Prepare Virtualization:

With the QEMU (Quick Emulator) which is a generic and open source machine emulator we emulated the machines processors through dynamic binary translation. We used this in combination with the Kernel-based virtual machine (KVM) which is the virtualization module in the Linux kernel that allows the kernel to function as a hypervisor. KVM is based on the hardware virtualization techniques of Intel (VT) or AMD (AMD-V). With this QEMU basically emulated the target CPU architecture by translating machine instructions and system calls. (QEMU is basically the backend all in one hypervisor, it is a type 2 hypervisor).

Interaction between QEMU and KVM:

As previously mentioned, QEMU can run independently, but due to the emulation being performed entirely in software it is extremely slow. To overcome this, QEMU allows you to use KVM as an accelerator so that the physical CPU virtualization extensions can be used. So to conclude: QEMU is a type 2 hypervisor that runs within user space and performs virtual hardware emulation, whereas KVM is a type 1 hypervisor that runs in kernel space, that allows a user space program access to the hardware virtualization features of various processors.[3]

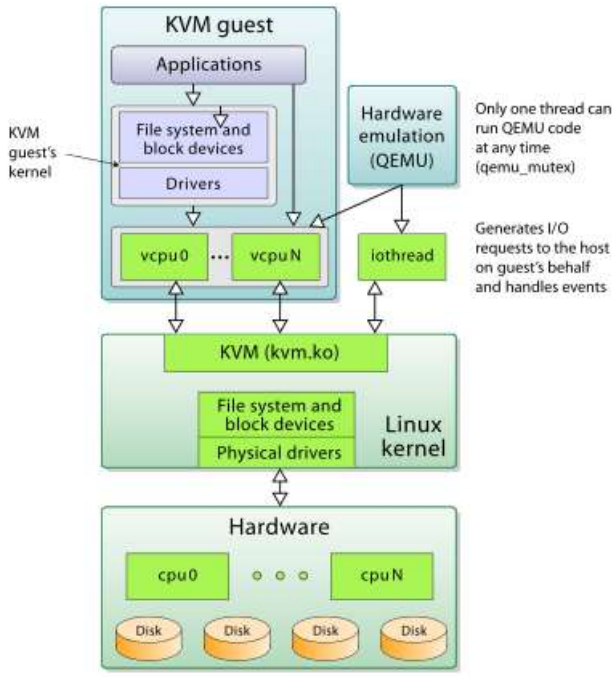


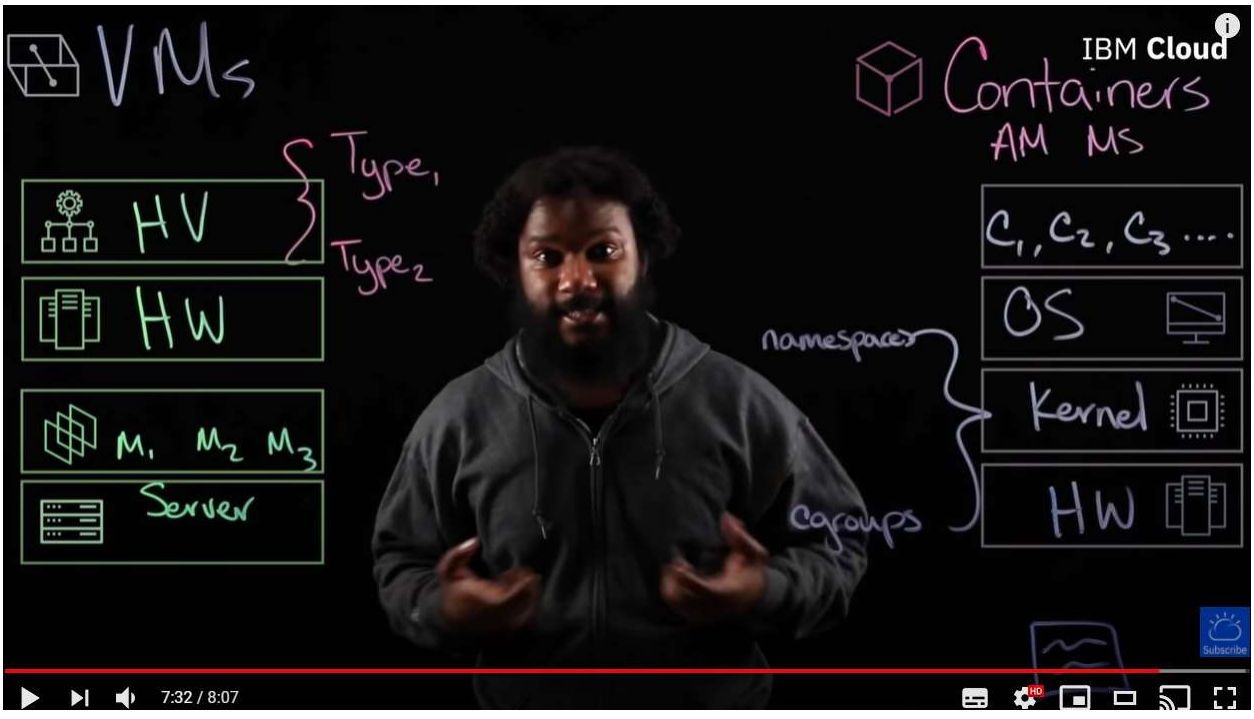
Figure 1 - High-level overview of the KVM/QEMU virtualization environment.[4]

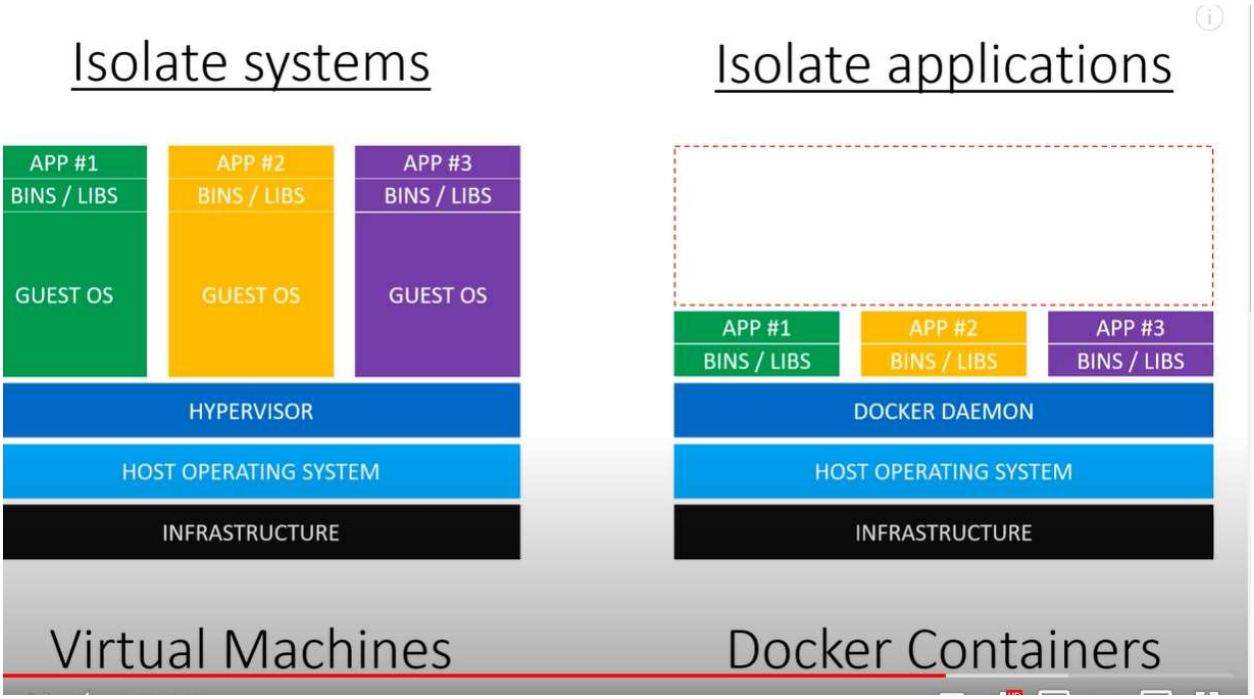
With QEMU we created a disk image which is a template of the emulated environment that we want (or a snapshot of a system at a particular time). That image we need to establish the container because a container is a running instance of an image. On that image we uploaded Linux as the underlying operating system. To interact with QEMU and KVM we could use libvirt which an abstraction on top of these lower level interactions. It provides an API to interact with the lower level systems. Libvirt is also a daemon which is computer program that runs as a background process, rather than being under the direct control of an interactive user.

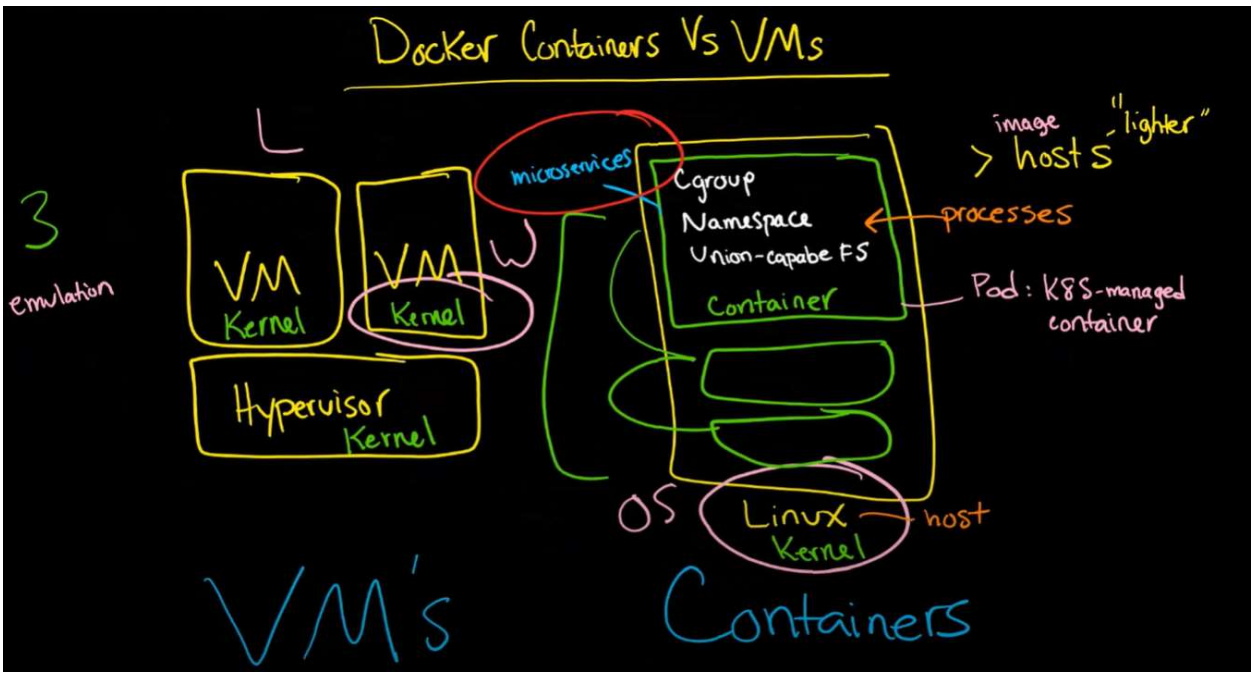
After installing QEMU as well as installing Ubuntu on the disk image we installed docker. Docker is an open source project to pack, shift and run any application as a lightweight container.

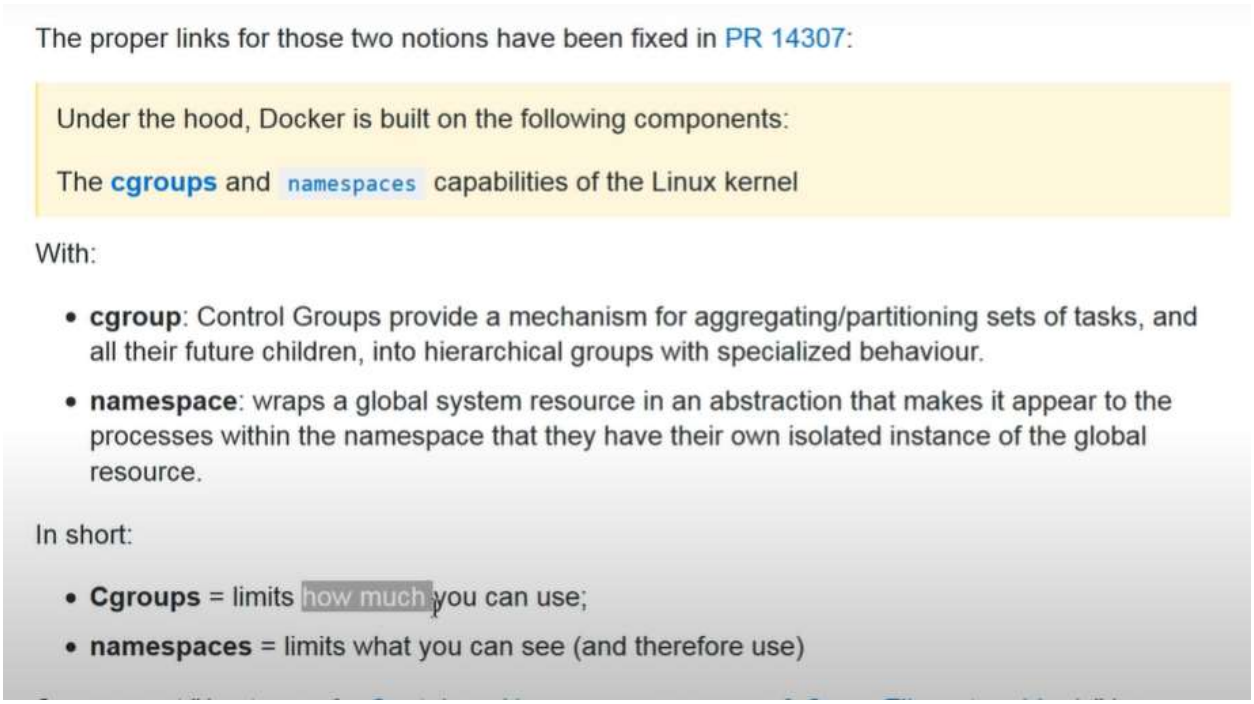
A Container allows to run applications without any operating system on bare metal servers (It's like wine for applications, but it doesn't need Linux). I can run multiple containers using multiple operating systems on one VM, instead of using multiple VM's. The usage of containers is also called OS virtualization or process isolation (explained like this directly from IBM). An operating system is adapted so it functions as multiple, discrete systems to support different users running applications simultaneously on a single machine. OS virtualization makes it possible to deploy and run distributed applications without launching an entire VM for each application. Instead, multiple isolated systems, called containers, are run on a single control host and access a single kernel. One advantage of containers is, that we have infinite portability. Because our container is being defined in a single file, (called docker file when using docker). We have a few lines of text which are saying exactly how to run our container, with what libraries etc. and we can run our application.

You can think of VMs as houses and containers as apartments.




Isolate systems
Isolate applications
APP #1
BINS / LIBS
GUEST OS
APP #2
BINS / LIBS
GUEST OS
APP #3
BINS / LIBS
GUEST OS
HYPERVISOR
HOST OPERATING SYSTEM
INFRASTRUCTURE
APP #1
BINS / LIBS
APP #2
BINS / LIBS
APP #3
BINS / LIBS
DOCKER DAEMON
HOST OPERATING SYSTEM
INFRASTRUCTURE
Virtual Machines
Docker Containers


Docker Containers Vs VMs
3
emulation
VM
Kernel
VM
Kernel
Hypervisor
Kernel
microservices
Cgroup
Namespace
Union-capabe FS
Container
processes
Pod: K8S-managed container
image
> hosts
"lighter"
OS
Linux
Kernel
host
VM's
Containers



Fork Benchmark:

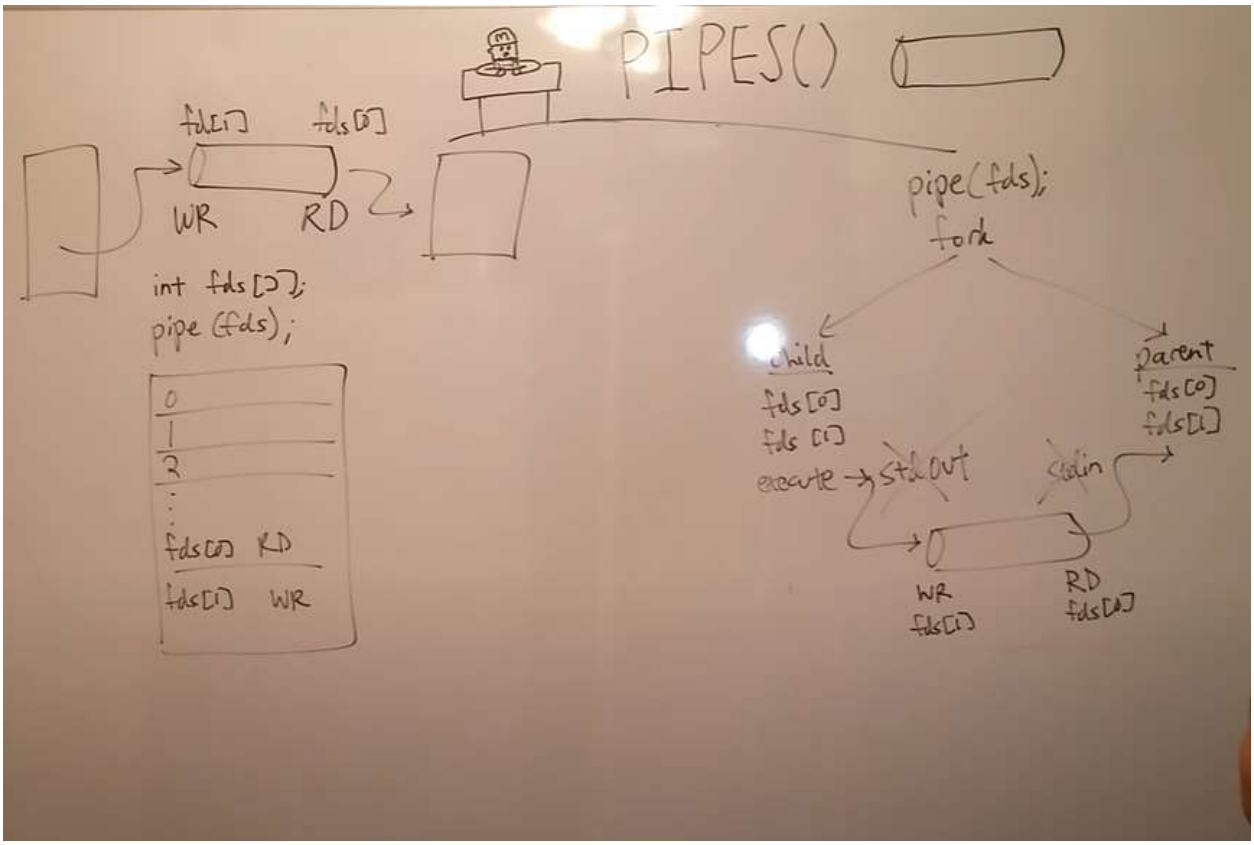
This benchmark will measure how fast new processes can be spawned. In addition to the system resources, this measures overhead introduced through system calls in the Linux kernel. The program performs a parallel computation of an integer range sum. forkbench.c receives a start and end value of an integer range and computes the sum of all numbers in that range. The range is split in two halves and two child processes are forked to compute the respective subproblem, until the recursion ends when start and end of the range are the same. In addition to the sum, the program counts the number of executed fork operations, measures the total execution time, and outputs the number of forks per second to stdout. Your finished program must use the fork and pipe system calls.

Fork() is a system process system call used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork().

- Fork() creates a copy of the calling process
- Son/child process arises as a copy of father/parent processes
- Gives the father back the PID of the son
- Gives the son back the value 0

Pipe is a form of redirection that is used to send the output of one program to another program.

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. For further details, see pipe(7).



After doing the fork benchmark test we are now running an Iperf3 to test the network speed.

Iperf3 is a tool for performing network throughput measurements. It can test either TCP or UDP throughput. To perform an iperf3 test the user must establish both a server and a client.

So we are basically comparing the loopback uplink speed of the host to the uplink speed of the virtual guests to the host?