Add a file or directory
- Create an environment variable.
- What process to run when launching a container from this image

And such instructions above are stored in a file called the DOCKERFILE. While building an image using a DOCKERFILE, it adds a new layer on top of existing base image for ever instruction to give an final image with several layers as shown in Fig:5.2 as the layers 2 to 4 within the ubuntu 14.04 image.

Docker Containers

When you create a container using Docker, the Docker client pulls the requested image (if doesn't exist locally, pulls from registry). Creates new container using this image. The container adds a Read-Write (R/W) layer on the image on top of the read-only image layers on which all the changes to the container are stored, thus changes made to container doesn't affect its image. Hence multiple containers can use the same image and by only adding one additional R/W layer for each container as shown in Fig:5.2. On deleting a container only this R/W layer is deleted.

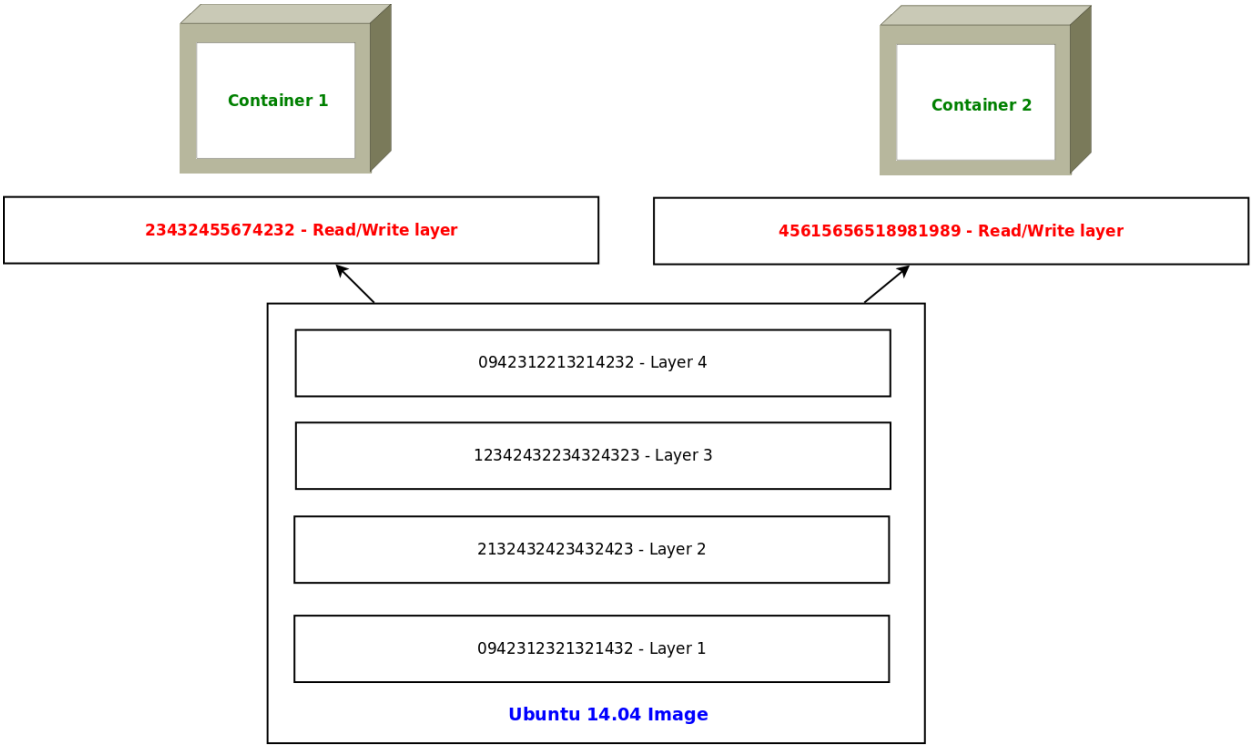

Figure 5.2: Disk layers in Docker

Data volumes

Lastly, Docker volumes could be used for multiple containers to share common data by mounting a shared mount point onto multiple containers. This can be used to share data to process among containers.

5.3 Summary

This chapter introduced the various components that make up a container. It also discussed how these components interact with each other to give us the real container. It also described the different types of containers and concluded with a discussion on docker containers.

6. Demonstration: lxc and Docker

In the previous demonstration section, we looked at how to work with cgroups and namespaces. Now, in this section we will look at how to work with containers discussed in the previous chapter. We will start with using lxc and then move on to Docker. We have selected lxc and Docker so as to give you an hands-on on both system and application containers. We conclude to demonstration with an demonstration on Dockers copy-on-write(COW) based disk images, to give an more boarder perspective on disk images used in Docker and how they differ from typical disk images used in other container managers.

The installation demos are specific to ubuntu, and commands may change based on your distro. We highly recommend you to use ubuntu at this point because container management systems are well versed with ubuntu and other distros might cause trouble.

6.1 Linux Containers (lxc)

This section gives a set of demonstration on setting up of lxc and how to use it to manage containers. The lxc version for the purposes of demonstration is 1.1 although, the version of lxc shouldn't be making much of a difference.

6.1.1 Installing and running a simple container

This demo aims at installing lxc in your system and running a simple container. It is highly recommended to use Linux kernel version of 3.8+ as this is a requirement for unprivileged root containers discussed in the user namespace section and lxc currently supports this feature.

1. Switch to root user
   `sudo su`
2. To install lxc, run the following command
   `apt-get install lxc`
3. Create a new ubuntu container, this process might take time for the first time as it will have to fetch the ubuntu image from its repository. Once you have a local copy of your image, the time taken to deploy a container there after would be lesser. -n specfies the name of the container and -t specifies the image template to be used.
   `lxc-create -t ubuntu -n ubuntu-01`

```
Ubuntu 14.04.4 LTS ubuntu-01 console

ubuntu-01 login:  * Stopping save kernel messages   ...done.
ubuntu
Password:
Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 4.1.3 x86_64)

 * Documentation:  https://help.ubuntu.com/

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

ubuntu@ubuntu-01:~$ cd /
ubuntu@ubuntu-01:/$ ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
ubuntu@ubuntu-01:/$ hostname
ubuntu-01
ubuntu@ubuntu-01:/$ sudo poweroff
[sudo] password for ubuntu:

Broadcast message from ubuntu@ubuntu-01
        (/dev/lxc/console) at 11:37 ...

The system is going down for power off NOW!
ubuntu@ubuntu-01:/$ wait-for-state stop/waiting
 * Asking all remaining processes to terminate...
   ...done.
 * All processes ended within 1 seconds...
   ...done.
 * Deactivating swap...
   ...fail!
 * Unmounting local filesystems...
   ...done.
mount: / is busy
 * Will now halt
root@prashanth-VPCCB15FG:/home/prashanth# lxc-info -n ubuntu-01
Name:           ubuntu-01
State:          STOPPED
root@prashanth-VPCCB15FG:/home/prashanth# sudo lxc-destroy -n ubuntu-01
Destroyed container ubuntu-01
root@prashanth-VPCCB15FG:/home/prashanth#
```

Figure 6.1: Running a lxc container

4. Start the created container using, -F specifies to run the container in foreground
   `lxc-start -F -n ubuntu-01`
5. You would be directed to a terminal which asks for your username and password. Default username and password is ubuntu
6. Once you are logged in, now you are inside the ubuntu container. Browse through the container and you would realise that the whole container is isolated from the host system. You have your own ROOTFS all though your kernel is still shared with the host system.
7. Try running top and have a look at the processes running
   `top`
8. To come out of a container, stopping the container

```
sudo poweroff
```

9. To destroy a container permanently, Refer Fig:6.1
   ```
   lxc-destroy -n ubuntu-01
   ```

6.1.2 Some helpful commands in lxc

A beginner might be overwhelmed with the list of commands lxc provides. Here is demo of the useful commands in lxc.

1. To start and run a container in background
   ```
   lxc-start -d -n ubuntu-01
   ```
2. To attach a console to running container
   ```
   lxc-console -n ubuntu-01
   ```
3. To attach unattach shell from running container
   ```
   Ctrl + a
   q
   ```
4. To freeze a container
   ```
   lxc-freeze -n ubuntu-01
   ```
5. To unfreeze a container
   ```
   lxc-unfreeze -n ubuntu-01
   ```
6. To clone a container
   ```
   lxc-clone -o ubuntu-01 -n ubuntu-01-cloned
   ```

6.1.3 Python API to use lxc

This demonstrates managing lxc using python API. It creates a container from python and runs the same.

1. Switch to python environment using sudo privileges
   ```
   sudo python3
   ```
2. Import lxc library
   ```
   import lxc
   ```
3. Initialize a new container class
   ```
   container=lxc.Container("container-1")
   ```
4. Create a new container
   ```
   container.create("ubuntu")
   ```
5. Start a new container
   ```
   container.start()
   ```

```
root@prashanth-VPCCB15FG:/home/prashanth# ls /var/cache/lxc
trusty
root@prashanth-VPCCB15FG:/home/prashanth# ls /var/lib/lxc/
ubuntu-01
root@prashanth-VPCCB15FG:/home/prashanth# cat /var/lib/lxc/ubuntu-01/config
# Template used to create this container: /usr/share/lxc/templates/lxc-ubuntu
# Parameters passed to the template:
# For additional config options, please look at lxc.container.conf(5)

# Uncomment the following line to support nesting containers:
#lxc.include = /usr/share/lxc/config/nesting.conf
# (Be aware this has security implications)


# Common configuration
lxc.include = /usr/share/lxc/config/ubuntu.common.conf

# Container specific configuration
lxc.rootfs = /var/lib/lxc/ubuntu-01/rootfs
lxc.utsname = ubuntu-01
lxc.arch = amd64

# Network configuration
lxc.network.type = veth
lxc.network.link = lxcbr0
lxc.network.flags = up
lxc.network.hwaddr = 00:16:3e:28:d3:a7
root@prashanth-VPCCB15FG:/home/prashanth# ls /var/lib/lxc/ubuntu-01/rootfs
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys
root@prashanth-VPCCB15FG:/home/prashanth#
```

Figure 6.2: Exploring lxc images

6. Stop the container
   `container.stop()`

   You will notice that on running top Docker contains only about a couple of processes, where as running the same command in lxc shows many processes. This illustrates the difference between an system and application container.

6.1.4 Disk Images in lxc

This demonstrates how disk images are handled in lxc. It walks through the host fs to show where the actual files lie in a default configuration. Fig:6.2 shows this.

1. All the local cache for images exist at /var/cache/lxc/
   `ls /var/cache/lxc/`
2. Each of the container created exists at /var/lib/lxc/, and appear as the container name
   `ls /var/lib/lxc/`
3. The config file for a container can be found at /var/cache/lxc/¡container-name¿/config. The configurations relating to resource constrains, setup etc. will be specified here

```
cat /var/lib/lxc/ubuntu-01/config
```

4. This is the actual directory containing the root mount for a container
   `ls /var/lib/lxc/ubuntu-01/rootfs`

   Notice that the ROOTFS displayed here matches with the one running on the container. Hence lxc uses this same directory as the mount point to the container. For each created container there exists a ROOTFS at /var/lib/lxc/¡container-name¿/rootfs

6.1.5 Exercise

Create a ubuntu container running on vivid version of ubuntu. Set the resource limits of this container to say memory of 256M, cpu pinned to 0-1 etc by making changes to the config file of that container. Now run the container and run applications to fully utilize the memory and see what happens. Does it behave similar to memory cgroups discussed in chapter 4 ?

6.2 Docker

This section gives a set of demonstration on setting up of Docker and how to use it to manage application containers. The Docker version for the purposes of demonstration is 1.10 although, the version of Docker shouldn't be making much of a difference.

6.2.1 Installing required dependencies for Docker

This demo aims at installing the dependencies for installing Docker in your system and running a simple container.

1. Switch to root user
   `sudo su`
2. Update your current system
   `apt-get update`
3. Make sure the APT works fine with https and CA certificates installed
   `apt-get install apt-transport-https ca-certificates`
4. Add new GPG (GNU Privacy Guard) key
   `apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80`
   `--recv-keys 58118E89F3A912897C070ADBF76221572C52609D`
5. Open /etc/apt/sources.list.d/Docker.list with any editor and remove all existing content and replace it with the the line given below (Make sure to use your ubuntu version) and now save and close vim `vim /etc/apt/sources.list.d/Docker.list`
   `ADD ‘‘deb https://apt.Dockerproject.org/repo ubuntu-trusty main’’`

6. Again update your package index
   `apt-get update`
7. Purge the old repository if it exists
   `apt-get purge lxc-Docker`
8. Verify that pulling is from the right repository
   `apt-cache policy Docker-engine`
9. Again update your package index
   `apt-get update`
10. One last package
    `apt-get install linux-image-extra-$(uname -r)`

6.2.2 Installing Docker

Now lets get to the actual installation of the Docker engine.

1. Update your package index
   `apt-get update`
2. Install Docker engine
   `apt-get install Docker-engine`
3. Start Docker service
   `service Docker start`
4. To verify installation run
   `Docker run hello-world`

6.2.3 Running a Docker Container

This demonstrates creating, running, stopping and removing a Docker container running ubuntu distro. Fig:6.3 shows this.

1. This command creates and runs a new Docker container in a single command.
   `Docker run --name my-container -i -t ubuntu /bin/bash`
2. Now you will be switched into the container. Now navigate around to verify the same. Run simple commands like
   `ls /`
   `hostname`
3. Try running top and have a look at the processes running `top`
4. To leave to container, exit the shell
   `exit`

```
root@prashanth-VPCCB15FG:/home/prashanth# docker run --name my-container -i -t ubuntu /bin/bash
root@953b2e3ac6c3:/# ls /
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@953b2e3ac6c3:/# hostname
953b2e3ac6c3
root@953b2e3ac6c3:/# exit
exit
root@prashanth-VPCCB15FG:/home/prashanth# hostname
prashanth-VPCCB15FG
root@prashanth-VPCCB15FG:/home/prashanth# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PO
953b2e3ac6c3        ubuntu              "/bin/bash"         35 seconds ago      Exited (0) 19 seconds ago
root@prashanth-VPCCB15FG:/home/prashanth# docker stop my-container
my-container
root@prashanth-VPCCB15FG:/home/prashanth# docker rm my-container
my-container
root@prashanth-VPCCB15FG:/home/prashanth# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
root@prashanth-VPCCB15FG:/home/prashanth#
```

Figure 6.3: Running Docker container

5. Now run hostname again and you would find it return the host machine's hostname
   `hostname`
6. To look at all containers present on Docker
   `Docker stop my-container`
7. To delete the container permanently
   `Docker rm my-container`

Running top in lxc displayed several processes, but Docker shows a very less number. This illustrates the difference between application and system container.

6.2.4 Disk Image layers in Docker

Lets move on to one of the final and one of the most import demonstration of this discussion, which is the Docker images. This demonstrates how disk image layers are handled in Docker.

1. You can list all the cached images on your local machine using, I have a lot of images displayed but you may have one or none
   `Docker images`
2. Now lets pull a new image
   `Docker pull ubuntu:15.04`
3. Now list the images again and you will find the new image in the cache as shown in Fig:6.4
   `Docker images`
4. This is the actual directory containing the root mount for a container
   `ls /var/cache/lxc/ubuntu-01/rootfs`
5. Now lets pull another new image
   `Docker pull ubuntu:15.10`

```
root@prashanth-VPCCB15FG:/home/prashanth# docker images
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
trusty                    latest              bd3d156d2df9        5 weeks ago         228.3 MB
kvmprashanth/static_web   latest              c67bfae4022e        5 weeks ago         227.6 MB
kvmprashanth/apache2      latest              2a8ea991d44a        5 weeks ago         223.8 MB
changed-ubuntu            latest              3e6d9a528b6a        7 weeks ago         187.9 MB
fedora                    latest              760a896a323f        7 weeks ago         204.5 MB
ubuntu                    latest              14b59d36bae0        7 weeks ago         187.9 MB
busybox                   latest              3240943c9ea3        7 weeks ago         1.114 MB
hello-world               latest              690ed74de00f        5 months ago        960 B
root@prashanth-VPCCB15FG:/home/prashanth# docker pull ubuntu:15.04
15.04: Pulling from library/ubuntu

9502adfba7f1: Pull complete
4332ffb06e4b: Pull complete
2f937cc07b5f: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:2fb27e433b3ecccea2a14e794875b086711f5d49953ef173d8a03e8707f1510f
Status: Downloaded newer image for ubuntu:15.04
root@prashanth-VPCCB15FG:/home/prashanth# docker images
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
trusty                    latest              bd3d156d2df9        5 weeks ago         228.3 MB
kvmprashanth/static_web   latest              c67bfae4022e        5 weeks ago         227.6 MB
kvmprashanth/apache2      latest              2a8ea991d44a        5 weeks ago         223.8 MB
changed-ubuntu            latest              3e6d9a528b6a        7 weeks ago         187.9 MB
fedora                    latest              760a896a323f        7 weeks ago         204.5 MB
ubuntu                    latest              14b59d36bae0        7 weeks ago         187.9 MB
busybox                   latest              3240943c9ea3        7 weeks ago         1.114 MB
ubuntu                    15.04               d1b55fd07600        10 weeks ago        131.3 MB
hello-world               latest              690ed74de00f        5 months ago        960 B
```

Figure 6.4: Docker pulling images

```
root@prashanth-VPCCB15FG:/home/prashanth# docker history ubuntu:15.04
IMAGE               CREATED             CREATED BY                                      SIZE
d1b55fd07600        10 weeks ago        /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
<missing>           10 weeks ago        /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/   1.879 kB
<missing>           10 weeks ago        /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic   701 B
<missing>           10 weeks ago        /bin/sh -c #(nop) ADD file:3f4708cf445dc1b537   131.3 MB
root@prashanth-VPCCB15FG:/home/prashanth# docker history ubuntu:15.10
IMAGE               CREATED             CREATED BY                                      SIZE
4e3b13c8a266        3 days ago          /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
<missing>           3 days ago          /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/   1.863 kB
<missing>           3 days ago          /bin/sh -c set -xe   && echo '#!/bin/sh' > /u   701 B
<missing>           3 days ago          /bin/sh -c #(nop) ADD file:43cb048516c6b80f22   136.3 MB
root@prashanth-VPCCB15FG:/home/prashanth#
```

Figure 6.5: Difference between layers of a same image

6. Now to show that Docker uses reusable layers, try the following commands
   `Docker history ubuntu:15.04`
   `Docker history ubuntu:15.10`

Looking at Fig:6.5 you will notice that image layer-3 between ubuntu:15.04 and ubuntu:15.10 is the same and it reuses this. This proves Docker using COW layers for images.

6.2.5 Exercise

Open two containers on two terminals side by side using lxc and Docker. Now note the differences in the processes running inside them, start-up time, and explore other areas where they differ.

6.3 Summary

Now that you have gotten a basic understanding of containers with decent amounts of hands-on, you can explore their overlapping features and how you can use them to your requirement. Containers is a relatively fast evolving technology, so I would suggest you to stay updated with the latest release and changes by container management system. Some related works to containers are discussed in the following chapter.

7. Related Work

There are several other technologies that have already come up or are just coming up which address similar problems which are addressed by container. This section provides a brief introduction to such technologies.

7.1 CoreOS

CoreOS is an linux distro designed exclusively for container management. It initially made use of Docker to manage its internals, but in 2014 they released their own container management system which is now known as Rocket. Now rocket lies in the kernel of CoreOS and is used as its container manager.

CoreOS, the minimal Linux distribution designed for massive server deployments, the upgrade processes are controlled by a cgroup. This means the downloading and installing of system updates doesn't affect system performance. Some people see huge potential of CoreOS in cloud provides of Platform as a service (PAAS). Only time will tells us if its actually feasible.

7.2 Virtual Machines

A Virtual Machine (VM)[4] is an isolated virtual environment created by OS that is running upon another piece of specialized software called hypervisor. Hypervisor typically emulates virtual devices with the help of other software mechanisms to virtual machines to treat them as real hardware. Hypervisor itself maybe either running directly on the physical hardware or on a host OS and they are called Bare-metal(Type-1) and Hosted(Type-2) hypervisors respectively. The end user has the same experience on a virtual machine environments as they would have on OS attached real physical hardware. They provide highly secure and independent systems. Virtual machines can even be configured to limit the amount of resources available to them by the hypervisor.

Virtual machines have been extensively used by cloud providers to host customers for over a decade now. Many people believe that containers are here to replace VMs as containers provide lesser overheads compared to VMs and provide similar features. But it is also to be remembered that you can create a container guest OS which is based on the same kernel as the host OS, meaning you can only run a linux based distro OS on a linux based host OS and not any other OS. But with a VM you could potentially have any host OS and run any other guest OS on top of it. And moreover containers are prone to security

glitches due to sharing of host syscall table. Hence there are many VM supporters who claim that VMs would eventually become as fast as containers with the proper hardware support.

7.3 OSv

OSv is an operating system designed for the Cloud and is treated as a library operating system. It is intended to run on top of an hypervisor. It provides isolation benefits of VM but with lesser overheads. The disadvantage of OSv is that it supports limited system calls and hence applications will have to be ported with changes to support it. It also concludes that containers are equal or better than hypervisors in most dimensions.

7.4 LXD

LXD is being called as the next generation hypervisor for containers. It is built on top of existing lxc framework but adds a demon similar to an traditional hypervisor. It claims to provide better levels of isolation and security compared to traditional container technologies, although this claim is still to be tested at depths. LXD promises to bring more support from hardware vendors to support containers just like how hardwares like VTX support hypervisors today. Containers have traditionally suffered to perform live migration, and this technology promises to address this issue. If all the claims offered by LXD are looked into, it should be really worth to have a look at as an replacement to the existing container managers.

8. Conclusion

In conclusion, containers have been here for a while now and are close to reaching stability. The major limitation of a container is the ability to only run OS of host kernel type, and not other OSes which are of a different kernel. There are several articles which speak about its security flaws due to the exposure of host system call interface. However this problem has been addressed with more care over the years but hasn't been completely solved yet.

Linux cgroups and namespaces have provided an amazing platform for a system like containers the develop however, more efforts need to be put in stabilizing its security flaws. COW disk images have showed great potentials in lowering the amount of storage space required but this comes at a cost of minimal performance overhead and to balance between the two is a requirements decision.

Virtual machines has been a great technology but the performance overheads it carries along with is a heart breaker. System containers stretch this arena by providing something in the between isolation and performance provided by native and virtual machines. It provides isolation close to that of virtual machines, at a performance cost of near native.

Newer technologies like CoreOS, LXD have to be explored in more depth by cloud providers to see how efficiency is handled by these technologies. LXD's advanced security mechanisms would have to be looked into and see if they provide better forms of isolation by not compromising much on performance.

Application container managers like Docker, Rocket are excellent choices for application developers as they provide an isolated environment for development which makes it easy to transfer from one person to another along with its dependencies. Application containers generally provide better performance then system containers, however it must be remembered that both have a different purpose as application containers are typically only used to hold a single application instance, system containers on the other hand provide a complete OS feel.

In all containers are here to stay, and have tremendous potential for the future in virtualization for usage across different areas by providing good amounts of resource control and isolation by with minimal overhead. But only time will tell us if it would actually replace virtual machines or not.

Bibliography

[1] D. Inc., "Docker offical documentation," 2016.

[2] M. Campbell-Kelly and W. Aspray, "A history of the information machine," *Notes*, vol. 30, no. 1, 1996.

[3] A. Silberschatz, P. B. Galvin, G. Gagne, and A. Silberschatz, *Operating system concepts*, vol. 4. Addison-Wesley Reading, 1998.

[4] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, 2005.

[5] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *OSDI*, vol. 99, pp. 45–58, 1999.

[6] D. P. Bovet and M. Cesati, *Understanding the Linux kernel.* " O'Reilly Media, Inc.", 2005.

[7] K. Kolyshkin, "Virtualization in linux," *White paper, OpenVZ*, vol. 3, p. 39, 2006.

[8] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 275–287, ACM, 2007.

[9] P. B. Menage, "Adding generic process containers to the linux kernel," in *Proceedings of the Linux Symposium*, vol. 2, pp. 45–57, Citeseer, 2007.