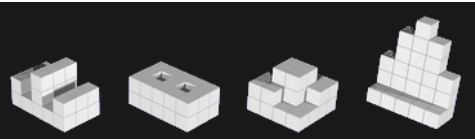
Linux Containers

Basic Concepts



Lucian Carata

FRESCO Talklet, 3 Oct 2014

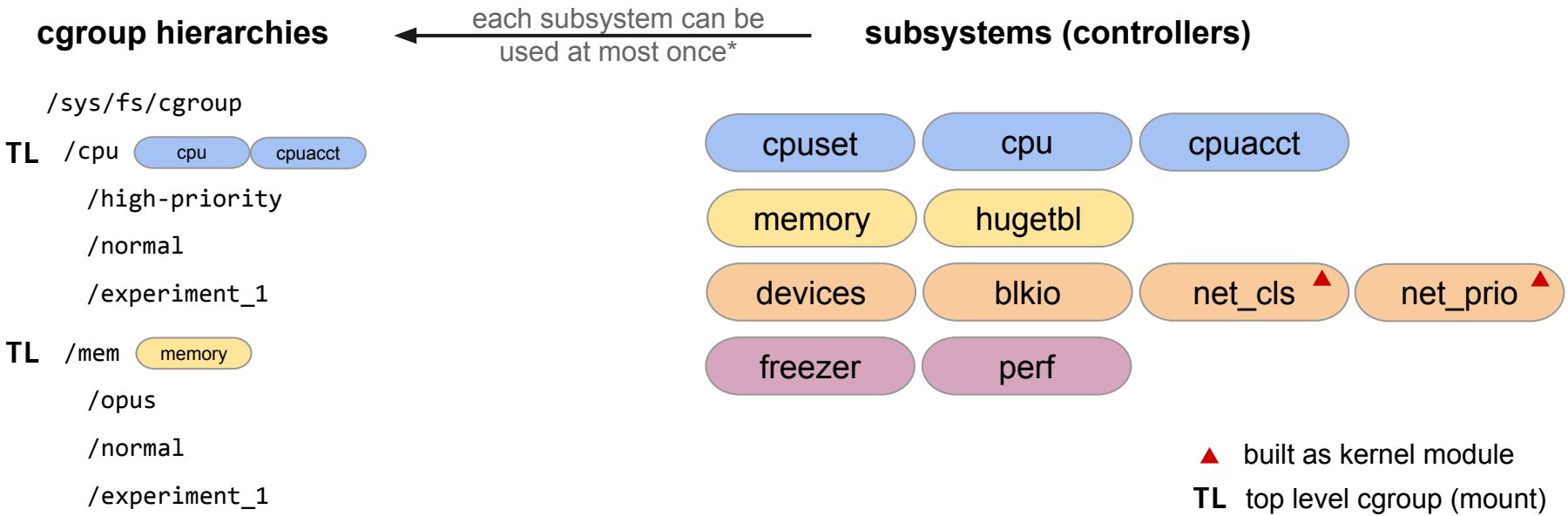
Underlying kernel mechanisms

cgroups	manage resources for groups of processes
namespaces	per process resource isolation
seccomp	limit available system calls
capabilities	limit available privileges
CRIU	checkpoint/restore (with kernel support)

cgroups - user space view

low-level filesystem interface similar to sysfs (/sys) and procfs (/proc)

new filesystem type “cgroup”, default location in /sys/fs/cgroup



cgroups - user space view

cgroup hierarchies

```
/sys/fs/cgroup
TL  /cpu  [cpu] [cpuacct]
      /high-priority
      /normal
      /experiment_1
TL  /mem  [memory]
      /opus
      /normal
      /experiment_1
```

common

tasks
cgroup.procs
release_agent TL
notify_on_release
cgroup.clone_children
cgroup.sane_behavior

cpuacct

cpuacct.stat
cpuacct.usage
cpuacct.usage_percpu

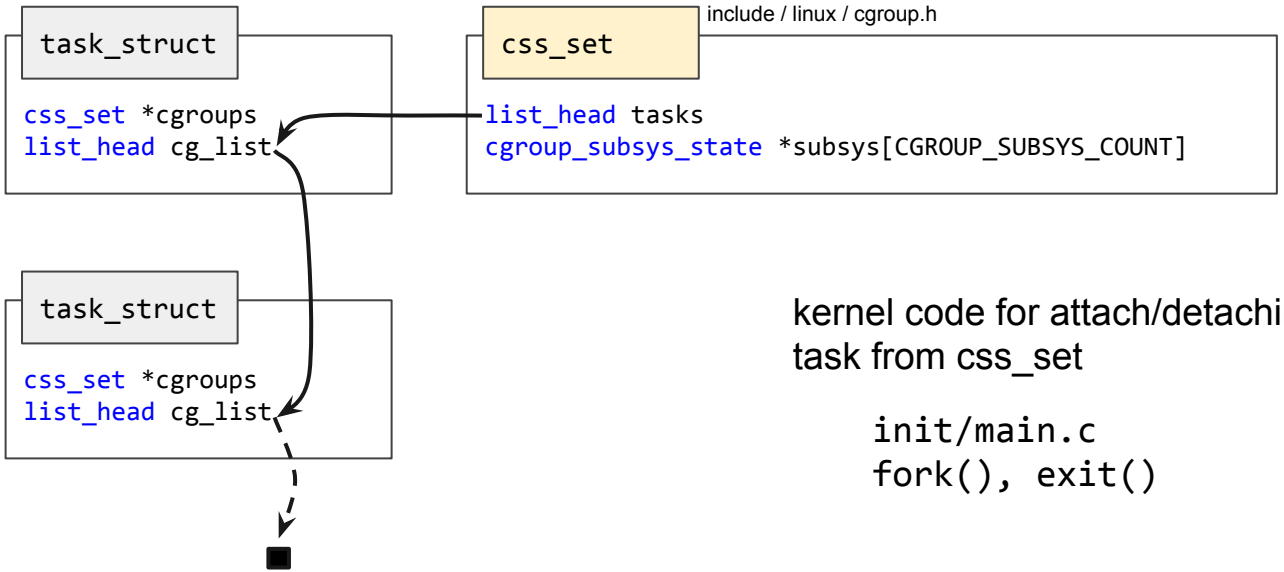
cpu

cpu.stat
cpu.shares
cpu.cfs_period_us
cpu.cfs_quota_us
cpu.rt_period_us
cpu.rt_runtime_us

cpuset | memory | hugetbl | devices | blkio

net_cls ▲ | net_prio ▲ | freezer | perf

cgroups - kernel space view

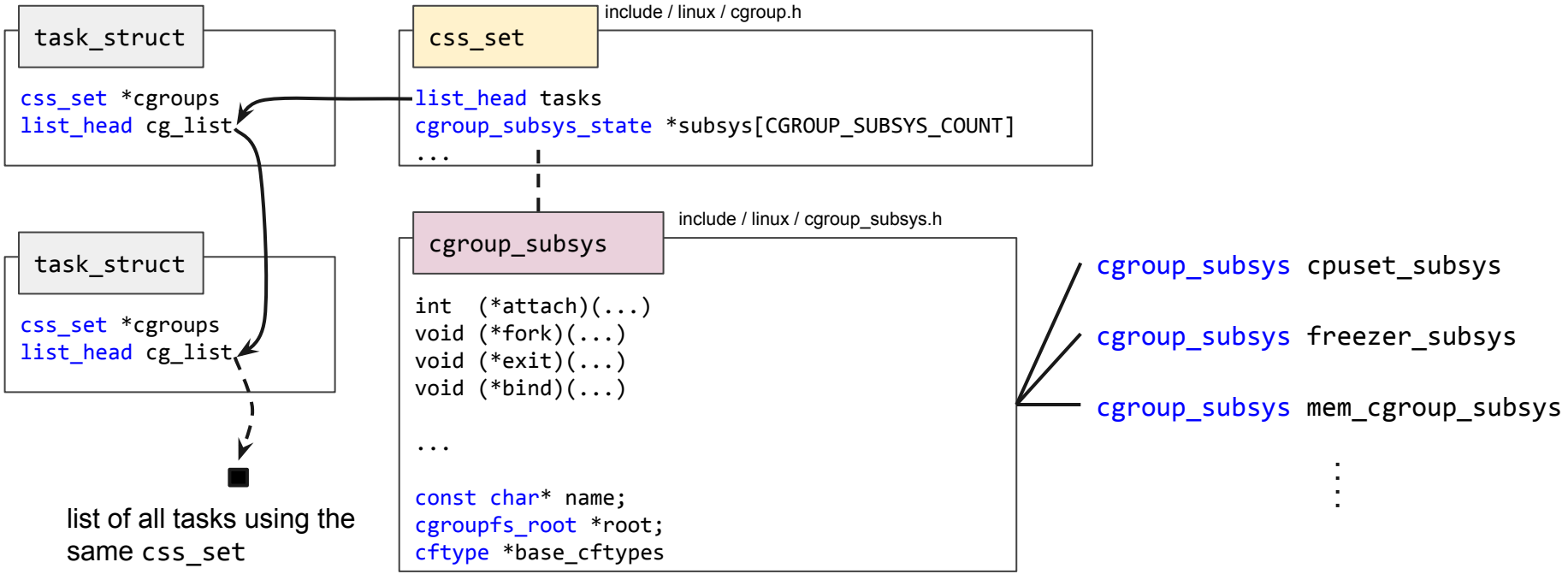


list of all tasks using the same `css_set`

kernel code for attach/detaching task from css_set

```
init/main.c
fork(), exit()
```

cgroups - kernel space view



cgroups - kernel space view

include / linux / cgroup_subsys.h

cgroup_subsys

```
int  (*attach)(...)
void (*fork)(...)
void (*exit)(...)
void (*bind)(...)

...

const char* name;
cgroupfs_root *root;
cftype *base_cftypes
```

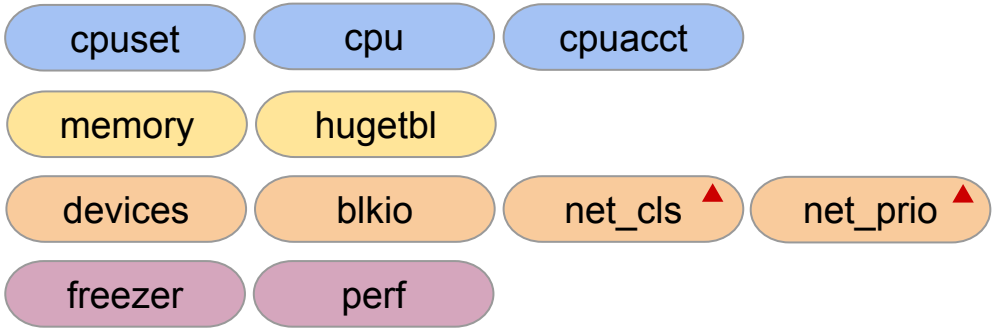
```
cgroup_subsys cpuset_subsys
                .base_cftypes = files
```

```
1819 static struct cftype files[] = {
1820 ▸-{
1821 ▸-▸-.name = "cpus",
1822 ▸-▸-.seq_show = cpuset_common_seq_show,
1823 ▸-▸-.write_string = cpuset_write_resmask,
1824 ▸-▸-.max_write_len = (100U + 6 * NR_CPUS),
1825 ▸-▸-.private = FILE_CPULIST,
1826 ▸-},
1827
1828 ▸-{
1829 ▸-▸-.name = "mems",
1830 ▸-▸-.seq_show = cpuset_common_seq_show,
1831 ▸-▸-.write_string = cpuset_write_resmask,
1832 ▸-▸-.max_write_len = (100U + 6 * MAX_NUMNODES),
1833 ▸-▸-.private = FILE_MEMLIST,
1834 ▸-},
1835
1836 ▸-{
1837 ▸-▸-.name = "cpu_exclusive",
1838 ▸-▸-.read_u64 = cpuset_read_u64,
1839 ▸-▸-.write_u64 = cpuset_write_u64,
1840 ▸-▸-.private = FILE_CPU_EXCLUSIVE,
1841 ▸-},
1842
1843 ▸-{
1844 ▸-▸-.name = "mem_exclusive",
1845 ▸-▸-.read_u64 = cpuset_read_u64,
1846 ▸-▸-.write_u64 = cpuset_write_u64,
1847 ▸-▸-.private = FILE_MEM_EXCLUSIVE,
1848 ▸-},
```

cgroups - summary

cgroup hierarchies ← each subsystem can be used at most once* — **subsystems (controllers)**

```
/sys/fs/cgroup
TL  /cpu   [cpu] [cpuacct]
      /high-priority
      /normal
      /experiment_1
TL  /mem   [memory]
      /opus
      /normal
      /experiment_1
```



▲ built as kernel module

TL top level cgroup (mount)

namespaces - user space view

Namespaces limit the scope of kernel-side **names** and **data structures** at process granularity

mnt (mount points, filesystems)	*CLONE_NEWNS*
pid (processes)	*CLONE_NEWPID*
net (network stack)	*CLONE_NEWNET*
ipc (System V IPC)	*CLONE_NEWIPC*
uts (unix timesharing - domain name, etc)	*CLONE_NEWUTS*
user (UIDs)	*CLONE_NEWUSER*

namespaces - user space view

Namespaces limit the scope of kernel-side **names** and **data structures** at process granularity

Three system calls for management

`clone()`	new process, new namespace, attach process to ns
`unshare()`	new namespace, attach current process to it
`setns(int fd, int nstype)`	join an existing namespace

namespaces - user space view

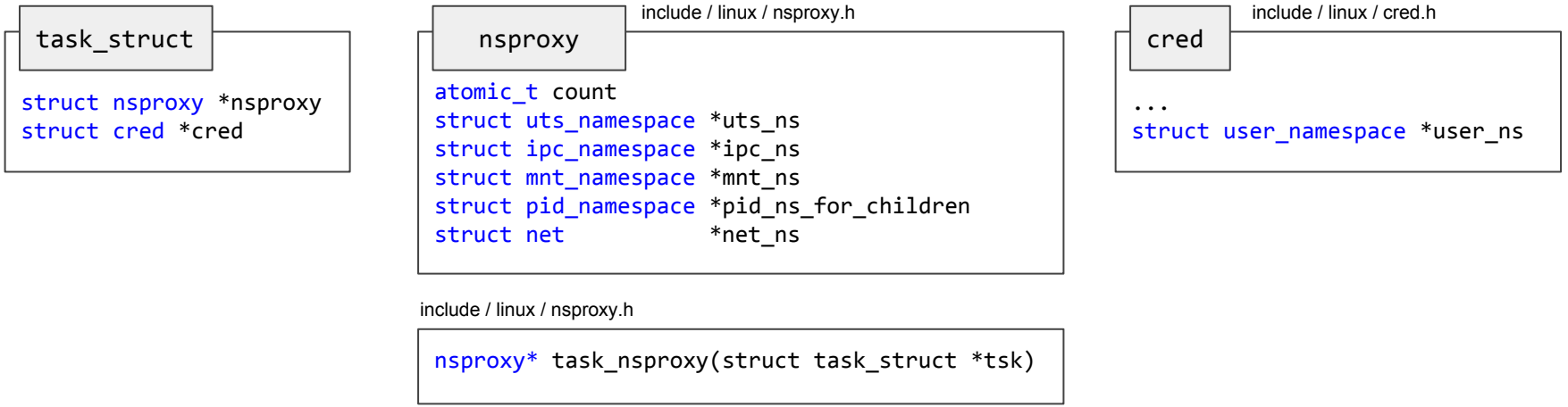
- each namespace is identified by an *inode* (unique)
- six(?) entries (inodes) added to `/proc/<pid>/ns/`

- two processes are in the same namespace if they see the same inode for equivalent namespace types (mnt, net, user, ...)

User space utilities

* IPROUTE (`ip netns add`, etc)
* `unshare`, `nsenter` (part of util-linux)
* `shadow, shadow-utils` (for user ns)

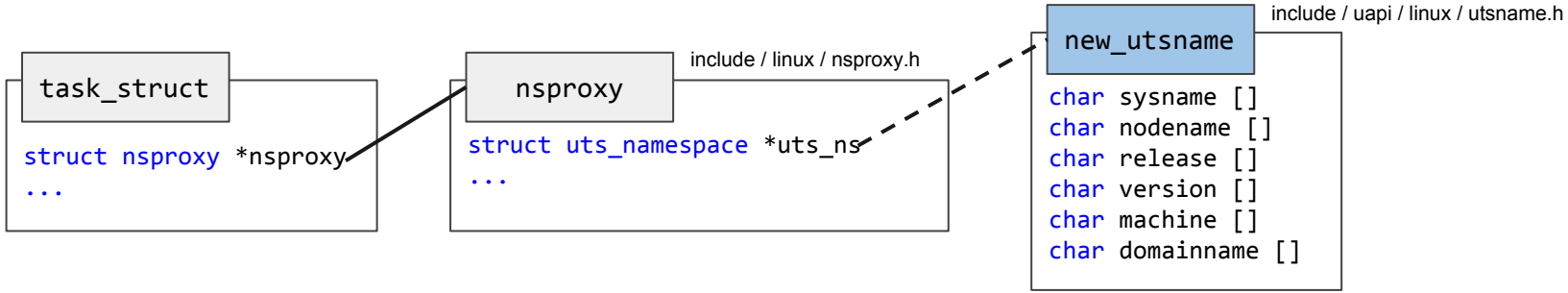
namespaces - kernel space view



- For each namespace type, a default namespace exists (the global namespace)
- `struct nsproxy` is shared by all tasks with the same **set** of namespaces

namespaces - kernel space view

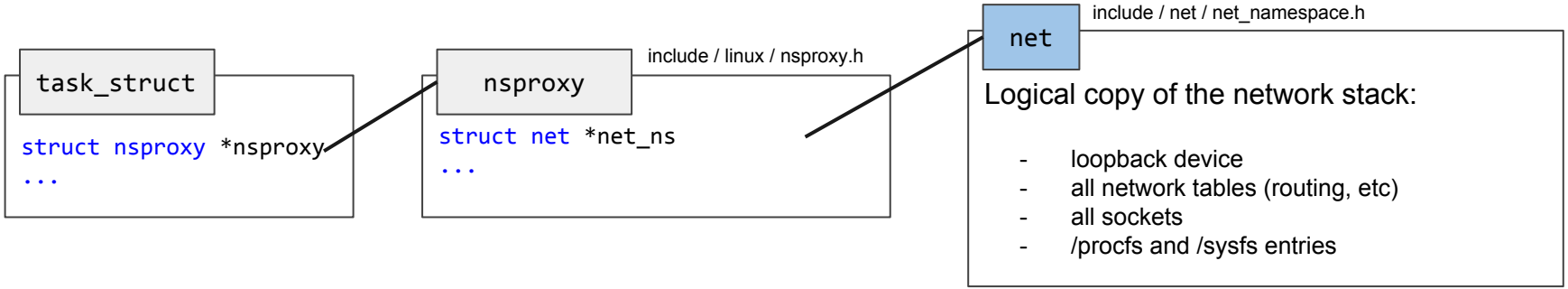
Example for **uts** namespace



- global access to hostname: `system_utsname.nodename`
- namespace-aware access to hostname: `&current->nsproxy->uts_ns->name->nodename`

namespaces - kernel space view

Example for **net** namespace



- a network device belongs to exactly one network namespace
- a socket belongs to exactly one network namespace
- a new network namespace only includes the loopback device
- communication between namespaces using veth or unix sockets

namespaces - summary

Namespaces limit the scope of kernel-side **names** and **data structures** at process granularity

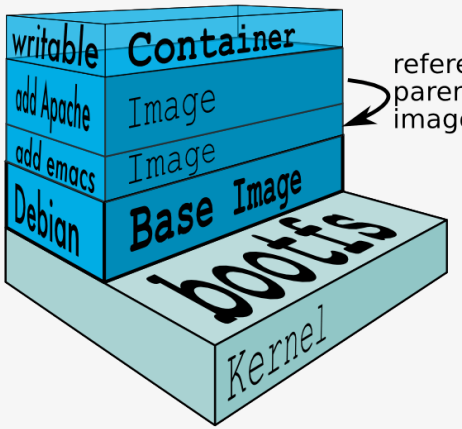
- **mnt** (mount points, filesystems)
- **pid** (processes)
- **net** (network stack)
- **ipc** (System V IPC)
- **uts** (unix timesharing - domain name, etc)
- **user** (UIDs)

Containers

- A light form of resource virtualization based on kernel mechanisms
- A container is a *user-space* construct

- Multiple containers run on top of the same kernel
  - illusion that they are the only one using resources (cpu, memory, disk, network)

- some implementations offer support for
  - container templates
  - deployment / migration
  - union filesystems



taken from the Docker documentation

Container solutions

Mainline

Google containers (lmctfy)

- uses cgroups only, offers CPU & memory isolation
- no isolation for: disk I/O, network, filesystem, checkpoint/restore
- adds some cgroup files: `cpu.lat`, `cpuacct.histogram`

LXC: user-space containerisation tools

Docker

systemd-nspawn

Forks

Vserver, OpenVZ

Container solutions - LXC

An LXC container is a userspace process created with the `clone()` system call

- with its own pid namespace
- with its own mnt namespace
- net namespace (configurable) - lxc.network.type

Offers container templates `/usr/share/lxc/templates`

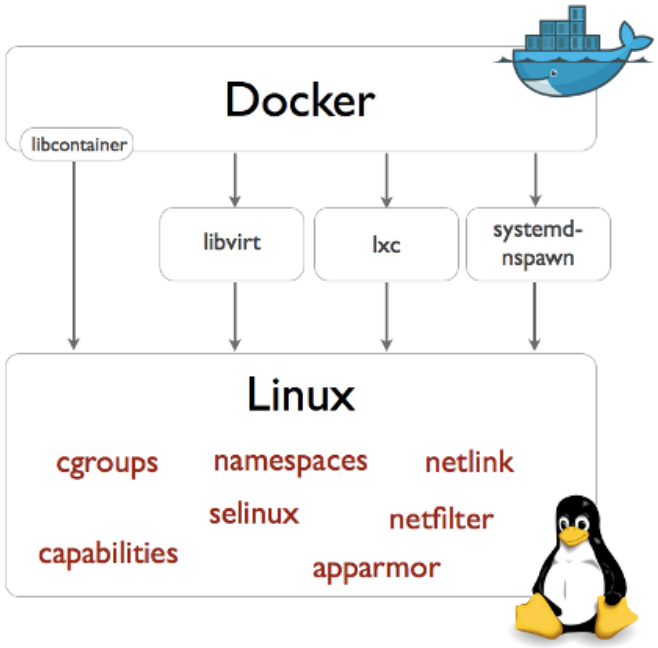
- shell scripts
- `lxc-create -t ubuntu -n containerName`
  - also creates cgroup `/sys/fs/cgroup/<controller>/lxc/containerName`

Container solutions - Docker

A Linux container engine

- multiple backend drivers
- application rather than machine-centric
- app build tools
- diff-based deployment of updates (AUFS)
- versioning (git-like) and reuse

- links (tunnels) between containers



taken from the Docker documentation

Questions?

Thank you!

Lucian Carata
lc525@cam.ac.uk

More details

cgroups: http://media.wix.com/ugd/295986_d73d8d6087ed430c34c21f90b0b607fd.pdf

namespaces: http://lwn.net/Articles/531114/ (and series)