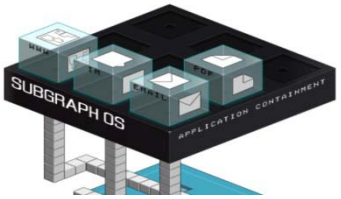
Under the Hood of Container

CoreOS

docker

rkt

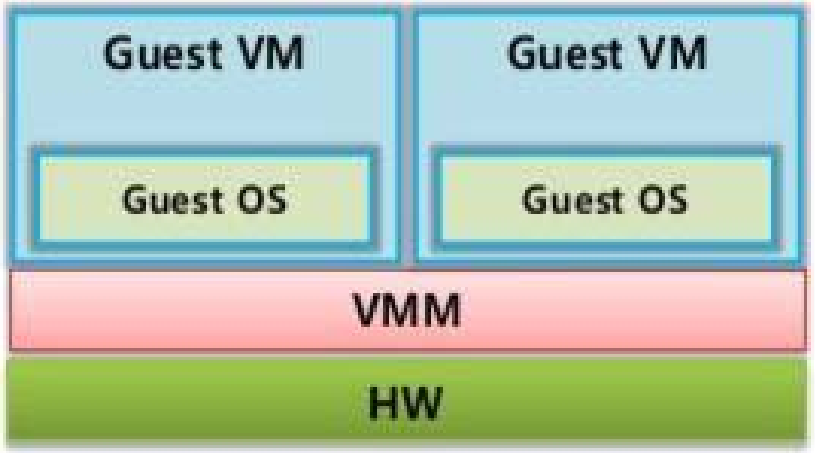


THE LXD

THE LEGION OF EXTRAORDINARY DANCERS

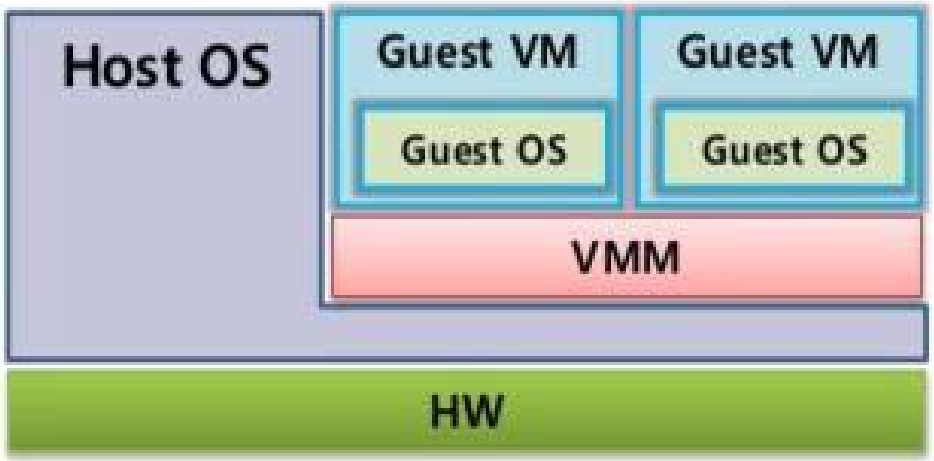
Traditional Virtualization

Type-1: VMM on HW



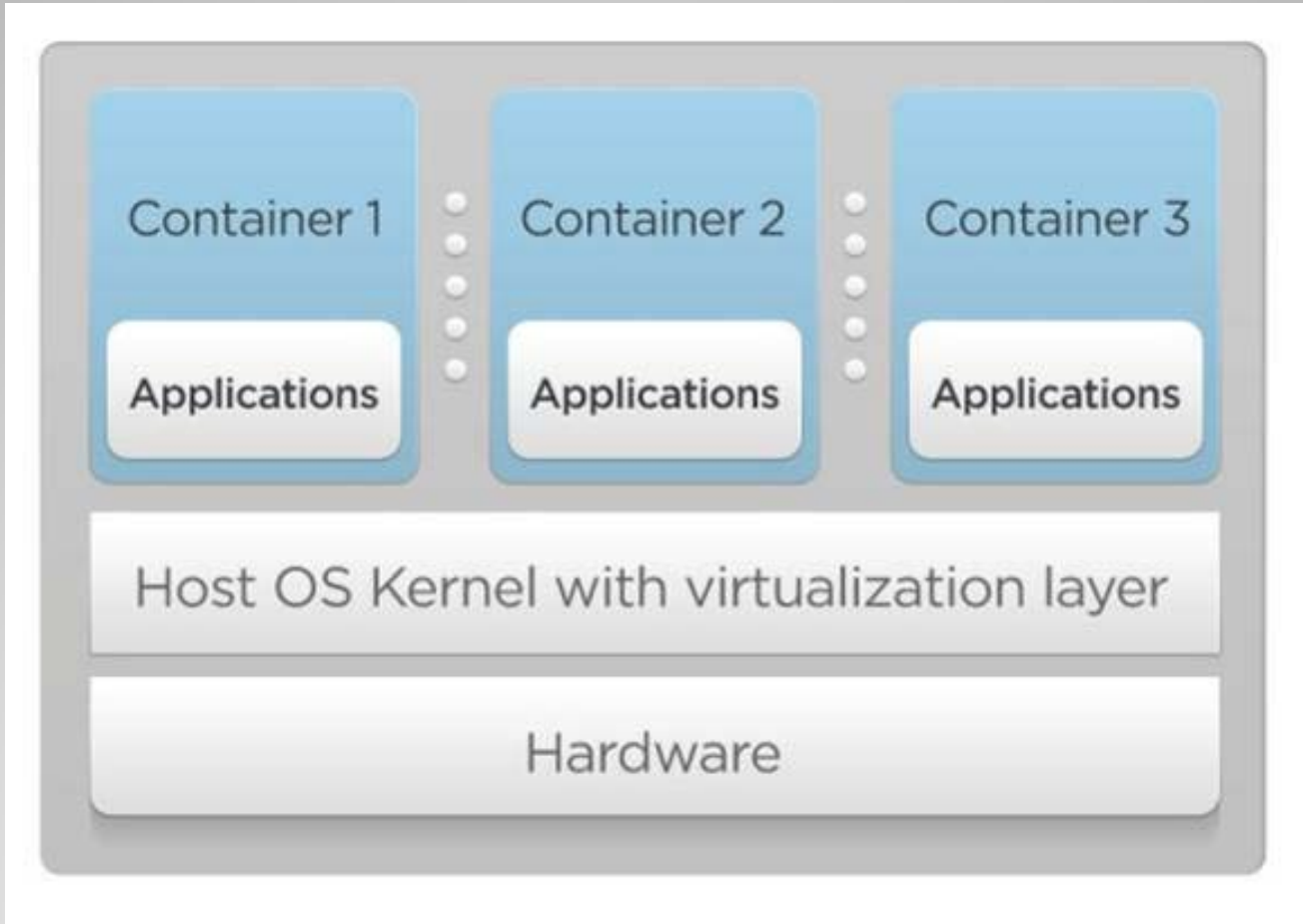
- Xen, VMware ESX server, Hyper-V
- Mostly for server, but not limited
- VMM by default
- OS-independent VMM

Type-2: Host OS on HW



- KVM, VMware Workstation, VirtualBox
- Mostly for client devices, but not limited
- VMM on demand
- OS-dependent VMM

Container Virtualization



High level View: Lightweight VM

- **I can get a shell on it**
  - through SSH or otherwise
- **It "feels" like a VM**
  - own process space
  - own network interface
  - can run stuff as root
  - can install packages
  - can run services
  - can mess up routing, iptables

Low-level View: chroot( ) on jail

- **It's not quite like a VM**
  - uses the host kernel
  - can't boot a different OS
  - can't have its own modules
  - doesn't need init as PID 1
  - doesn't need syslogd, cron...
- **It's just normal processes on the host machine**
  - contrast with VMs which are opaque

How are they implemented?
Are they in Kernel?
Lets See

- *$ find ($KERNEL_SOURCE_DIR) -type f | xargs grep '[lxc,lxd,docker]'*
  - Returns 0 results

- *$ find ($KERNEL_SOURCE_DIR) -type f | xargs grep 'container'*
  - Returns 1000+ results but all of them are irrelevant, as most of them come from "acpid container" module

Here is the Story

- Okay. There is no single technology named “Container” in Linux Kernel
- But “Container Technology” can be effectively made up with some of the independent(read orthogonal) technology from linux kernel
  - cgroup
  - namespace
  - capability
  - selinux
  - seccomp
  - apparmor
  - overlayfs
- Let me remind you. Each of them actually was invented for a different purpose

Control Groups

What do we get

- By using cgroup, system administrators gain fine-grained control over allocating, prioritizing, denying, managing, and monitoring system resources. Hardware resources can be appropriately divided up among tasks and users, increasing overall efficiency.

- Hardware resources can be appropriately divided up among tasks and users, increasing overall efficiency.

What are there in the toolbox?

- cpuset - Whole Cores And CPU Mapping
- cpuacct - CPU Cycle Accounting
- cpu - Less Than Core Granularity
- memory - Limits And Accounting
- blkio - Limits And Accounting
- net_cls - Network Classification
- net_prio - Network Priority
- checkpoint/restore - Migration

What are Play Rules?

- **Each subsystem has a hierarchy**
  - separate hierarchies for CPU, memory, block I/O…
- **Hierarchies are independent**
  - the trees for e.g. memory and CPU can be different
- **Each process is in a node in each hierarchy**
  - think of each hierarchy as a different dimension or axis
- **Each hierarchy starts with 1 node (the root)**
  - Initially, all processes start at the root node*
- **Each node = group of processes**
  - sharing the same resources

Demonstration 1

Dealing with CPU

- Keeps track of user/system CPU time
- Keeps track of usage per CPU
- Allows to set weights
- Can't set CPU limits
  - OK, let's say you give N%
  - Then the CPU throttles to a lower clock speed now what?
  - Same if you give a time slot instructions? their exec speed varies wildly

Features in “cpuset cgroup”

- Provides Physical CPU assignments & Memory limits
  - **cpuset.cpus**
    - a list of allowed CPUs
  - **cpuset.mems**
    - a list of allowed memory slots
  - **cpuset.cpu_exclusive**
    - CPUs exclusive to this group(no other group can use them)
  - **cpuset.mem_exclusive /cpuset.mem_hardwall**
    - memory slots exclusive to this group(no other group can use them)
  - **cpuset.sched_load_balance**
    - should the kernel balance the tasks between the CPUs in the current cpuset

Features in “cpuacct cgroup”

- CPU usage combined for all CPUs (in nanoseconds)
- CPU usage per-CPU (in nanoseconds)
- Per CPU and user/system(in USER_HZ)

Dealing with Memory: Accounting

- **Keeps track of pages used by each group**
  - file (read/write/mmap from block devices)
  - anonymous (stack, heap, anonymous mmap)
  - active (recently accessed)
  - inactive (candidate for eviction)
- **Each page is “charged” to a group**
- **Pages can be shared across multiple groups**
  - e.g. multiple processes reading from the same files
  - when pages are shared, only one group “pays” for a page

Dealing with Memory: Limiting

- Each group can have its own limits.
- Limits are optional
- Two kinds of limits:
  - Soft Limits
    - Soft limits are not enforced. They influence reclaim under memory pressure
  - Hard Limits
    - Hard limits will trigger a per-group OOM killer
- Limits can be set for different kinds of memory
  - Physical Memory
  - Kernel Memory
  - Total Memory

Features in “ memory cgroup”

- **Only Memory**
  - memory.usage_in_bytes - show current res_counter usage for memory
  - memory.limit_in_bytes - set/show limit of memory usage
  - memory.failcnt - show the number of memory usage hits limits
  - memory.max_usage_in_bytes - show max memory usage recorded
- **Memory + Swap**
  - memory.memsw.usage_in_bytes - show current res_counter usage
  - memory.memsw.limit_in_bytes - set/show limit
  - memory.memsw.failcnt - show the number of hits limits
  - memory.memsw.max_usage_in_bytes - show max Memory+Swap usage recorded
  - memory.soft_limit_in_bytes - set/show soft limit of memory usage
  - memory.stat - show various statistics
  - memory.use_hierarchy - set/show hierarchical account enabled
  - memory.force_empty - trigger forced move charge to parent
  - memory.pressure_level - set memory pressure notifications
  - memory.swappiness - set/show swappiness parameter of vmscan
  - memory.move_charge_at_immigrate - set/show controls of moving charges
  - memory.oom_control - set/show oom controls.
  - memory.numa_stat - show the number of memory usage per numa node

Features in “ memory cgroup”

- **Kernel Memory limits**
  - memory.kmem.limit_in_bytes - set/show hard limit for kernel memory
  - memory.kmem.usage_in_bytes - show current kernel memory allocation
  - memory.kmem.failcnt - show the number of kernel memory usage hits limits
  - memory.kmem.max_usage_in_bytes - show max kernel memory usage recorded
  - memory.kmem.tcp.limit_in_bytes - set/show hard limit for tcp buf memory
  - memory.kmem.tcp.usage_in_bytes - show current tcp buf memory allocation
  - memory.kmem.tcp.failcnt - show the number of tcp buf memory use hits limits
  - memory.kmem.tcp.max_usage_in_bytes - show max tcp buf memory usage recorded

Dealing with Block I/O

- Keeps track of I/Os for each group
  - Per Block Device
  - Read Vs. Write
  - Sync Vs. Async

- Set throttle (limits) for each group
  - Per Block Device
  - Read Vs. Write
  - Ops Vs. Bytes
- Set relative weights for each group

- Note: most writes go through the page cache. So classic writes will appear to be unthrottled at first

Features in “ blkio cgroup”

- Hope they are self explanatory
  - blkio.io_wait_time
  - blkio.io_merged
  - blkio.io_queued
  - blkio.avg_queue_size
  - blkio.group_wait_time
  - blkio.throttle.io_serviced
  - blkio.throttle.io_service_bytes
  - blkio.weight - allowed range 10 - 1000
  - blkio.weight_device - weight per device
  - blkio.leaf_weight[_device] - when competing with child cgroups
  - blkio.time - disk time allocated in miliseconds
  - blkio.throttle.read_bps_device
  - blkio.throttle.write_bps_device
  - blkio.throttle.read_iops_device

Dealing with Network

- Automatically set traffic class or priority, for traffic generated by processes in the group
- Only works for egress traffic
- net_cls will assign traffic to a class that has to be matched with tc/iptables, otherwise traffic just flows normally
- net_prio will assign traffic to a priority priorities are used by queuing disciplines

Features in “ net_* cgroup”

- Adding network class to each cgroup so you can later limit it with tc
  - Documentation/cgroups/net_cls.txt
- Prioritizing network traffic on interface
  - Documentation/cgroups/net_prio.txt

Dealing with Devices

- **Controls what the group can do on device nodes**
- **Permissions include read/write/mknod**
- **Typical use:**
  - *allow /dev/{tty,zero,random,null}*
- **A few interesting nodes:**
  - /dev/net/tun (network interface manipulation)
  - /dev/fuse (filesystems in user space)
  - /dev/kvm (VMs in containers, yay inception!)
  - /dev/dri (GPU)

Namespace

What you can see and can not see?

- **Provide processes with their own system view**
  - cgroups = limits how much you can use;
  - namespaces = limits what you can see
- **You can’t use/affect what you can’t see**
- **Available Namespaces**
  - pid
  - net
  - mnt
  - uts
  - ipc
  - user
- **Each process is in one namespace of each type**

pid namespace

- Processes within a PID namespace only see processes in the same PID namespace
- Each PID namespace has its own numbering starting at 1
- If PID 1 goes away, whole namespace is killed.
- Those namespaces can be nested. A process ends up having multiple PIDs
  - One per namespace in which its nested

net namespace

- Processes within a given network namespace get their own private network stack, including:
  - network interfaces (including lo)
  - routing tables
  - iptables rules
  - sockets
- You can move a network interface across netns

mnt namespace

- Processes can have their own root file system conceptually close to chroot
- Processes can also have “private” mounts
- Mounts can be totally private, or shared
- No easy way to pass along a mount from a namespace to another

uts namespace

- gethostname / sethostname
- That is enough.

ipc namespace

- Does anybody know about IPC?
- Does anybody care about IPC?
- Allows a process (or group of processes) to have own:
  - IPC semaphores
  - IPC message queues
  - IPC shared memory
- ... Without risk of conflict with other instances

user namespace

- Allows to map UID/GID; e.g.:
  - UID 0→1999 in container C1 is mapped to UID 10000→11999 on host;
  - UID 0→1999 in container C2 is mapped to UID 12000→13999 on host;
- UID in containers becomes irrelevant
- Just use UID 0 in the container
- It gets squashed to a non-privileged user outside
- Security improvement

Rules of Games

- Namespaces are created with the clone() syscall i.e. with extra flags when creating a new process
- Namespaces are materialized by pseudo-files
  - /proc/<pid>/ns
- When the last process of a namespace exits, it is destroyed but can be preserved by bind-mounting the pseudo-file It’s possible to “enter” a namespace with setns() exposed by the nsenter wrapper in util-linux

Demonstration 2

Welcome to the World of Containers

Chroot( ) : an useful friend

- A chroot on Unix operating systems is an operation that changes the apparent root directory for the current running process and its children. A program that is run in such a modified environment cannot name (and therefore normally cannot access) files outside the designated directory tree. The term "chroot" may refer to the chroot(2) system call or the chroot(8) wrapper program. The modified environment is called a chroot jail.

- At startup, programs expect to find scratch space, configuration files, device nodes and shared libraries at certain preset locations. For a chrooted program to successfully start, the chroot directory must be populated with a minimum set of these files. This can make chroot difficult to use as a general sandboxing mechanism.

- Only the root user can perform a chroot. This is intended to prevent users from putting a setuid program inside a specially crafted chroot jail (for example, with a fake /etc/passwd and /etc/shadow file) that would fool it into a privilege escalation.

Looking Back in Time

- BSD Jail was here. So are you.
  - https://www.freebsd.org/doc/handbook/jails.html
- Jails mainly aim at three goals:
  - **Virtualization**: Each jail is a virtual environment running on the host machine with its own files, processes, user and superuser accounts. From within a jailed process, the environment is almost indistinguishable from a real system.
  - **Security**: Each jail is sealed from the others, thus providing an additional level of security.
  - **Ease of delegation**: The limited scope of a jail allows system administrators to delegate several tasks which require superuser access without handing out complete control over the system.
- Unlike chroot jail, which restricts processes to a particular view of the file system, the FreeBSD jail mechanism restricts the activities of a process in a jail can with respect to the rest of the system. In effect, jailed processes are sandboxed. They are bound to specific IP addresses, and a jailed process cannot access divert or routing sockets.

Demonstration 3

Capability

- Capability provide fine-grained control over superuser permissions, allowing use of the root user to be avoided.
- Software developers are encouraged to replace uses of the powerful setuid attribute in a system binary with a more minimal set of capabilities.
- Many packages make use of capabilities, such as CAP_NET_RAW being used for the ping binary provided by iputils. This enables e.g. ping to be run by a normal user (as with the setuid method), while at the same time limiting the security consequences of a potential vulnerability in ping.

Capability: how do you implement it

- A process has three sets of bitmaps called the
  - Inheritable(i)
  - Permitted(p),
  - Effective(e)
- Each capability is implemented as a bit in each of these bitmaps which is either set or unset.
- When a process tries to do a privileged operation, the operating system will check the appropriate bit in the effective set of the process (instead of checking whether the effective uid of the process i 0 as is normally done).
  - For example, when a process tries to set the clock, the Linux kernel will check that the process has the CAP_SYS_TIME bit (which is currently bit 25) set in its effective set.

How does it play with others?

- Files have capabilities. Conceptually they have the same three bitmaps that processes have, but to avoid confusion we call them by other names. The three sets are called the allowed set, the forced set, and the effective set.
- Only executable files have capabilities, libraries don't have capabilities (yet).
- Users and Groups don't have associated capabilities from the kernel's point of view, but it is entirely reasonable to associate users or groups with capabilities. By letting the "login" program set some capabilities it is possible to make role users such as a backup user that will have the CAP_DAC_READ_SEARCH capability and be able to do backups. This could also be implemented as a PAM module, but nobody has implemented one yet.

Capability Lifecycle

- The capability rules are the rules used to set the capabilities of the new process image after an exec. They work like this:

  pI' = pI
  pP' = fP | (fI & pI)
  pE' = pP' & fE [NB. fE is 0 or ~0]

  I=Inheritable, P=Permitted, E=Effective // p=process, f=file
  'indicates post-exec().

- Now to make sense of the equations think of fP as the Forced set of the executable, and fI as the Allowed set of the executable. Notice how the Inheritable set isn't touched at all during exec().

- Bits can be transferred from Permitted to either Effective or Inheritable set.

- Bits can be removed from all sets.

Three Pillars of Protection

- Minimum Privilege
- Inherited Privilege
- Bounded Privilege

Demonstration 4

seccomp

- Seccomp filtering provides a means for a process to specify a filter for incoming system calls. The filter is expressed as a Berkeley Packet Filter (BPF) program, as with socket filters, except that the data operated on is related to the system call being made: system call number and the system call arguments. This allows for expressive filtering of system calls using a filter program language with a long history of being exposed to user land and a straightforward data set.

- Additionally, BPF makes it impossible for users of seccomp to fall prey to time-of-check-time-of-use (TOCTOU) attacks that are common in system call interposition frameworks. BPF programs may not dereference pointers which constrains all filters to solely evaluating the system call arguments directly. System call filtering isn't a sandbox. It provides a clearly defined mechanism for minimizing the exposed kernel surface. It is meant to be a tool for sandbox developers to use.

Seccomp: how do we use it?

- http://man7.org/linux/man-pages/man2/seccomp.2.html

Demonstration 5

SElinux

- Security-Enhanced Linux (SELinux) is a Linux kernel security module that provides a mechanism for supporting access control security policies, including United States Department of Defense–style mandatory access controls (MAC).

- SELinux is a set of kernel modifications and user-space tools that have been added to various Linux distributions. Its architecture strives to separate enforcement of security decisions from the security policy itself and streamlines the volume of software charged with security policy enforcement.[2][3] The key concepts underlying SELinux can be traced to several earlier projects by the United States National Security Agency (NSA).

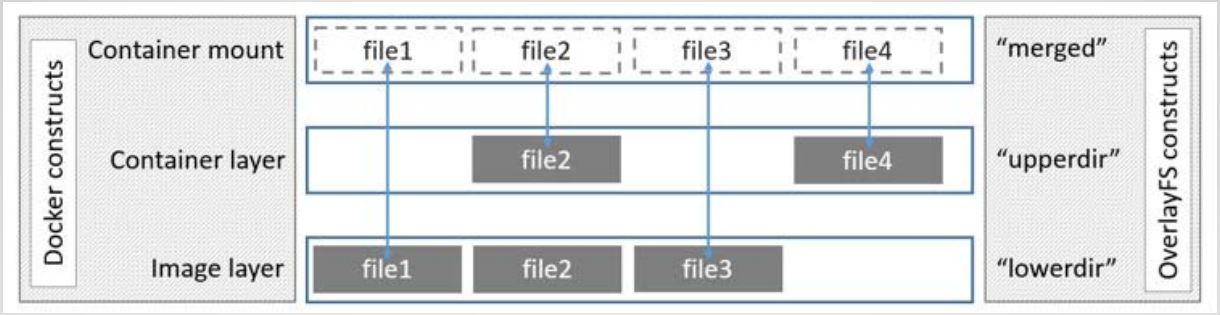
Lemme introduce myself

- Adding security context
  - Labelling of <User:Object:Action>
- Exposed via -Z flag of relevant command
- Selinux mode
  - Enforced
  - Permissive
  - Disabled

Demonstration 6

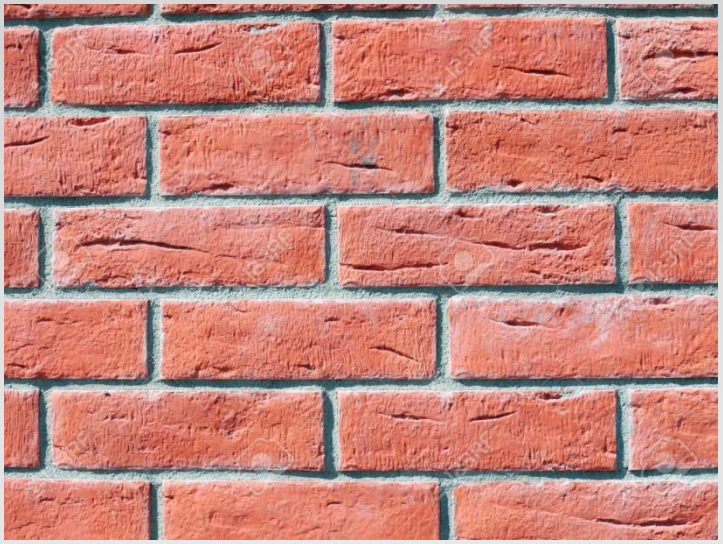
Overlay File System

- An overlay file system combines two file systems - an 'upper' file system and a 'lower' file system. When a name exists in both file systems, the object in the 'upper' file system is visible while the object in the 'lower' file system is either hidden or, in the case of directories, merged with the 'upper' object.



Demonstration 7

Let everything put together



Demonstration 8