Imperial College London

LSDS
Large-Scale Data & Systems Group

Containers/Serverless

Jana Giceva
jgiceva@doc.ic.ac.uk

Department of Computing
Imperial College London
http://lsds.doc.ic.ac.uk

Fall 2018

Virtualization Revision

Revision exercise

There are three VMs running on a four core machine. Assume we have a spare core on each machine. Assume that there is an Open vSwitch (OVS) network configured that contains all the three VMVs, which can also use DPDK to accelerate performance.

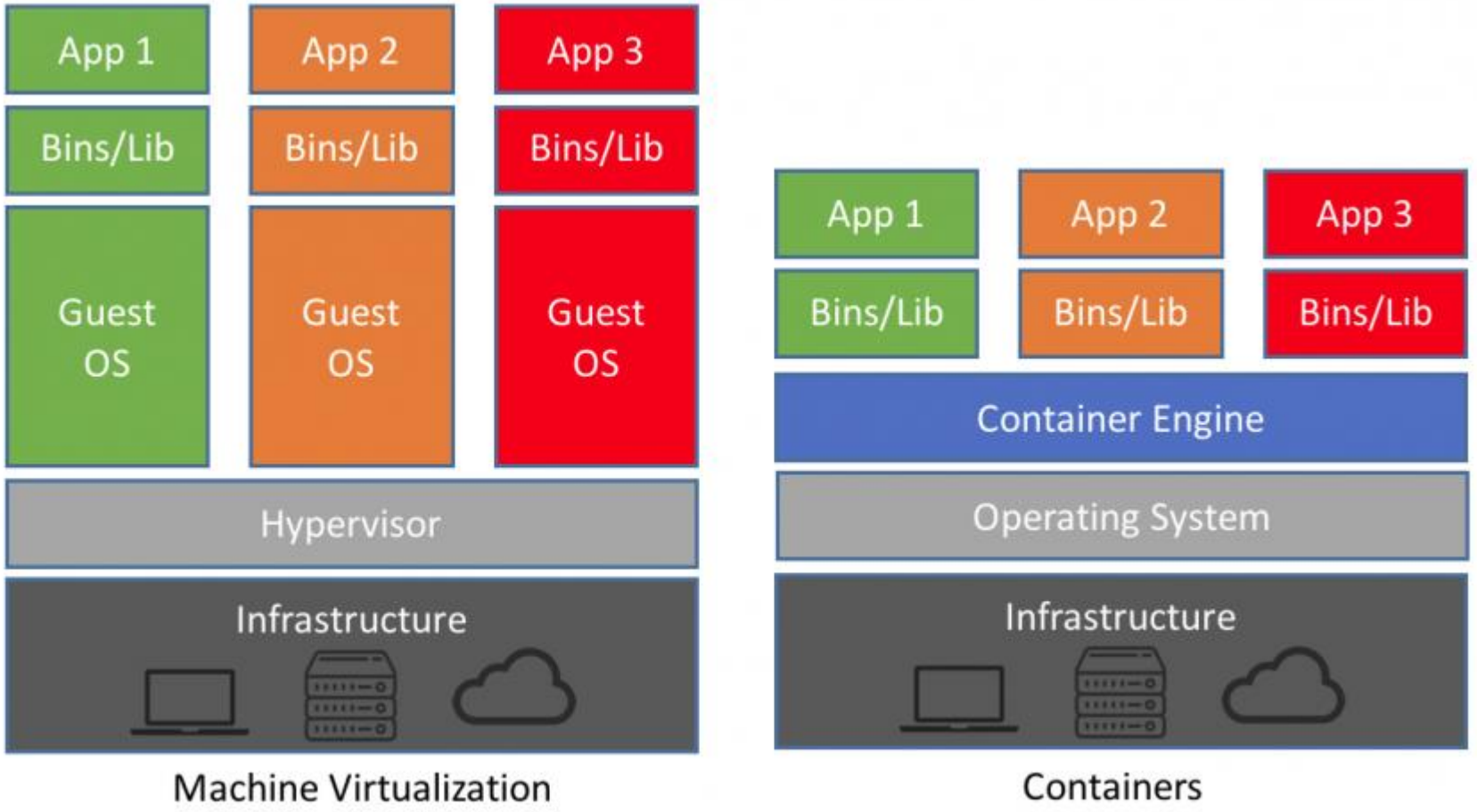
1. How does the VM execute safely at high performance?
2. How can the VM safely access memory?
3. How does the VM safely access DMA memory?
4. How can the application on the VM safely and efficiently access the network?
5. What happens when a packet arrives?

Containers – OS virtualization

What's one problem with VMs?

Based on the reading for today "Container-based OS virtualization: a scalable, high-performance alternative to hypervisors".

Architecturally: VMs vs. Containers



src: https://blog.netapp.com/blogs/containers-vs-vms/

VMs vs. Containers

Virtual Machines (VMs)

- Virtualize the hardware
- Heavy-weight in terms of layers of system software (GBs in size)
- Take minutes to start
- Allows multiple OS to execute concurrently
- Provide high level of isolation:
  - Fault
  - Resource
  - …

Containers

- Virtualize the OS, by creating "protected" portions of the OS
- Light-weight (MBs in size)
- Take seconds to start
- Share a common OS and kernel
- Don't offer the same level of isolation:
  - A kernel crash caused by one guest, will effect everyone else.
- Interface offered is at the level of system calls and ABI – much more prone for security problems that the API exposed by the Hypervisor

Linux Containers

LXC – userspace interface for the Linux containment features.

Something in the middle between a chroot and a full fledged virtual machine.

The goal is to create an environment as close as possible to standard Linux installation, without the need for a separate kernel.

Features:

- **Kernel namespaces** (ipc, uts, mount, pid, network and user)
- **Control groups** (cgroups)
- **Chroot** (using pivot_root)
- Kernel capabilities
- AppArmor and SELinux profiles
- Seccomp policies

Core technology behind Linux Containers (LXC)

Kernel namespaces

- Responsible for isolation
- Separates the container's root user from the host's user database.

Control groups (cgroups)

- Resource accounting and limiting
- For both physical and logical resources

Copy-on-write file system

- Using AuFS (Advanced Multi-Layer Unification Filesystem) as a file system for containers
- Layered FS that can transparently overlay one or more existing filesystems

Traditional Isolation Mechanisms

Process isolation

- Each process with its own virtual address space

Each process has an owner (user UID)

- Used for most access checks

UID passed onto objects such as files of owner

- Determine which system resources a user can access

Fine grained Access Control Lists (ACLs)

- List of permissions attached to an object, specifying which users have access to objects
- Also specifying which operations are allowed on given objects
- Handy, but increases complexity

chroot

What does chroot do in a Unix OS?

- Provides the illusion that a given directory is the root of the file system
- The modified environment is called chroot jail.

When a process runs in chroot, it cannot access any files outside the current "root".

However, there are limitations:

- A chrooted process can mount a device and have access to the files on the mounted file system
- A chrooted process with sufficient privileges can "break out" using a second chroot
- Does not restrict the use of resources like I/O, bandwidth, disk space or CPU time

Understanding chroot

Normally a process on start-up expects to find /bin, /usr/bin, etc. populated

- This is not the case with chroot
- Makes it hard to use as a general sandboxing mechanism
- You need to do a "bind mount" and basically show /usr/lib as /chroot/usr/libb

How does chroot work?

- It changes path look up: lookup up /a is translated to chroot-home/a where chroot-home is where the chroot was started

How to check if a process is chrooted or not?

- See its root directory at /proc/pid/root. For chrooted processes, it will point to the new root

So what is the problem with chroot?

- You need all the dependencies of an application copied into the chroot jail
- For large application, it is difficult to figure out all the dependencies
- Chroot isolation is not perfect – isolates just the file system.
- Not the system resources, system users, running processes and the network subsystem which are still shared between the chrooted and host processes.

FreeBSD Jail

FreeBSD jails (builds upon the chroot concept, but) virtualize access to the file system, the users, and the networking file system

A jail is characterized by four elements:

1. A directory subtree: the starting point from which a jail is entered. Once inside the jail, the process is not permitted to escape outside of this subtree.

2. A hostname: which will be used by the jail.

3. An IP address: which is assigned to the jail. Often an alias address for an existing network interface

4. A command: the path name of an executable to run inside the jail. Path is relative to the root directory of the jail environment

Jails have their own set of users and their own root account, which are limited to the jail environment.

FreeBSD jail cont.

When a process is placed in jail

- All its descendants will also be in jail
- Can only communicate with other processes inside the same jail
- Cannot access any privileged resources (non-root)
- File system is limited similar to chroot
- Can only use a single IP address (both for sending and receiving

Restrictions apply even if it is running with root privileges:

- Cannot load kernel modules
- Cannot change network configuration
- Cannot create devices, mount file systems, etc.

Processes outside the jail

- Can see processes inside the jail
- Can directly modify files inside the jail

Solaris has something similar called Solaris Zones

Kernel Namespaces

Kernel namespaces – lightweight way to virtualize a process

- Isolation – enables a process to have different views of the system than other processes
- Split kernel resources into one instance per namespace
- This partitions processes, users, network stacks and other components into separate analogous pieces in order to provide processes a unique view
- Originated in 1992 in the "The use of Name Spaces in Plan 9" operating system paper
- Much like Solaris Zones
- No hypervisor layer

Example, one bash shell can have a different hostname than a different bash shell if they are running in different namespaces.

Kernel namespaces – cont.

Three system calls are used for namespaces:

- clone() – creates a ***new process*** and a ***new namespace***
  - The process is attached to the new namespace
  - Process creation and termination methods, fork() and exit() methods, were patched to handle the new namespace CLONE_NEW* flags
- unshare() – does not create a new process; creates a ***new namespace;*** attaches the current process to it.
- setns() – a new system call was added for joining an existing namespace

/proc/pid/ns lists the namespace number for each namespace for each process

- ls -al /proc//ns
- lrwxrwxrwx 1 root root 0 Apr 24 17:29 ipc -> ipc:[4026531839]
- lrwxrwxrwx 1 root root 0 Apr 24 17:29 mnt -> mnt:[4026531840]
- lrwxrwxrwx 1 root root 0 Apr 24 17:29 net -> net:[4026531956]
- lrwxrwxrwx 1 root root 0 Apr 24 17:29 pid -> pid:[4026531836]

Kernel namespaces – cont.

There are currently 6 namespaces:

1. mnt (mount points, filesystems)
2. pid (processes)
3. net (network stack)
4. ipc (system V IPC)
5. uts (hostname)
6. user (UIDs)

Kernel namespaces – what do they isolate?

1. Mount namespaces

- Isolate the set of the file system mount mounts as seen by a group of processes.
- As a result, processes that live in different namespaces can have different vies of the FS hierarchy.
- `mount()` and `unmount()` sys calls no longer operate on global set of mount points – but with the mount namespace associated with the calling process.
- Similar to `chroot` jails, but more secure and more flexible.
- First namespace to be implemented in Linux in 2002.

2. UTS namespaces

- Isolate two system identifiers: `nodename` and `domainname` returned by `uname()` sys call.
- Allows each container to have its own hostname and NIS domain name.

3. IPC namespaces

- Isolate certain InterProcess Communication (IPC) resources – namely, the System V IPC objects and POSIX message queues.

Kernel namespaces – what do they isolate? II

4. PIC namespaces

- Isolate the process ID number space.
- Processes in different PID namespaces can have the same PID → making container migration between hosts easy, as one can maintain the PID.
- Allow each container to have its own `init` (PID 1) – the ancestor of all processes that manages various system initialization tasks and reaps the orphaned child processes when they terminate.
- A process can see only processes contained in its own PID namespace and the ones nested below that PID namespace.

5. Network namespaces

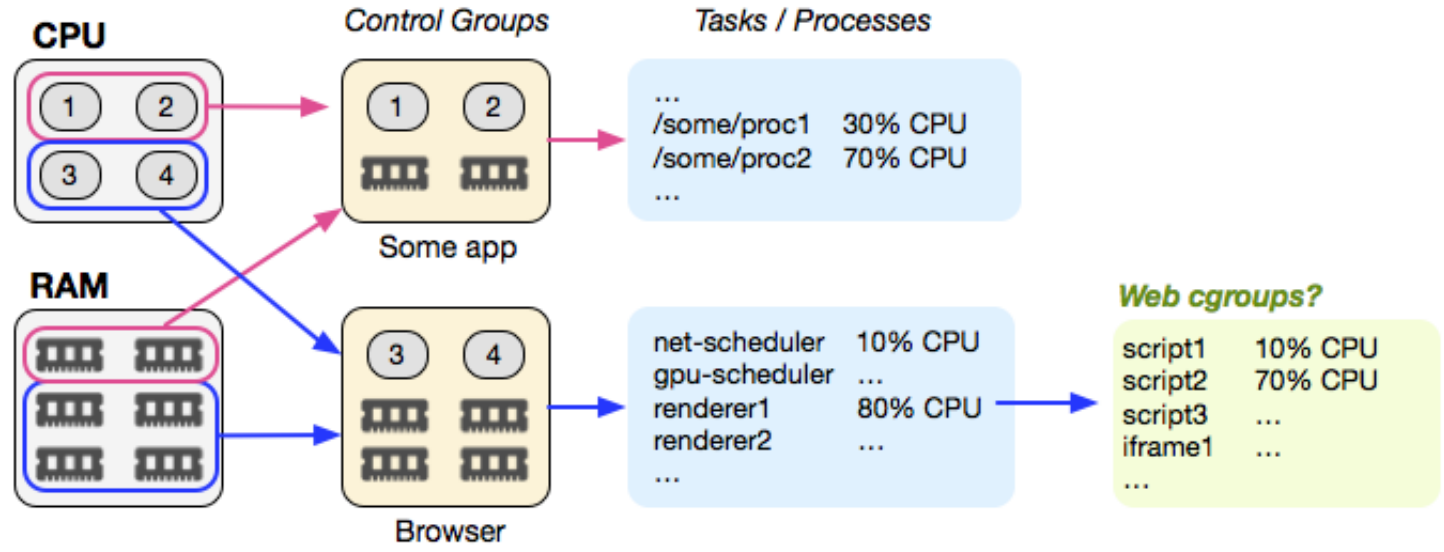
- Isolation of system resources associated with networking.
- Each network namespace has its own network device, IP addresses, IP routing tables, `/proc/net` directory, port numbers, etc.
- So each container can have its own (virtual) network device

6. User namespaces

- Isolate the user and group ID number spaces.
- Process's user and group ID can be different inside and outside a user namespace with different privileges.

Control Groups (cgroups)

Implement resource accounting and limiting among a hierarchy of user-defined groups of tasks (processes).



src: https://www.igvita.com/2016/03/01/control-groups-cgroups-for-the-web/

Ensure that a user-defined group of tasks will gets its fair share of the critical system resources (e.g., memory, CPU, disk I/O, network bandwidth, etc.).

Control groups – organization

Definitions:

- A **cgroup –** set of tasks with a set of parameters for one or more subsystems.
- A **subsystem** – represents a single resource, such as CPU time or memory.
- A **hierarchy** – set of cgroups arranged in a tree, such that every task in the system is in exactly one of the cgroups in the hierarchy, and a set of subsystems.

The cgroup model:

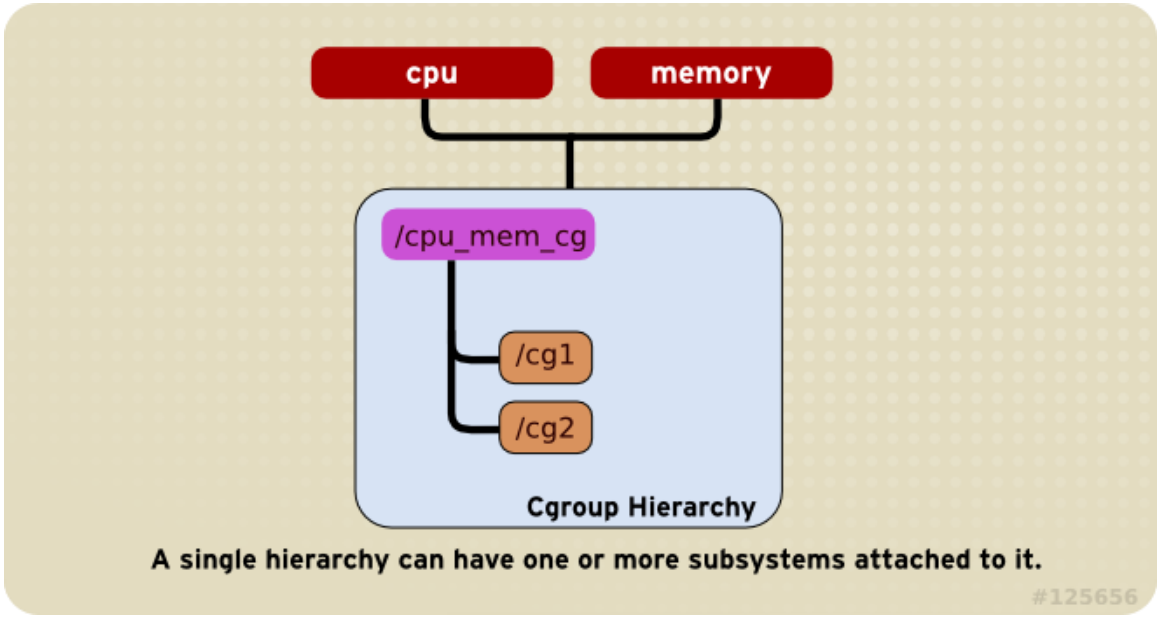
- Similar to processes, cgroups are hierarchical.
- Similar to processes, child cgroups will inherit certain attributed from their parent cgroup.
- But, many separate hierarchies of cgroups can exist simultaneously
- Each hierarchy can be attached to one or more subsystems.

How it cgroups&subsystems together?

System processes are called tasks in cgroup terminology.

Few rules on the relationship between subsystems, cgroup hierarchies, and tasks.

Rule 1: A single hierarchy can have one or more subsystem attached to it.

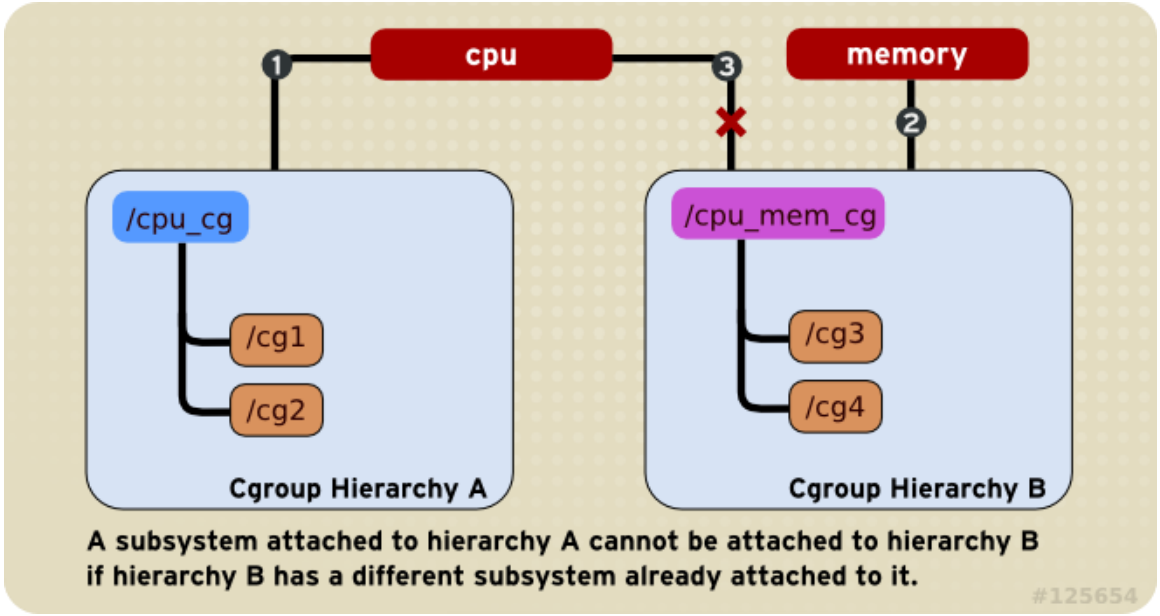


src: [RedHat Cgroups documentation]

How it cgroups&subsystems together?

Few rules on the relationship between subsystems, cgroup hierarchies, and tasks.

Rule 2: Any single subsystem cannot be attached to more than one hierarchy if one of those hierarchies has a different subsystem attached to it already.

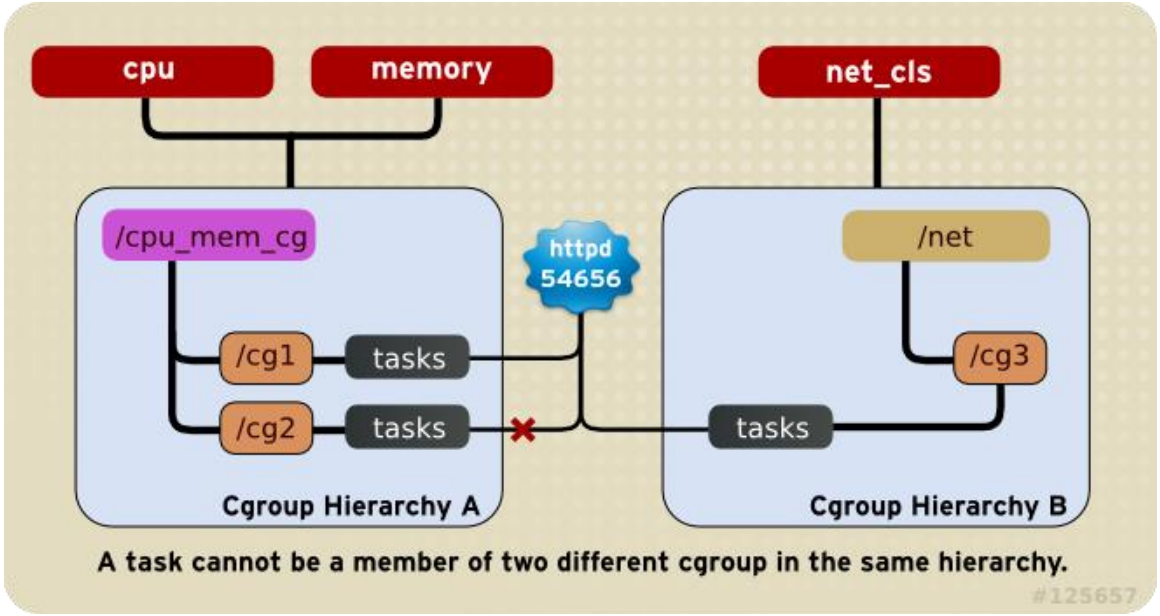


src: [RedHat Cgroups documentation]

How it cgroups&subsystems together?

Few rules on the relationship between subsystems, cgroup hierarchies, and tasks.

Rule 3: For any single hierarchy, each task on the system can be a member of *exactly one* cgroup in that hierarchy. A task can be in multiple cgroups as long as they are in different hierarchies.

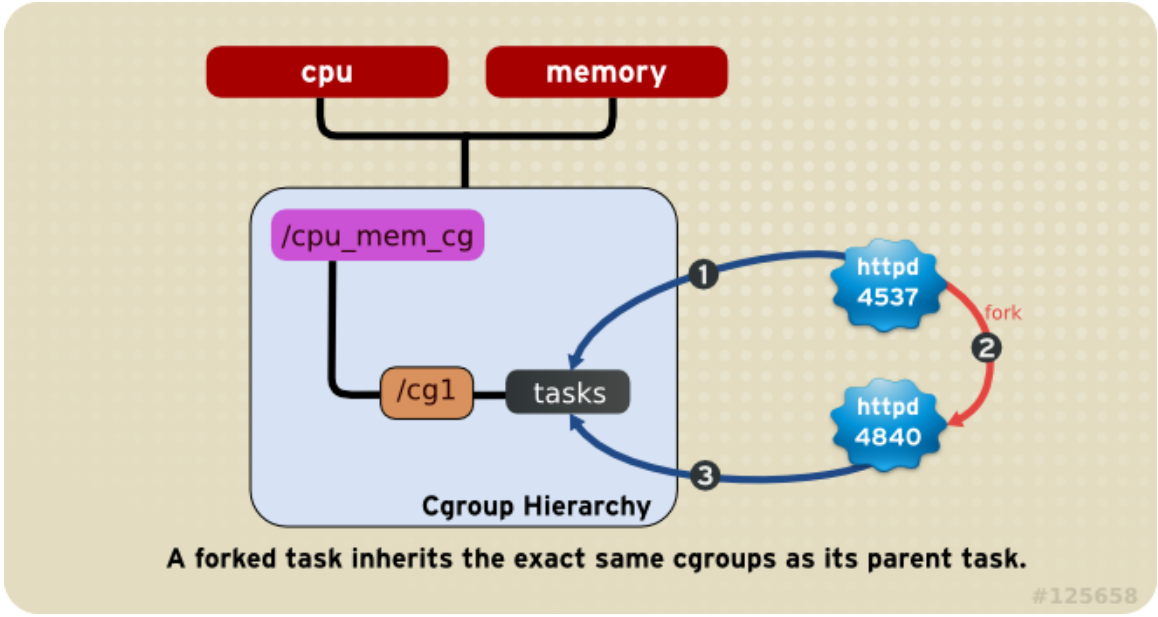


src: [RedHat Cgroups documentation]

How it cgroups&subsystems together?

Few rules on the relationship between subsystems, cgroup hierarchies, and tasks.

Rule 4: Any child task automatically inherits the cgroup of its parent, but can be moved to different cgroups if needed. Once forked, parent and child are independent



src: [RedHat Cgroups documentation]

Cgroups implementation

Cgroups uses a Virtual File System

- All entries are created in it
- All actions are performed via file system actions (create/remove directory, read/write files, etc.)
- Mounting a new point in the cgroup FS creates a "hierarchy"

How are cgroups implemented? Changes to:

- In boot phase (`/init/main.c`) to perform various initializations.
- In process creation and destroy methods, `fork()` and `exit()`.
- A new file system of type "cgroup" (VFS)
- Process descriptor additions (`struct task_struct`)
- Add procfs entries:
  - For each process: /proc/pid/cgroup
  - System-wide: /proc/cgroups

Cgroups subsystems

Cgroup subsystems:

- cpuset, cpu, cpuacct, memory, devices, freezer, net_cls, net_prio, blkio, ns, perf_event

1. cpuset

- Provide a mechanism for assigning a set of CPUs and memory nodes to a set of tasks

2. cpu

- Schedules CPU access to cgroups: choice between CFS and RT + other config parameters

3. cpuacct

- The CPU accounting subsystem generates automatic reports on CPU resources used by the tasks in a cgroup (including tasks in child groups).

4. devices

- Allows or denies access to devices by tasks in a group.

Cgroups subsystems cont.

Cgroup subsystems:

- cpuset, cpu, cpuacct, memory, devices, freezer, net_cls, net_prio, blkio, ns, perf_event

5. freezer

- Suspends or resumes tasks in a cgroup.

6. memory

- Generates automatic reports on memory resources used by tasks in a cgroup and sets limits on memory use of those tasks.

7. net_cls

- Tags network packets with a class identifier (classid) that allows the Linux traffic controller (tc) to identify packets originating from a particular cgroup. Can then assign different priorities.

8. net_prio

- Dynamically set the priority of traffic per each network interface for applications in diff. cgroups.

Cgroups subsystems cont.

Cgroup subsystems:

- cpuset, cpu, cpuacct, memory, devices, freezer, net_cls, net_prio, blkio, ns, perf_event

9. blkio

- Controls and monitors access to I/O on block devices by tasks in cgroups. Can limit access or bandwidth and provide information on I/O usage.

10. ns

- Provides way to group processes into separate namespaces. → similar to kernel namespaces

11. perf_event

- When this subsystem is attached to a hierarchy, 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/thread separately.

Popular container – Docker

Goals:

1. To manage **dependency hell**
2. To run an application **in isolation**

Built on top of:

1. Kernel namespaces
2. Control groups
3. UnionFS
4. Capabilities

UnionFS:

- Copy on write file system that is the union of all existing file systems.
- Allows several containers to share common data
- On write to UnionFS, the overwritten data is saved to a new path, specific to that container → do not affect the reads from other containers.

Docker

Goals:

1. To manage **dependency hell**
2. To run an application **in isolation**

Built on top of:

1. Kernel namespaces
2. Control groups
3. UnionFS
4. Capabilities

Root Capabilities:

- More fine-grained control than just root or non-root
- A process can be non-root, but given root-like privileges for a specific task (e.g., using a network port).
- A process can be root, and be denied access because of missing capabilities (e.g., no permission to mount new devices).
- Container root user can be significantly less powerful than the host root.

Docker’s components

Each container gets its own namespace(s) and cgroup(s)

Namespaces isolate containers from each other

- One container can not even see the list of processes in another container

Cgroups allow the admin to isolate resources used by each container and its children

Provides a whitelist of capabilities to root users inside a container

Running the docker daemon requires root privileges

State-of-the-art

LightVM:

- Goal: make VMs light and as fast as containers, while providing stronger isolation
- Fast instantiation, high instance density, quick pause/unpause, small memory and on-disk footprint
- Based on Xen – modifies Xen's control plane
- Lightweight too stack
- LightVM can boot a VM in 2.3ms, comparable to fork/exec on Linux (1ms) and two order to magnitude faster than Docker.

Unikernel:

- Built on observation "most containers and VMs run a single application".
- Idea: only include in VM what is necessary for that (single-address-space) application.
- Customized OS for a particular application or set of applications – only the code that is needed.
- Tinyx – tool that generates a unikernel for a given application (based on Mirage and ClickOS).
- Lighter weight, but lose the ability to run other applications.

Serverless

Motivating example

Let us assume you have a function, which talks to a server-side database, that is triggered every time a user visits a web-page.

The normal way to do this is to (1) run a web-server on the server side, (2) use a framework like Ruby on Rails to implement your logic.

- This requires that you have a physical machine (or an EC2 instance) to run your web-server
- You need to run the whole OS + web server stack on the machine
- If the traffic increases, you may need another machine

Problems:

- Servers are underutilized
- It is difficult to scale-out
- Managing an entire OS for a service is not always easy

Can containers help?

Yes, and no!

Containers help with this problem by only requiring the developer to create containers instead of full VMs.

The other problems remain:

1. You need to run and pay for a cluster to run your containers on.

2. If all you wanted was to run a single function in response to a user request, it seems like an overkill!

Serverless

Serverless is the next stop along the road to make all of this more efficient and convenient for cloud users.

Functions (FaaS) run in *compute containers:*

- Stateless
- Event-triggered
- Ephemeral
- Fully managed by a third party

With serverless:

- Focus on the application logic
- Zero administration
- Auto-scaling
- Increased velocity
- Pay-per-use

How does it compare to the other solutions?

1. Traditional stack
   - you need to manage the servers + server applications

2. Virtual machines
   - manage VMs + server applications + physical/virtual cluster (optional)

3. Containers
   - manage containers + server applications (inside containers) + physical/virtual cluster (optional)

4. Serverless
   - manage functions (nothing else, not even server applications)

Serverless in the Cloud

Serverless (FaaS) is provided by all major cloud providers:

Amazon Web Services, Microsoft Azure, IBM OpenWhisk, Google Cloud Platform

Amazon Lambda

Azure Functions

IBM OpenWhisk

Google Cloud Functions

Amazon's Serverless is called Lambda

- Amazon free tier provides 1 million lambda requests per month!
- Function start-up times on AWS Lambda: 10-100ms for a python or JavaScript function, >100ms (sometimes a few seconds) for a Java function (need to start the JVM)

Restrictions with Serverless Paradigm

Can’t have (machine/instance-bound) state.

- No data stored in memory or local disk.
- If needed, store it externally.

Execution duration timeout.

- FaaS functions are often limited in how long an invocation is allowed to run.
- AWS Lambda function execution terminates after 5 min.

Startup latency and “cold starts”.

- Start-up latency for a function depends on various factors (few milliseconds to several seconds).
- Warm start is possible if we reuse an instance of a FaaS on the host container from a prev. event.
- Frequency of “cold starts” can often be a limiting factor.

Serverless platforms

IBM was first to open-source FaaS – OpenWhisk, now an Apache project.

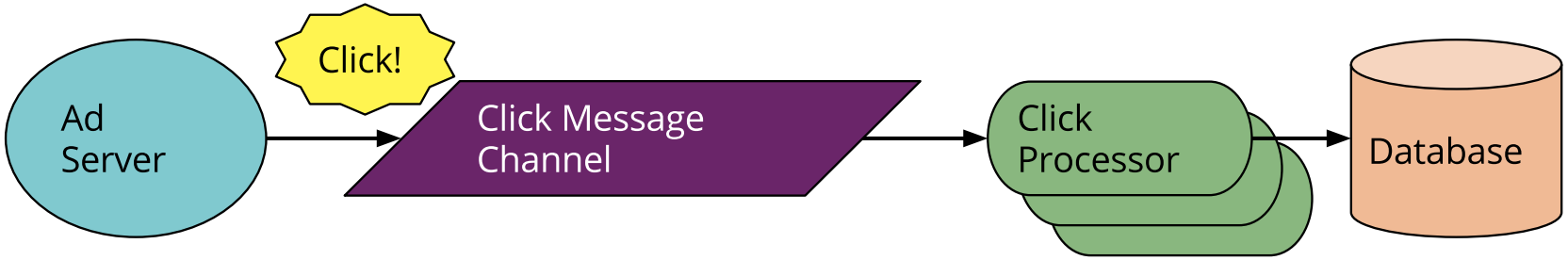
Microsoft also open sources their Azure Functions platform.

Other self-hosted FaaS implementations make us of an underlying container platform (e.g., Kubernetes).

Some notable projects: Galactic Fog, Fission, OpenFaaS, etc.

Good examples of Serverless functions

1. Auto-scaling Websites and APIs
2. UI-driven applications (e.g., image and video manipulation)
3. Message-driven applications



4. Query as a Service (or Stored Procedure as a Service)
5. Event streaming and event-driven apps

Need to change the application – make it "think" in terms of *microservices.*

Pros and Cons of using Serverless

+ Reduced operational cost
  - Outsourcing solution to the cloud vendor
+ Reduced development cost
+ FaaS: scaling costs
  - Horizontal scaling is automatic, elastic and managed by the provider.
+ Pay for what you use
  - Excellent for infrequent requests
  - Inconsistent (spiky) traffic
  - Optimizations pay off literally
+ Easier operational management
  - Reduced packaging and deployment complexity
+ Greener computing

- Vendor control and lock-in
  - Losing control over a lot of knobs
  - Cross cloud portability can be difficult
- Security concerns
  - Requires that you trust the provider even more
- Loss of server optimizations
  - Can't assume anything about the hardware
- No in-server state for FaaS
- Execution duration cutoff
- Start-up latency
- Debugging, Testing, Monitoring

References

Containers:

- *"Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors"* (EuroSys'07)
- https://linuxcontainers.org
- Detailed slides for namespaces and cgroups: http://www.haifux.org/lectures/299/netLec7.pdf
- https://www.freebsd.org/doc/handbook/jails.html
- https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment
- https://lwn.net/Articles/531114/
- https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01
- Initial version of cgroups: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt
- https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/cgroup-v2.rst

Serverless:

- https://serverless.com/framework/docs/
- https://martinfowler.com/articles/serverless.html
- Cloud providers documentation files