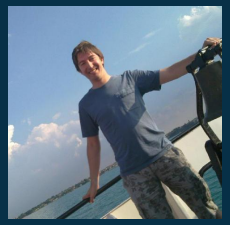
Container mechanics in Linux and rkt

FOSDEM 2016



Alban Crequy
github.com/alban

kinvolk.io

Jonathan Boulle
github.com/jonboulle
@baronboulle

CoreOS

a modern, secure, composable
container runtime

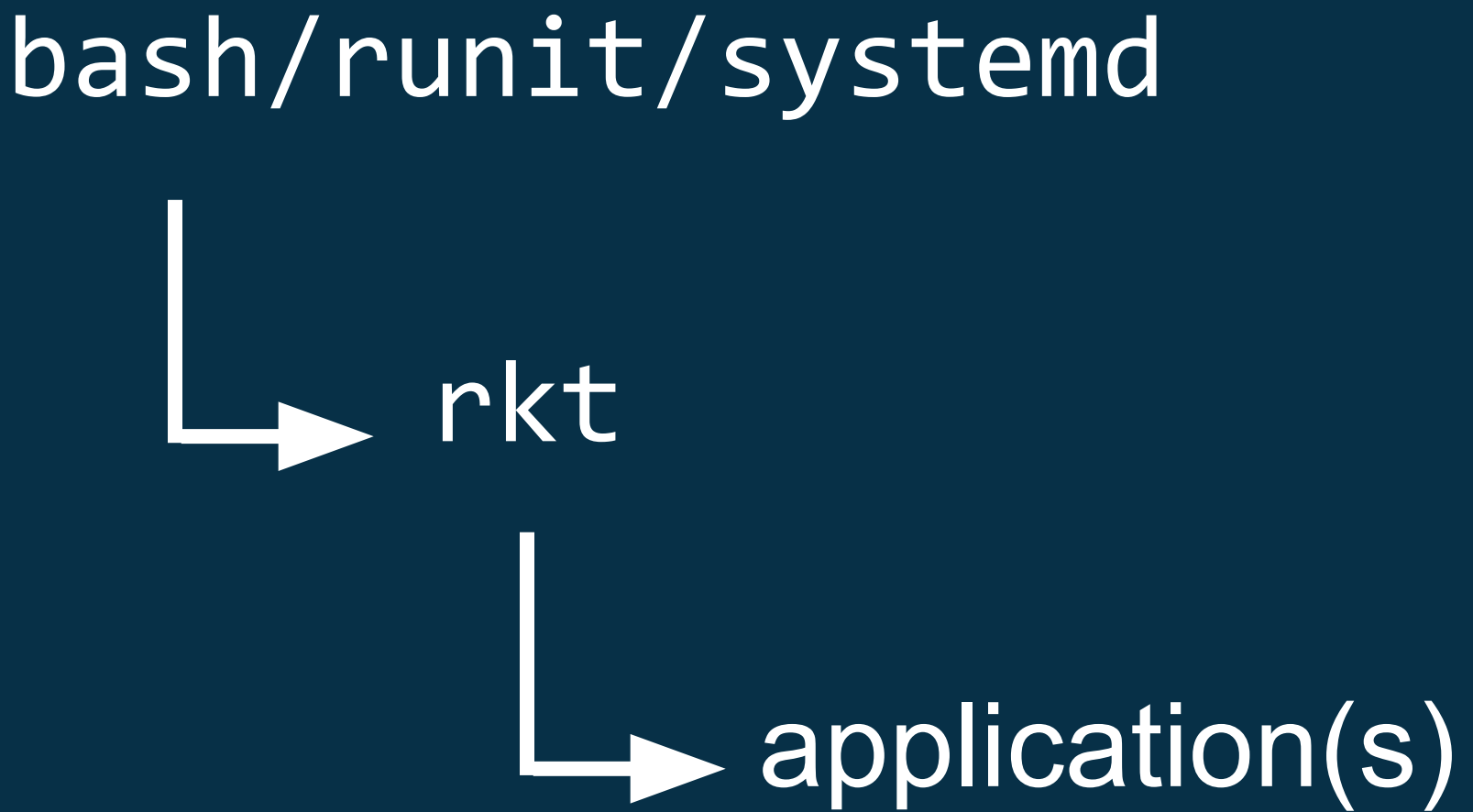
an implementation of appc
(image format, execution environment)

rkt

simple CLI tool
golang + Linux
self-contained

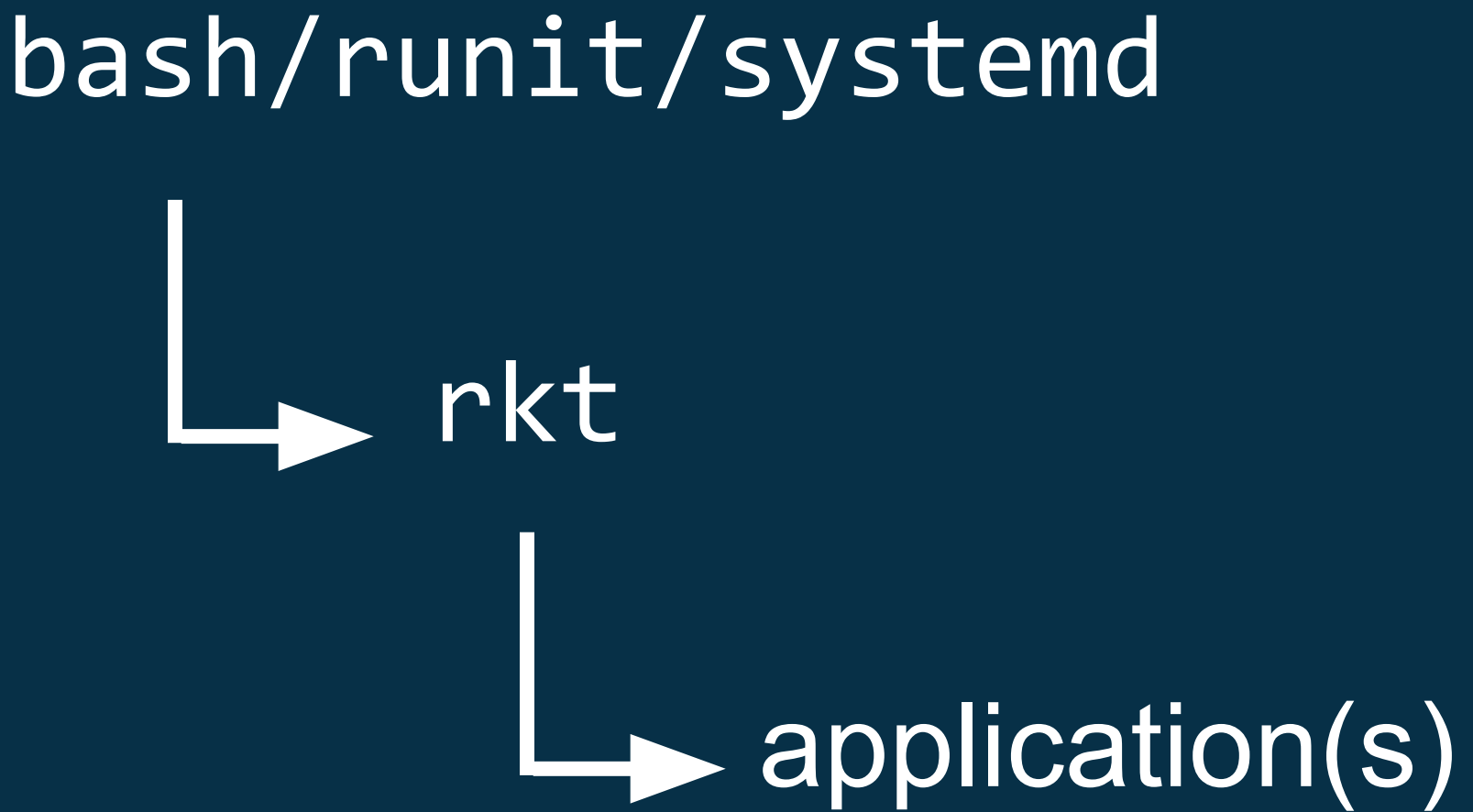
simple CLI tool

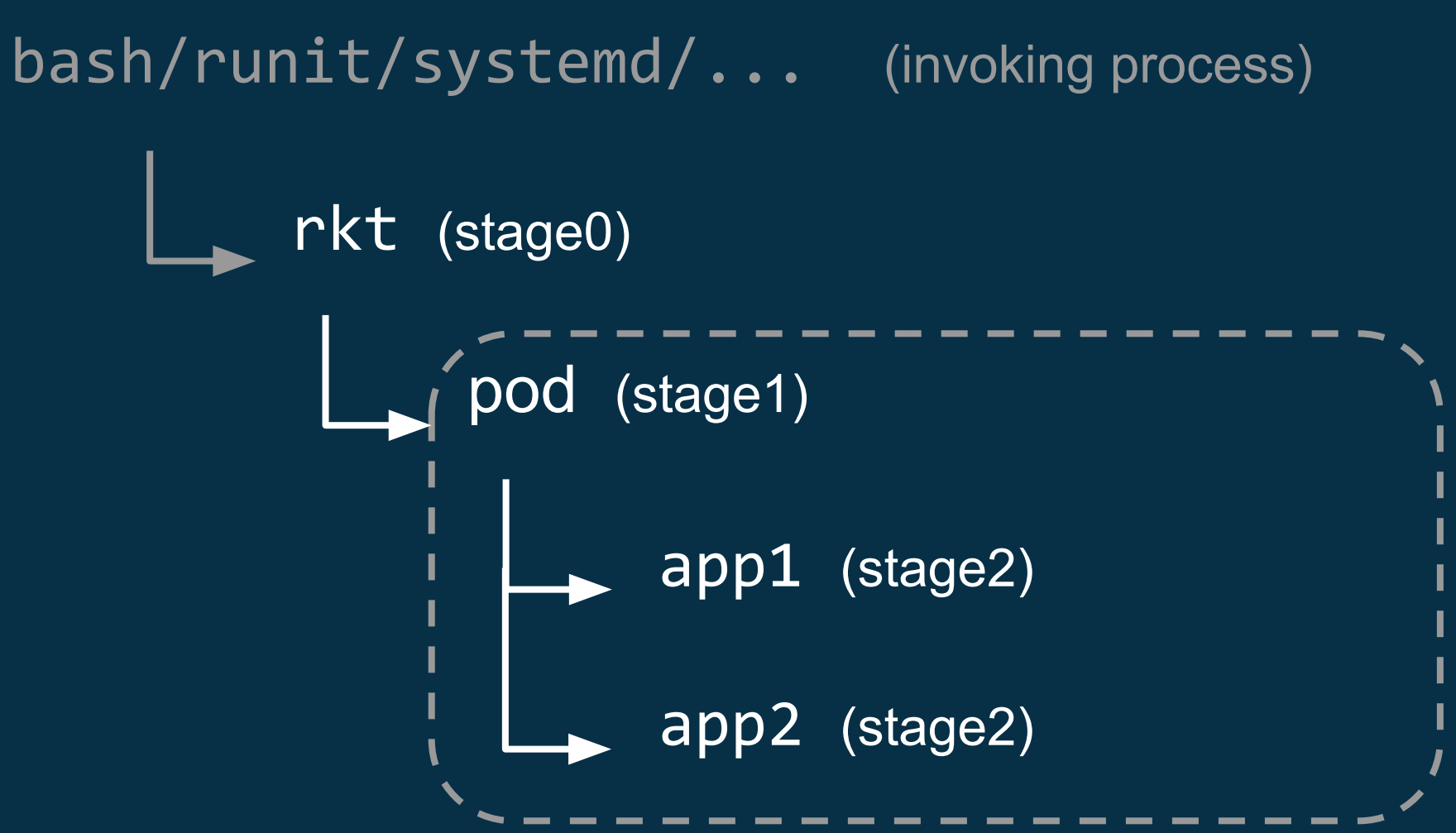
no (mandatory) daemon:
apps run directly under spawning process

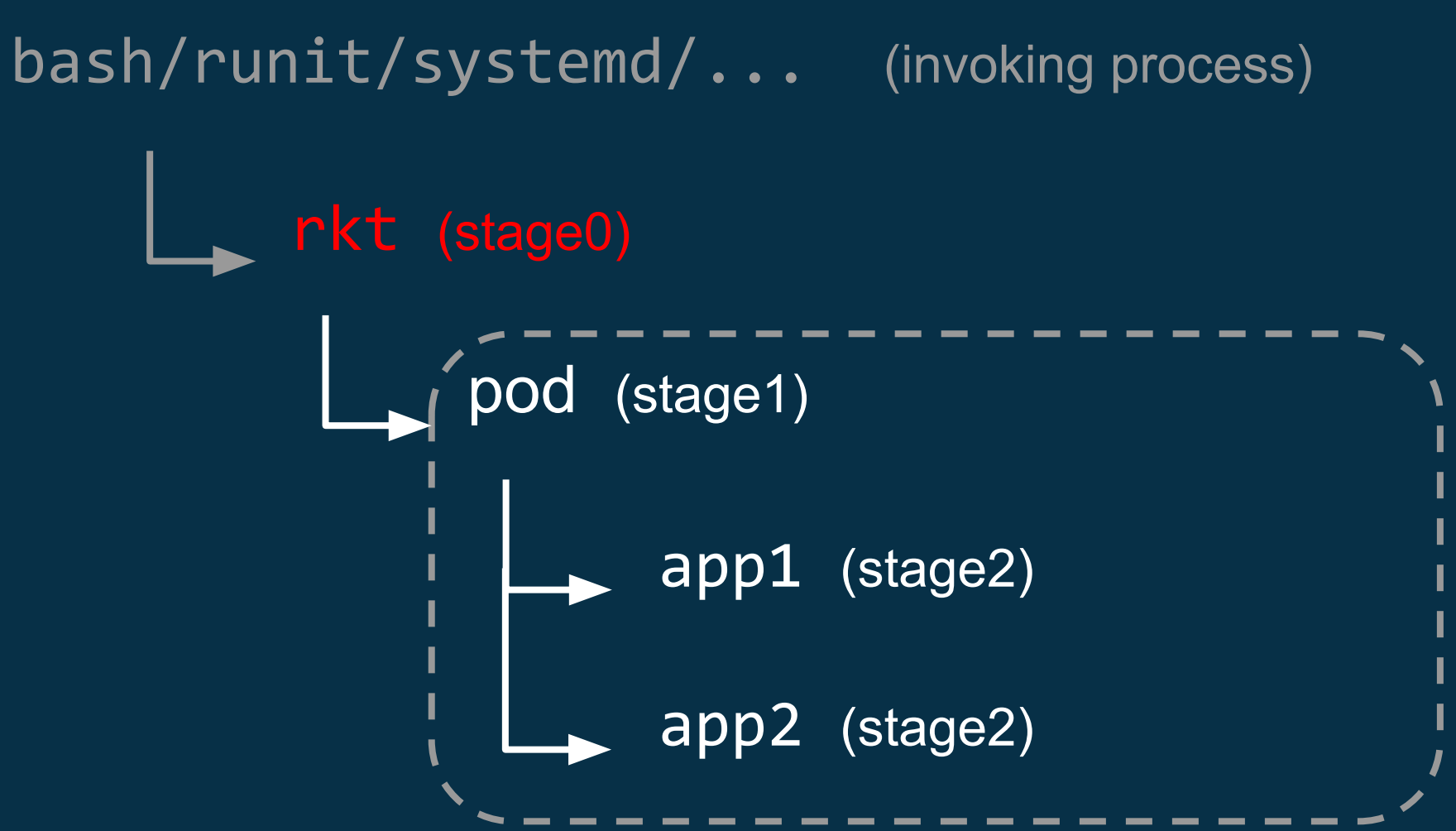

bash/runit/systemd
rkt
application(s)

rkt internals

modular architecture
execution divided into *stages*
stage0 → stage1 → stage2

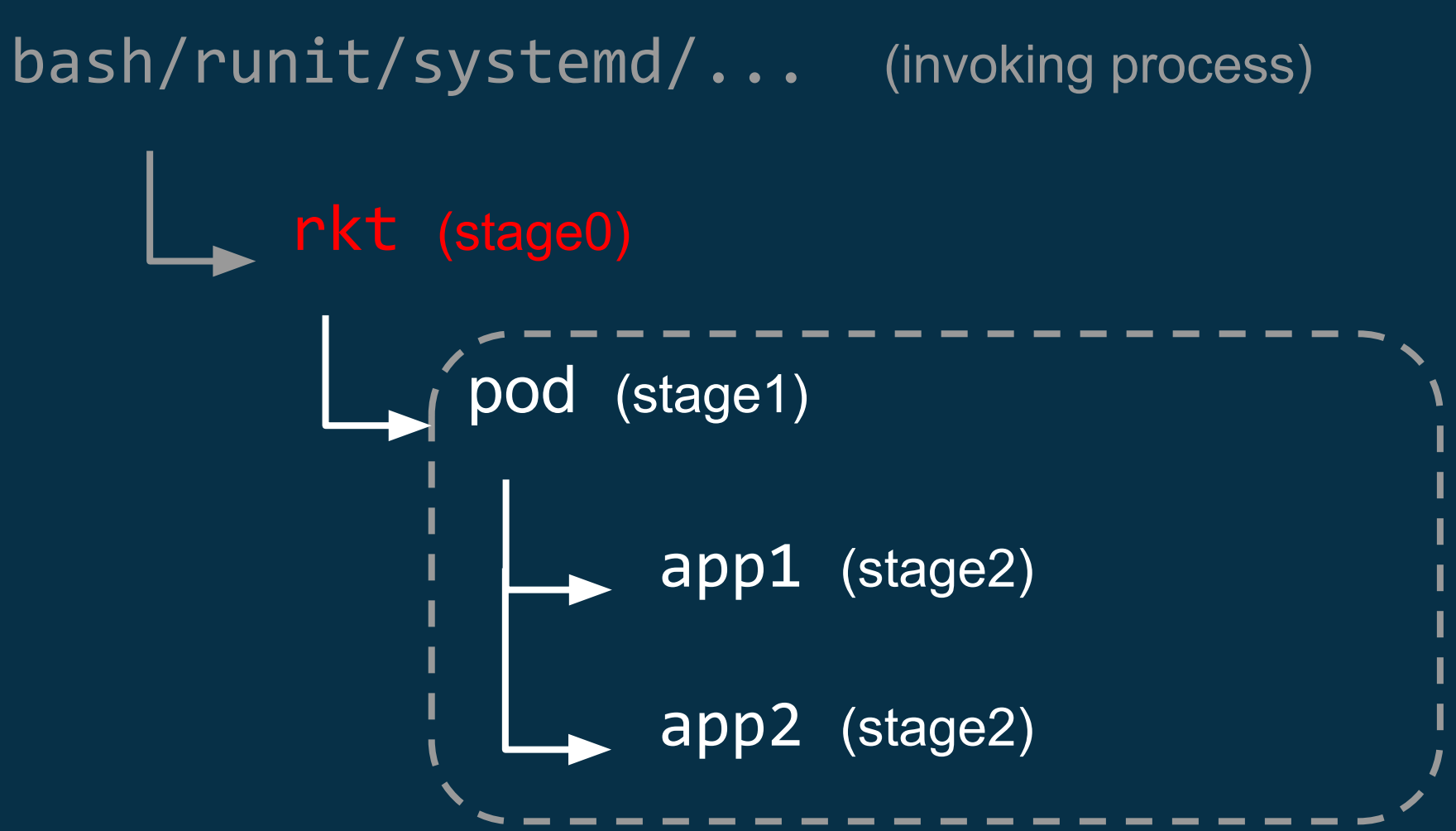

bash/runit/systemd
rkt
application(s)

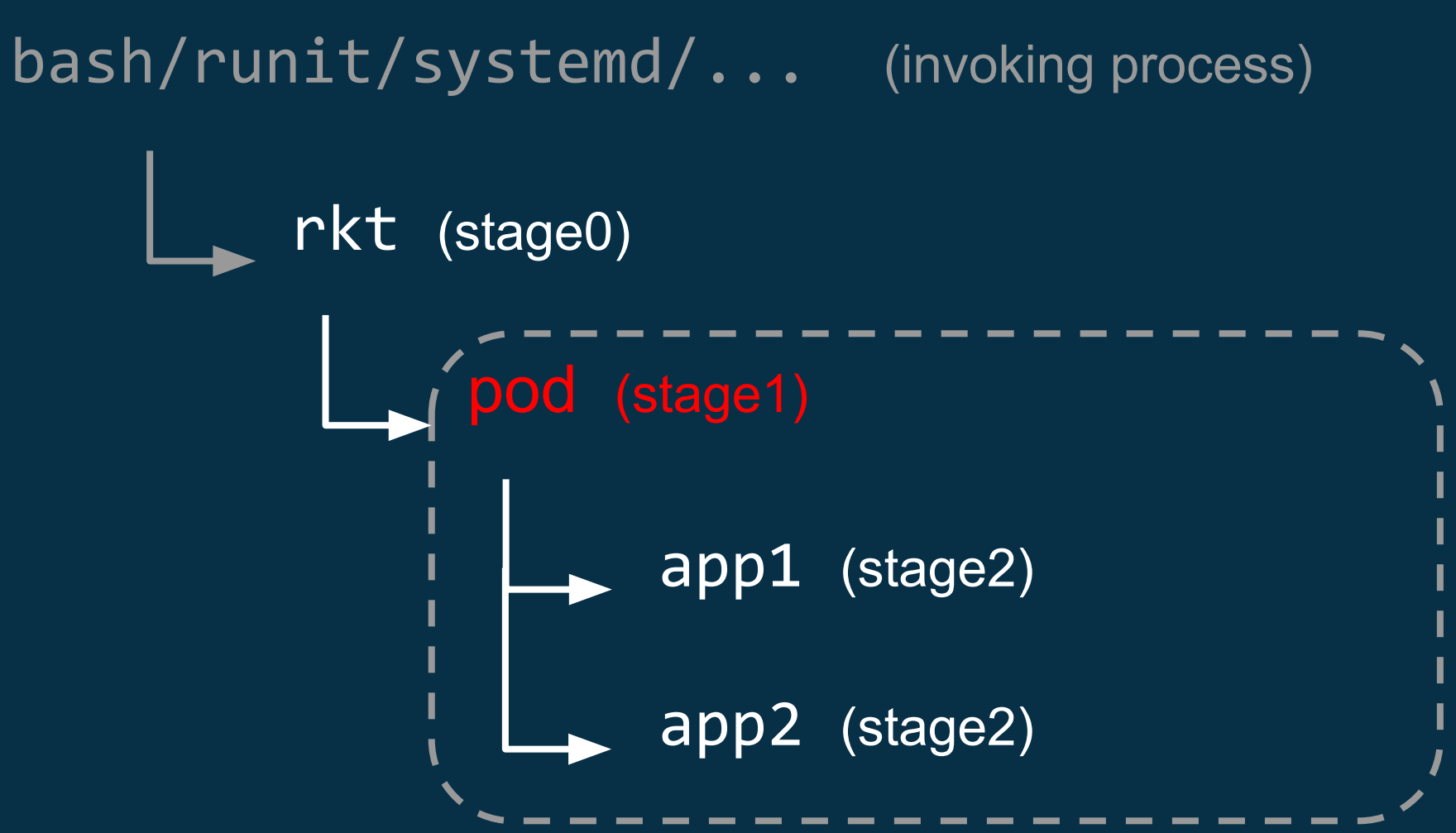

bash/runit/systemd/...
(invoking process)
rkt
(stage0)
pod
(stage1)
app1
(stage2)
app2
(stage2)


bash/runit/systemd/...
(invoking process)
rkt
(stage0)
pod
(stage1)
app1
(stage2)
app2
(stage2)

stage0 (`rkt` binary)

discover, fetch, manage application images
set up pod filesystems
commands to manage pod lifecycle


bash/runit/systemd/...
(invoking process)
rkt
(stage0)
pod
(stage1)
app1
(stage2)
app2
(stage2)


bash/runit/systemd/...
(invoking process)
rkt
(stage0)
pod
(stage1)
app1
(stage2)
app2
(stage2)

stage1

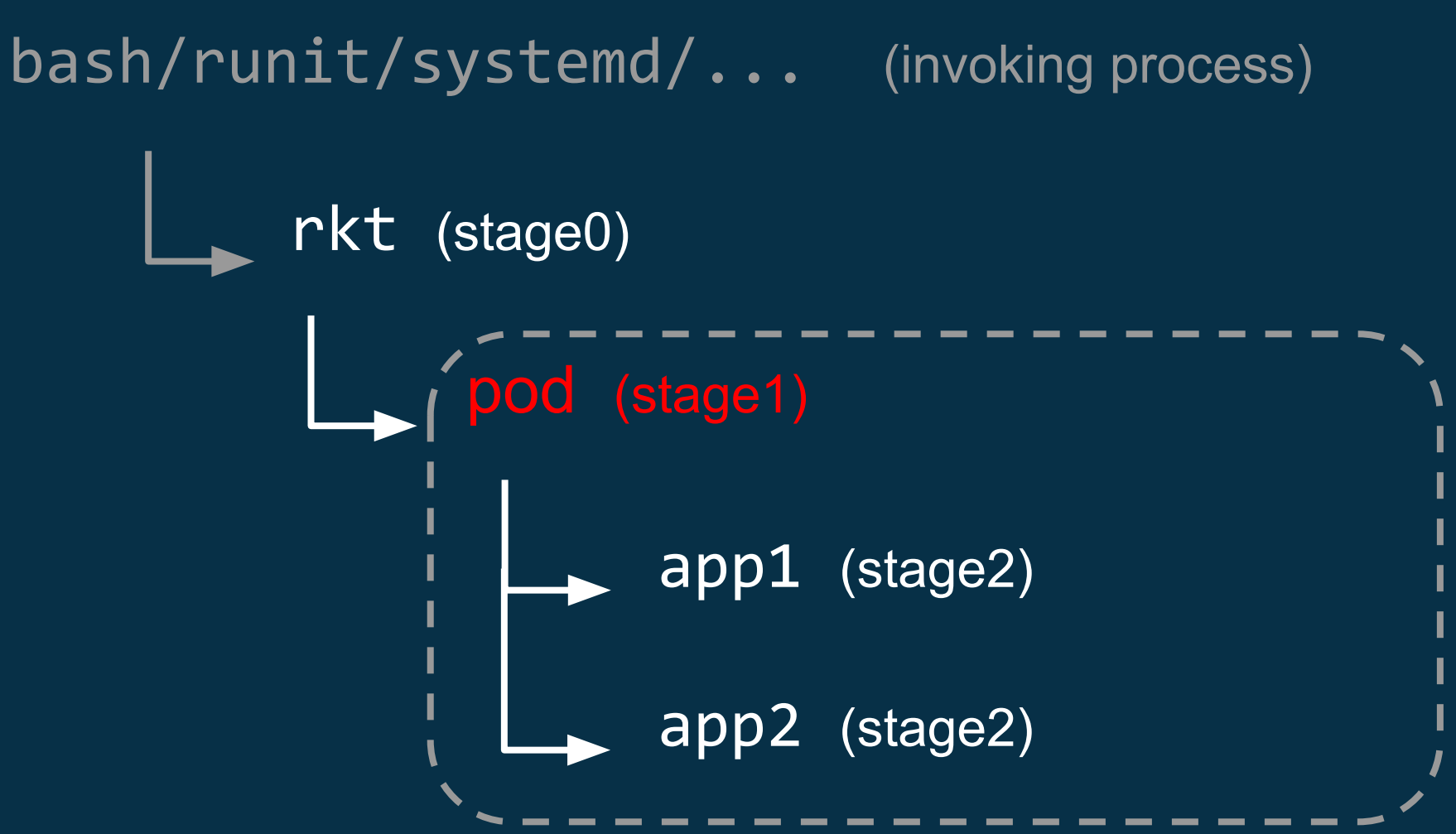
"the container"

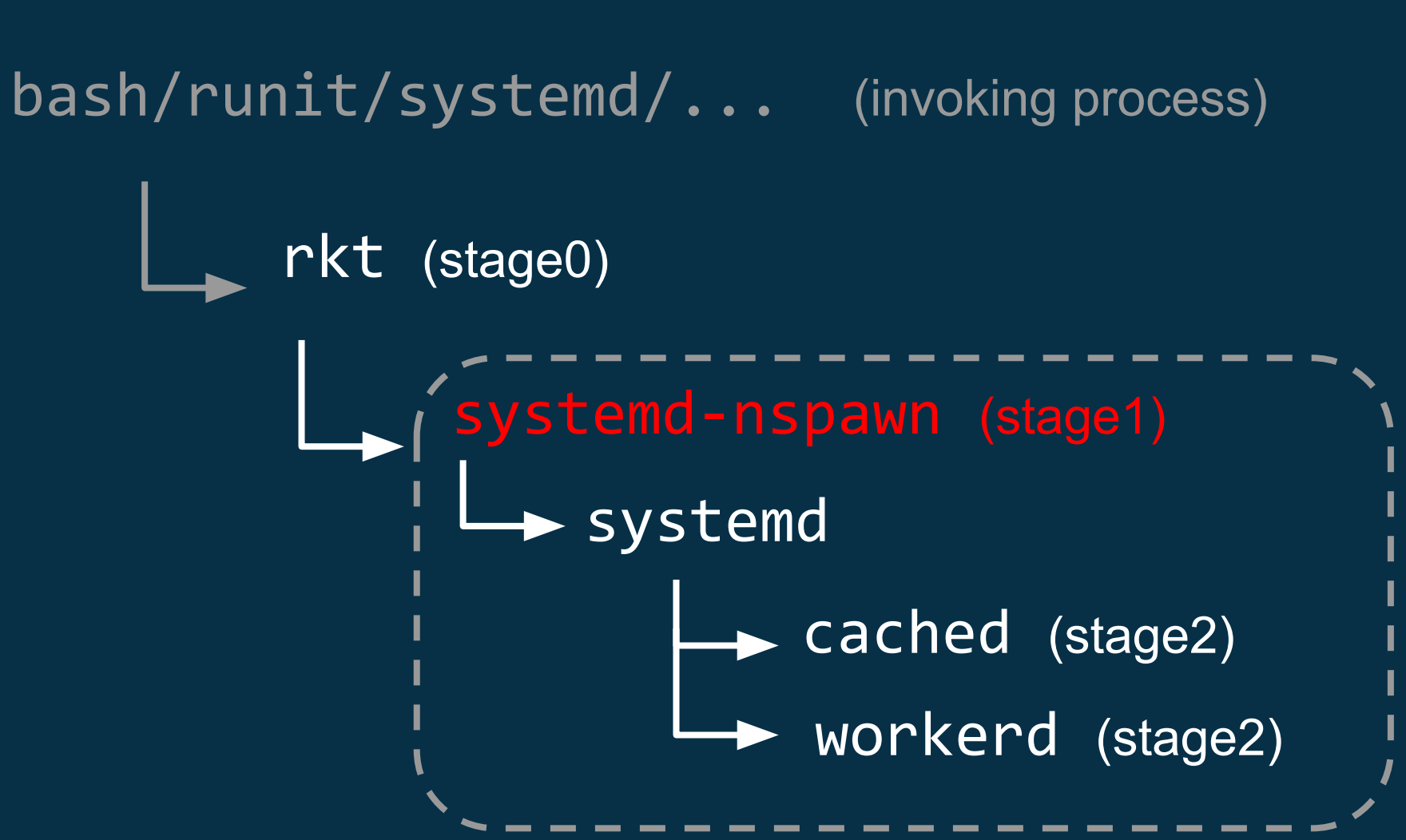
execution environment for pods
process lifecycle management
resource constraints (isolators)

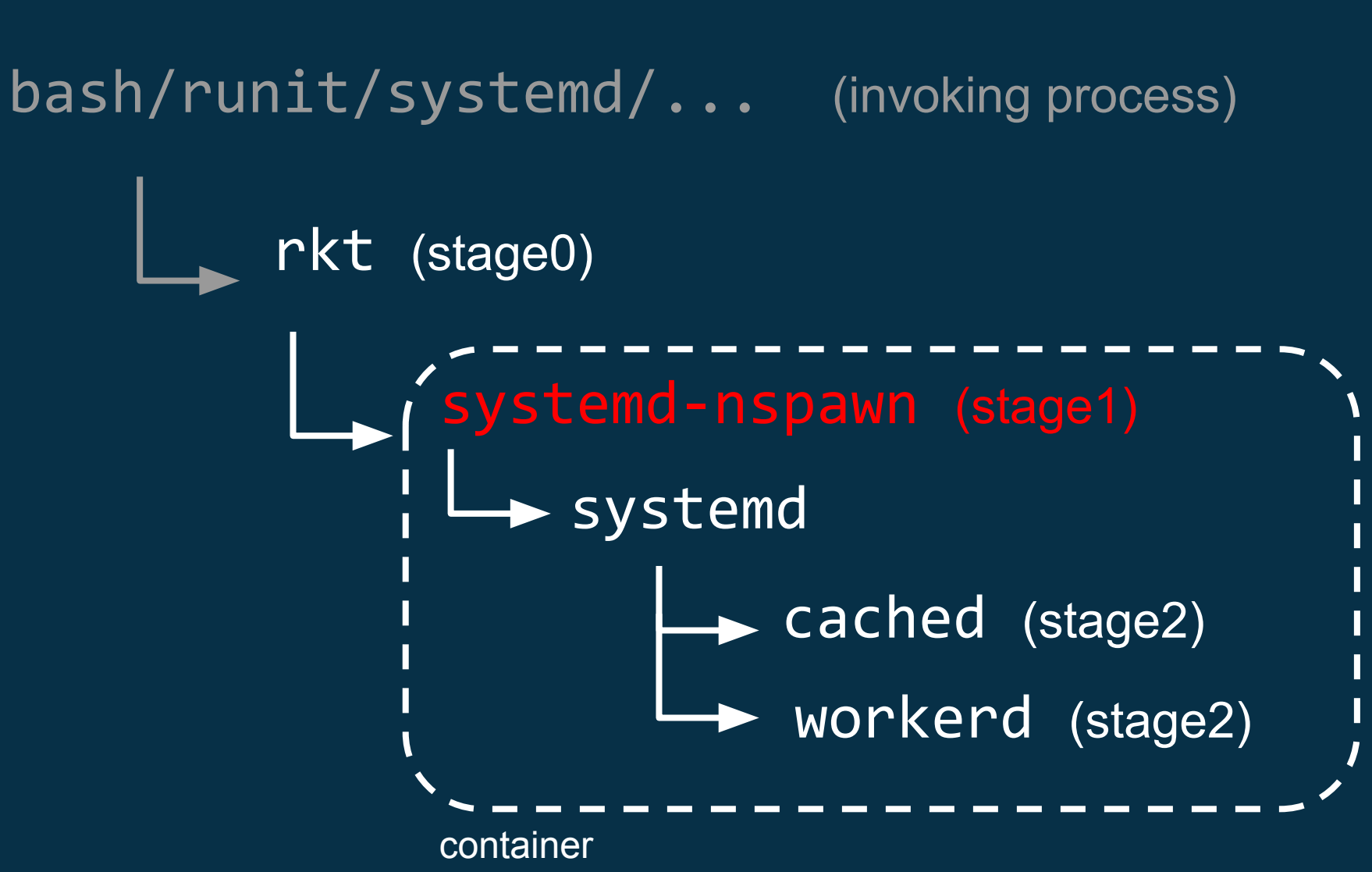
stage1 (swappable)

- binary ABI with stage0
  - rkt's stage0 calls `exec(stage1, args...)`

- default implementation
  - based on `systemd-nspawn + systemd`
  - Linux namespaces + cgroups for isolation
- kvm implementation
  - based on `lkvm + systemd`
  - hardware virtualisation for isolation


bash/runit/systemd/...
(invoking process)
rkt
(stage0)
pod
(stage1)
app1
(stage2)
app2
(stage2)


bash/runit/systemd/...
(invoking process)
rkt (stage0)
systemd-nspawn (stage1)
systemd
cached (stage2)
workerd (stage2)


bash/runit/systemd/...
(invoking process)
rkt (stage0)
systemd-nspawn (stage1)
systemd
cached (stage2)
workerd (stage2)
container

Containers on Linux

namespaces
cgroups
chroot

Linux namespaces

containers

hostname:
`thunderstorm`

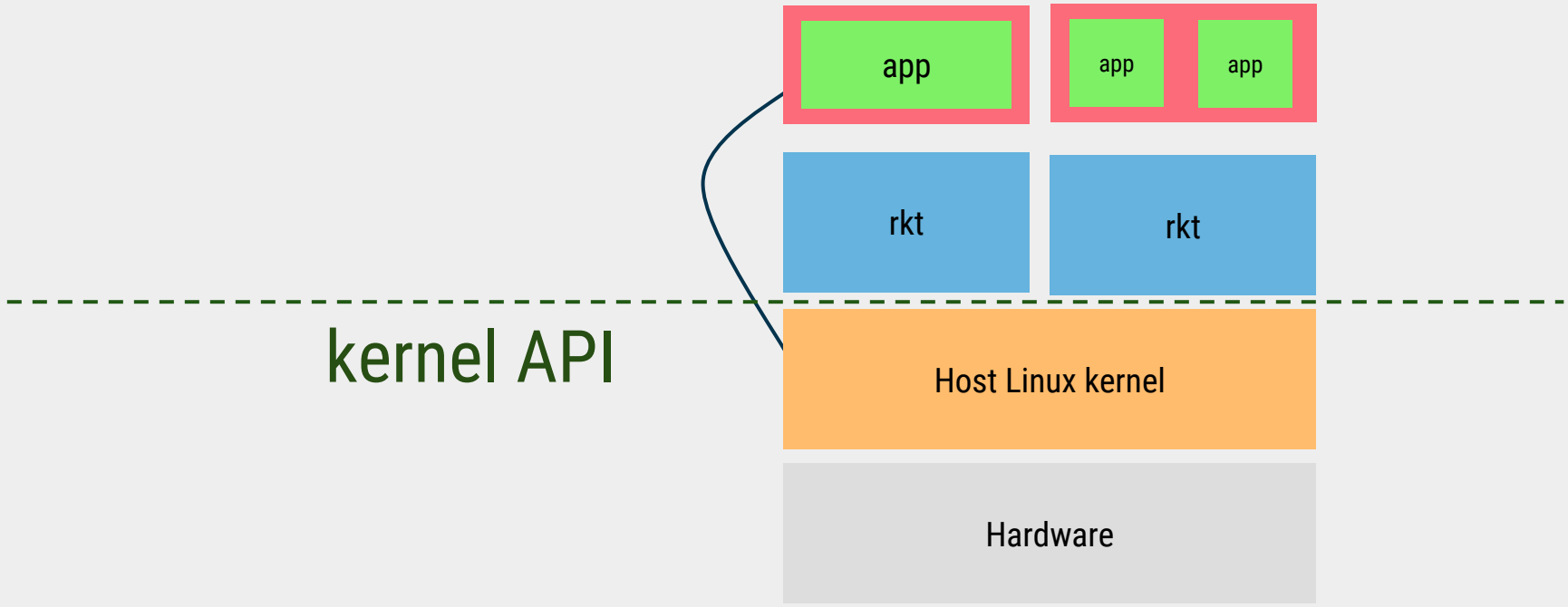
hostname:
`sunshine`

host

hostname:
`rainbow`

Containers: no guest kernel

system calls:
`example: sethostname()`



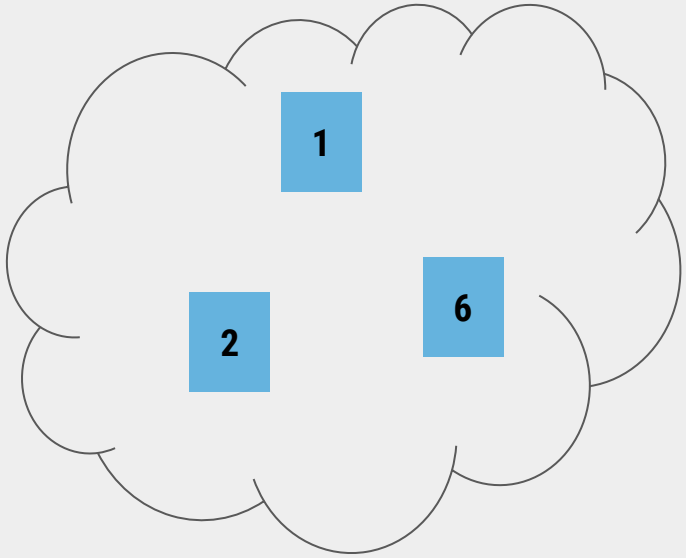
Containers with an example

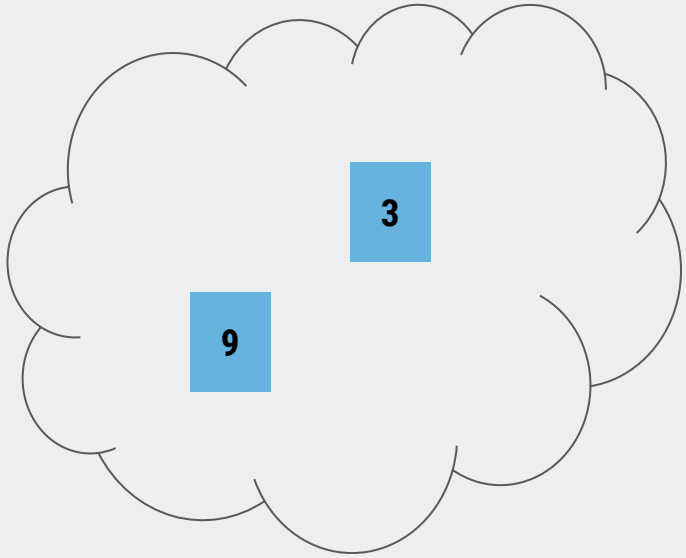
Getting and setting the hostname:

- **The system calls for getting and setting the hostname are older than containers**

```
int uname(struct utsname *buf);

int gethostname(char *name, size_t len);

int sethostname(const char *name, size_t len);
```

Processes in namespaces





`gethostname() -> “rainbow”`

`gethostname() -> “thunderstorm”`

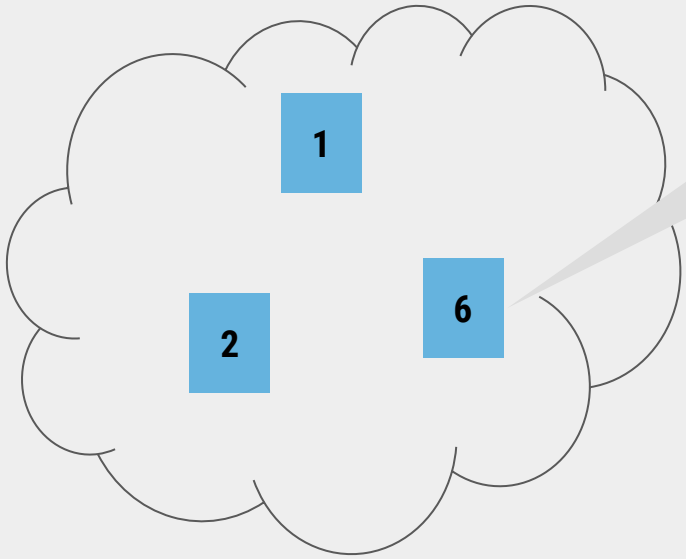
Linux Namespaces

Several independent namespaces

- **uts** (Unix Timesharing System) **namespace**
- **mount namespace**
- **pid namespace**
- **network namespace**
- **user namespace**

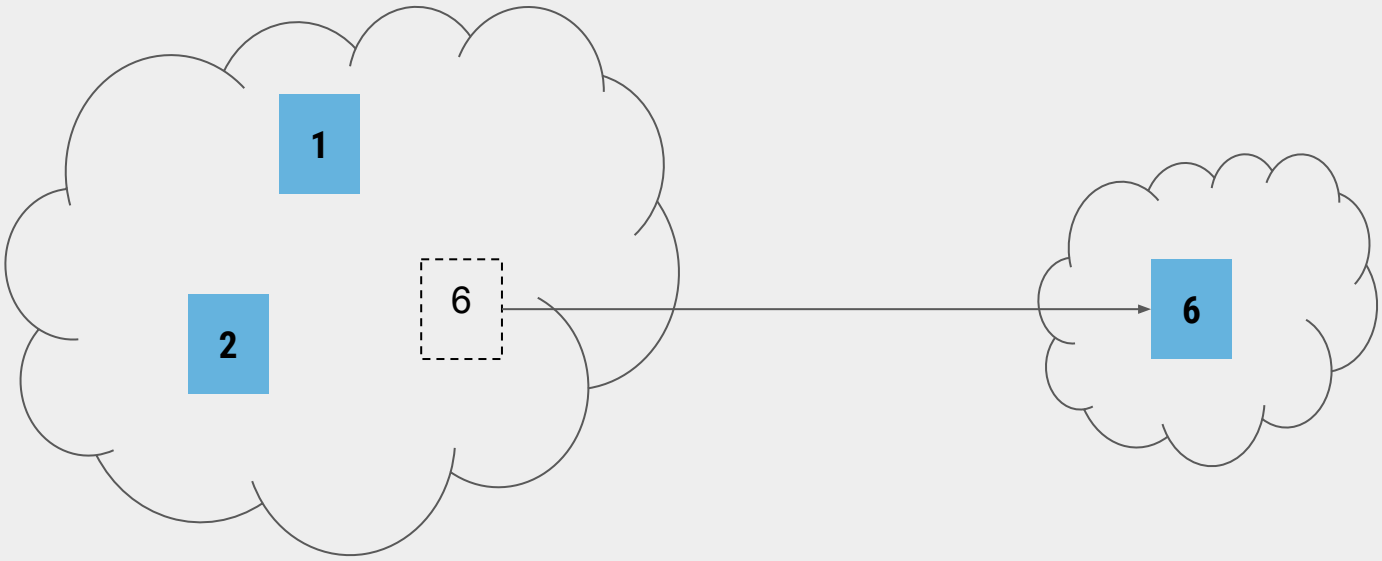
Creating new namespaces

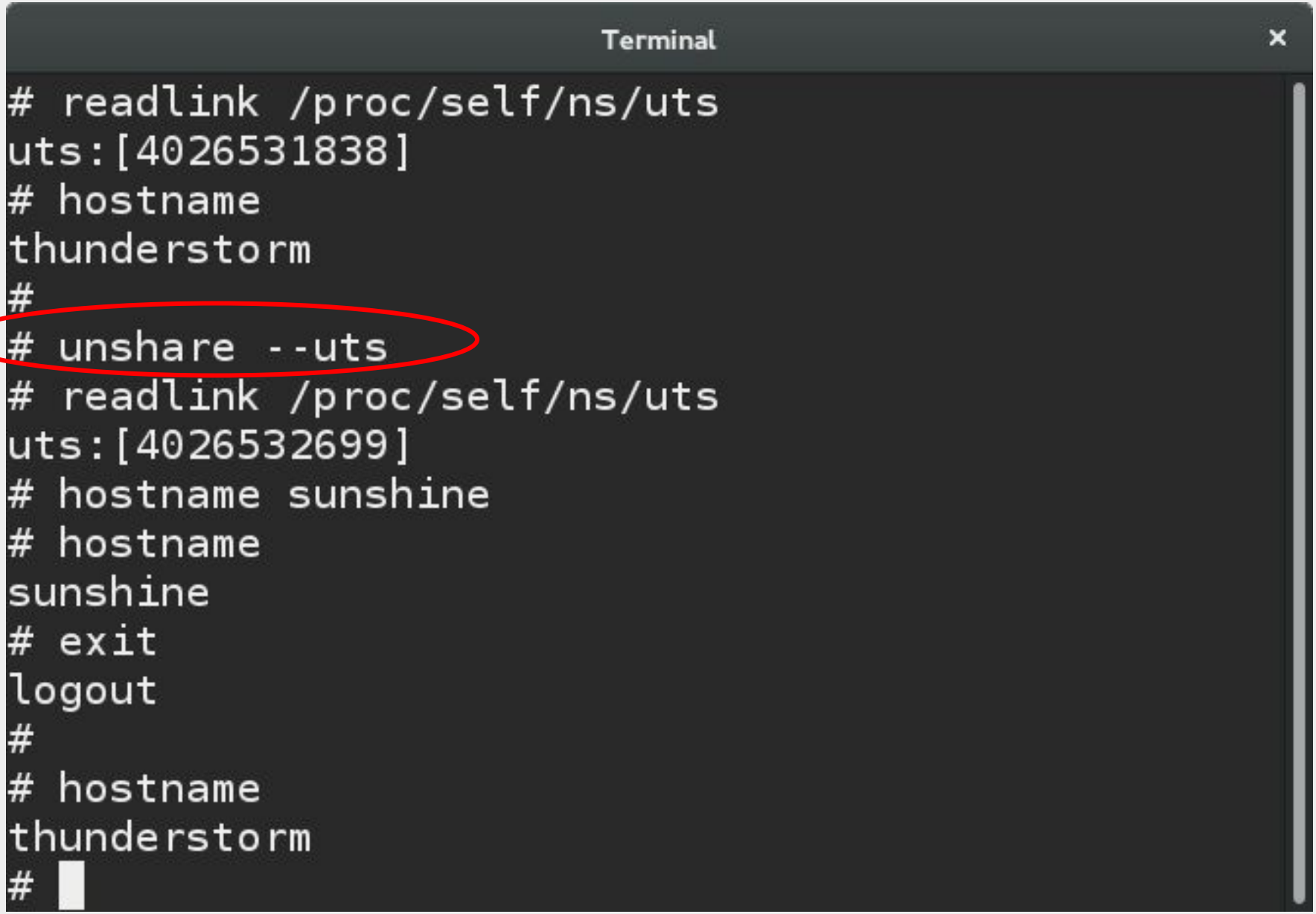
```
unshare(CLONE_NEWUTS);
```



“rainbow”

Creating new namespaces




Terminal
readlink /proc/self/ns/uts
uts:[4026531838]
hostname
thunderstorm
#
unshare --uts
readlink /proc/self/ns/uts
uts:[4026532699]
hostname sunshine
hostname
sunshine
exit
logout
#
hostname
thunderstorm
#

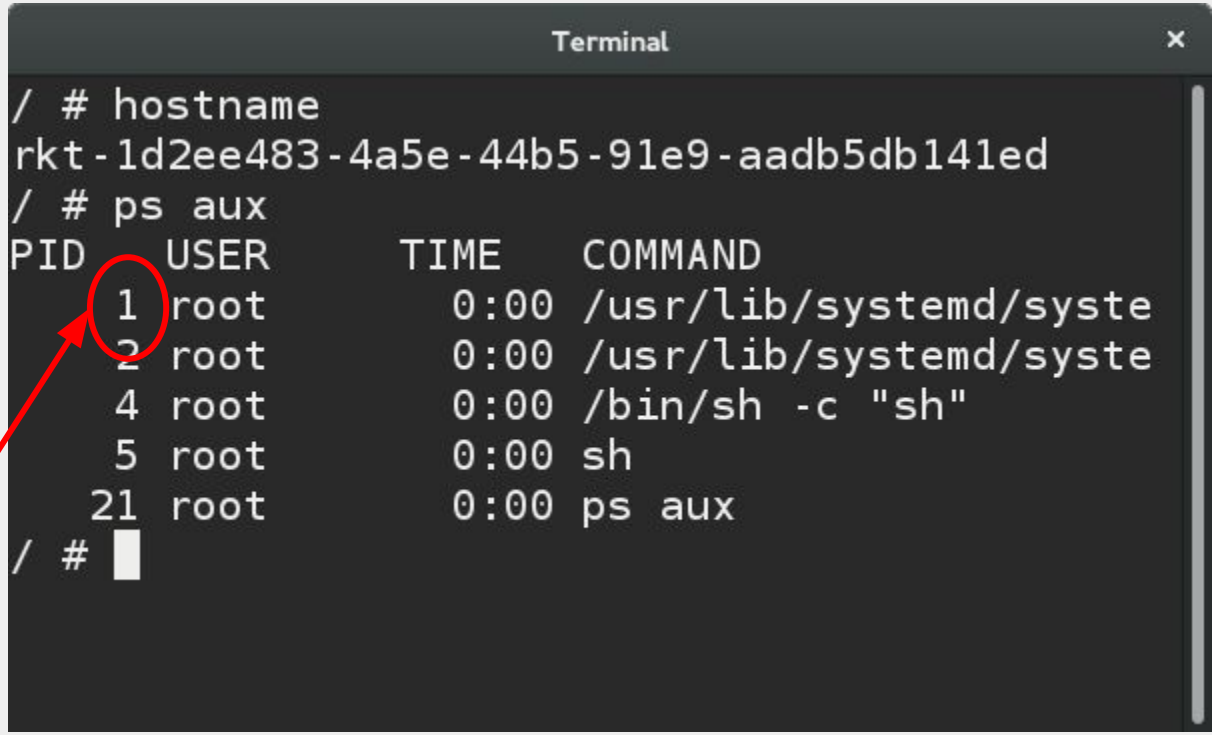
PID namespace

Hiding processes and PID translation

- the host sees all processes
- the container only its own processes

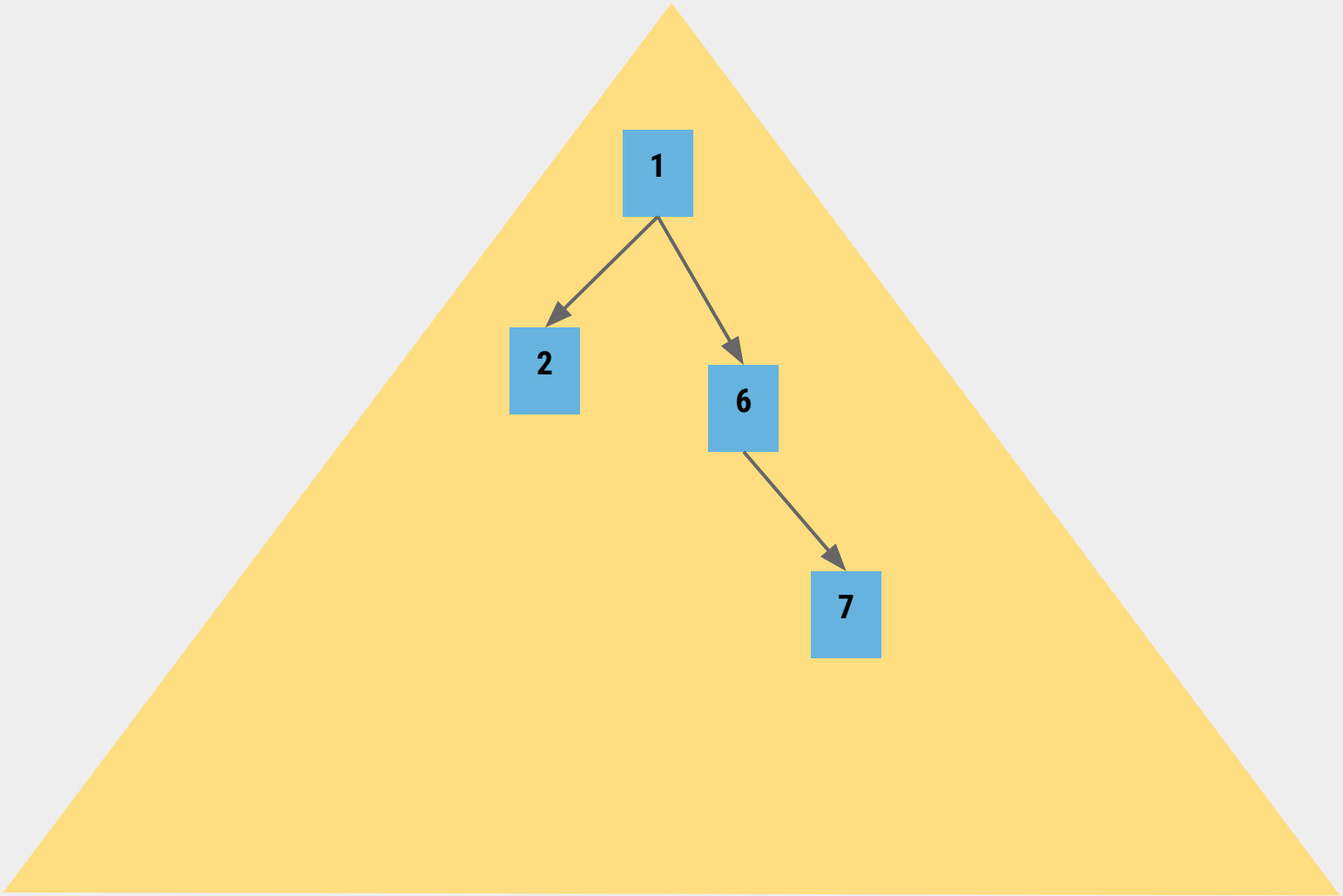
Hiding processes and PID translation

- the host sees all processes
- the container only its own processes



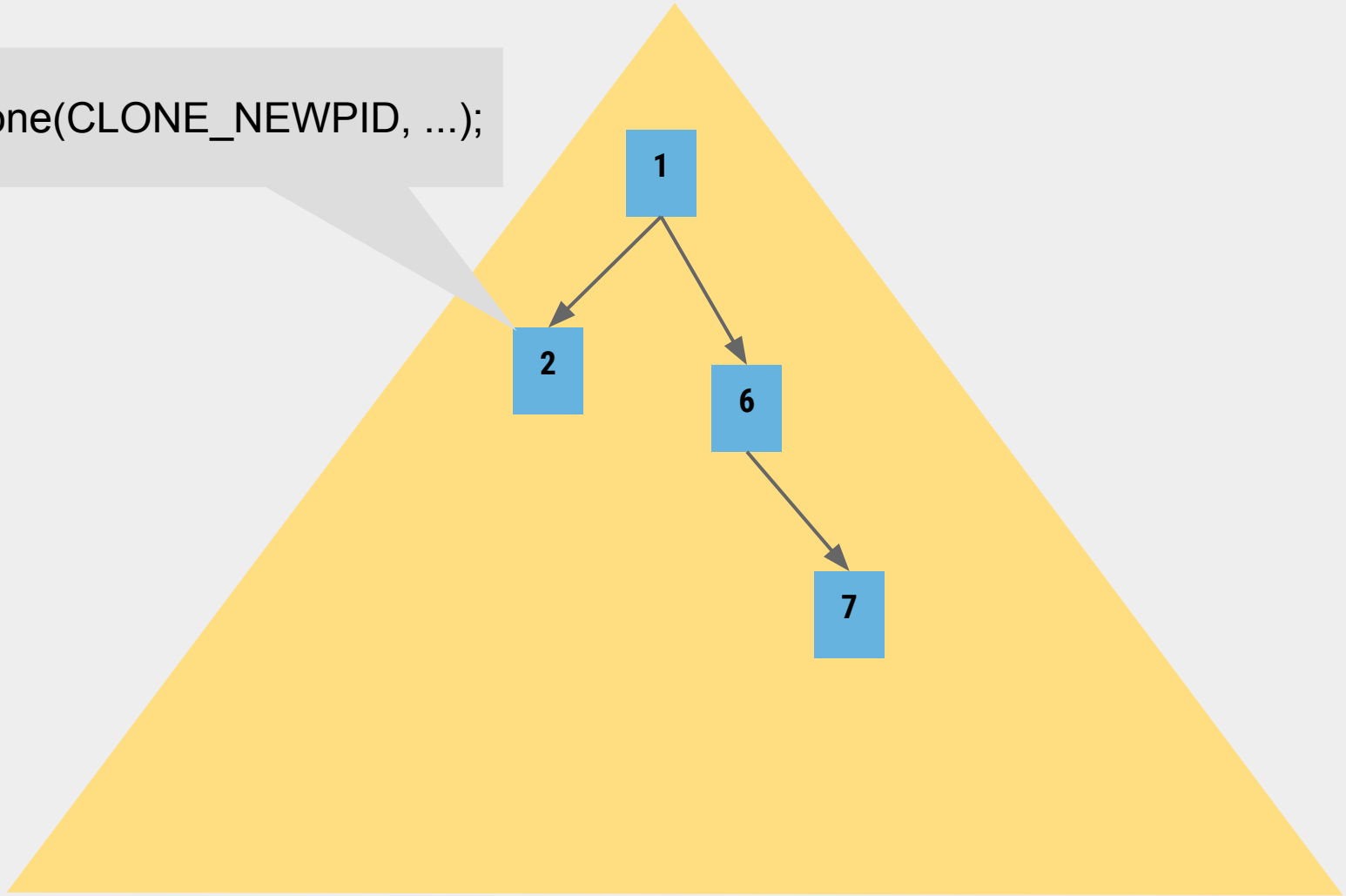
Actually pid 30920

Initial PID namespace

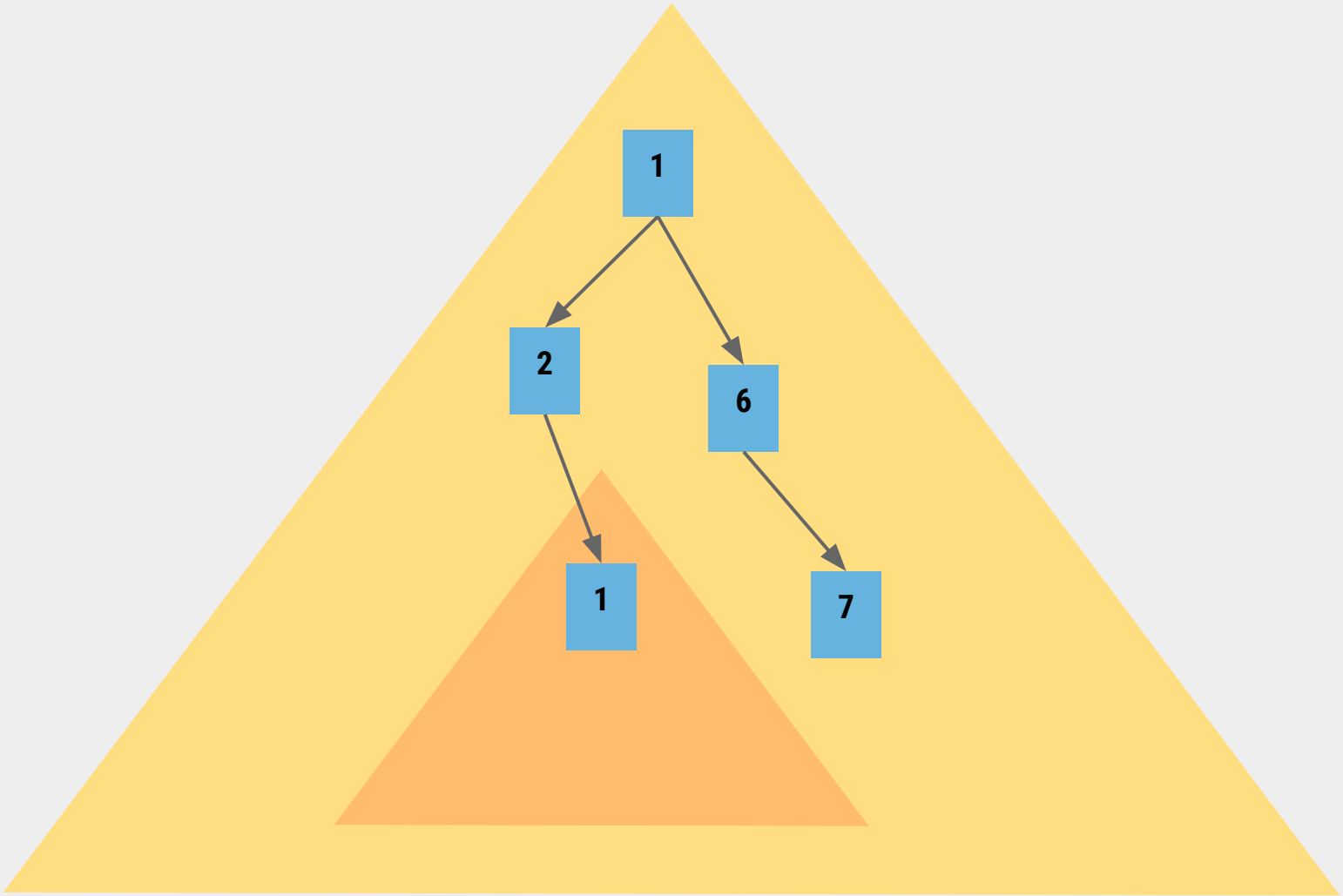


Creating a new namespace

clone(CLONE_NEWPID, ...);



Creating a new namespace



rkt

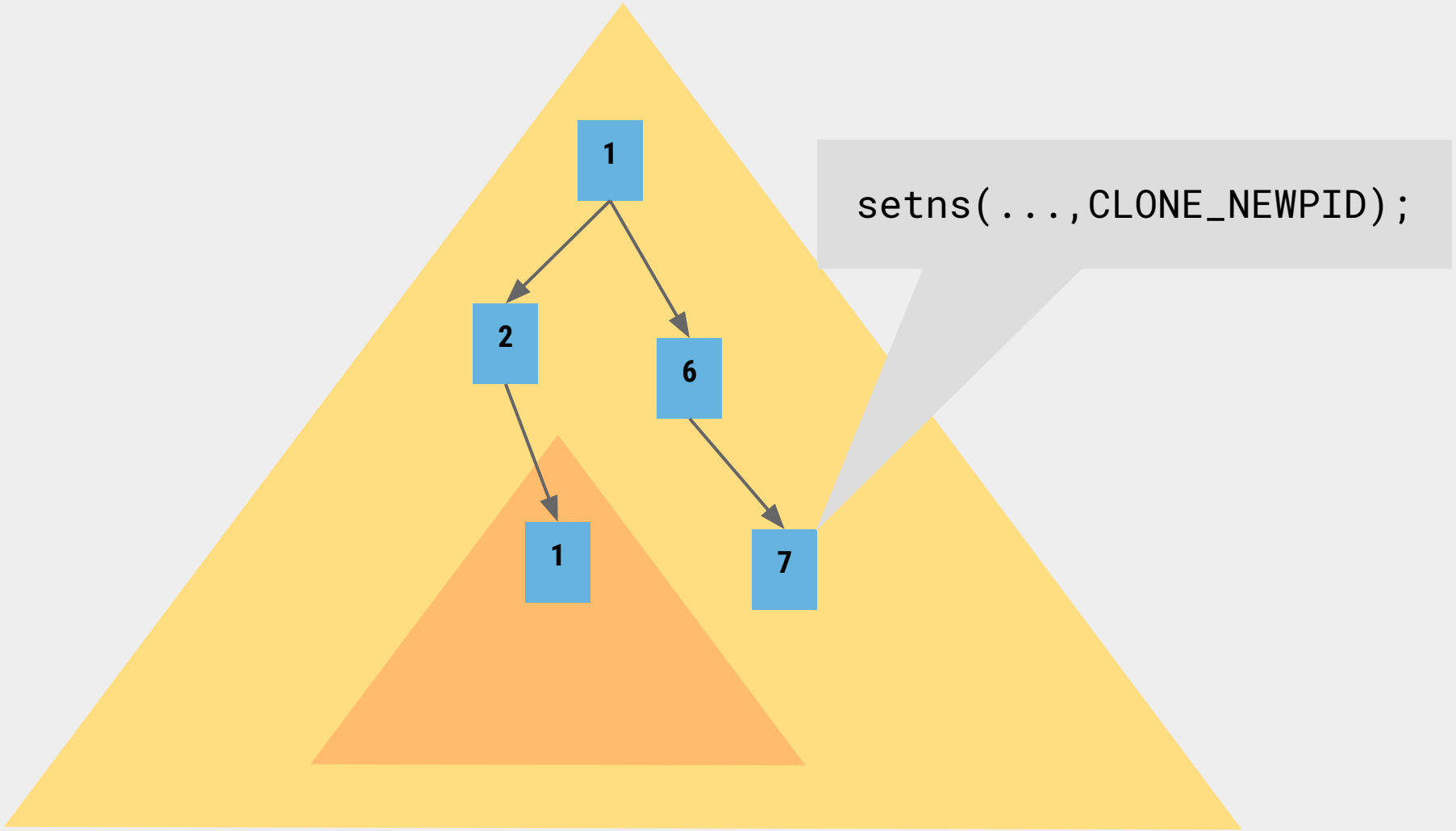
`rkt run ...`

- uses clone() to start the first process in the container with a new pid namespace
- uses unshare() to create a new network namespace

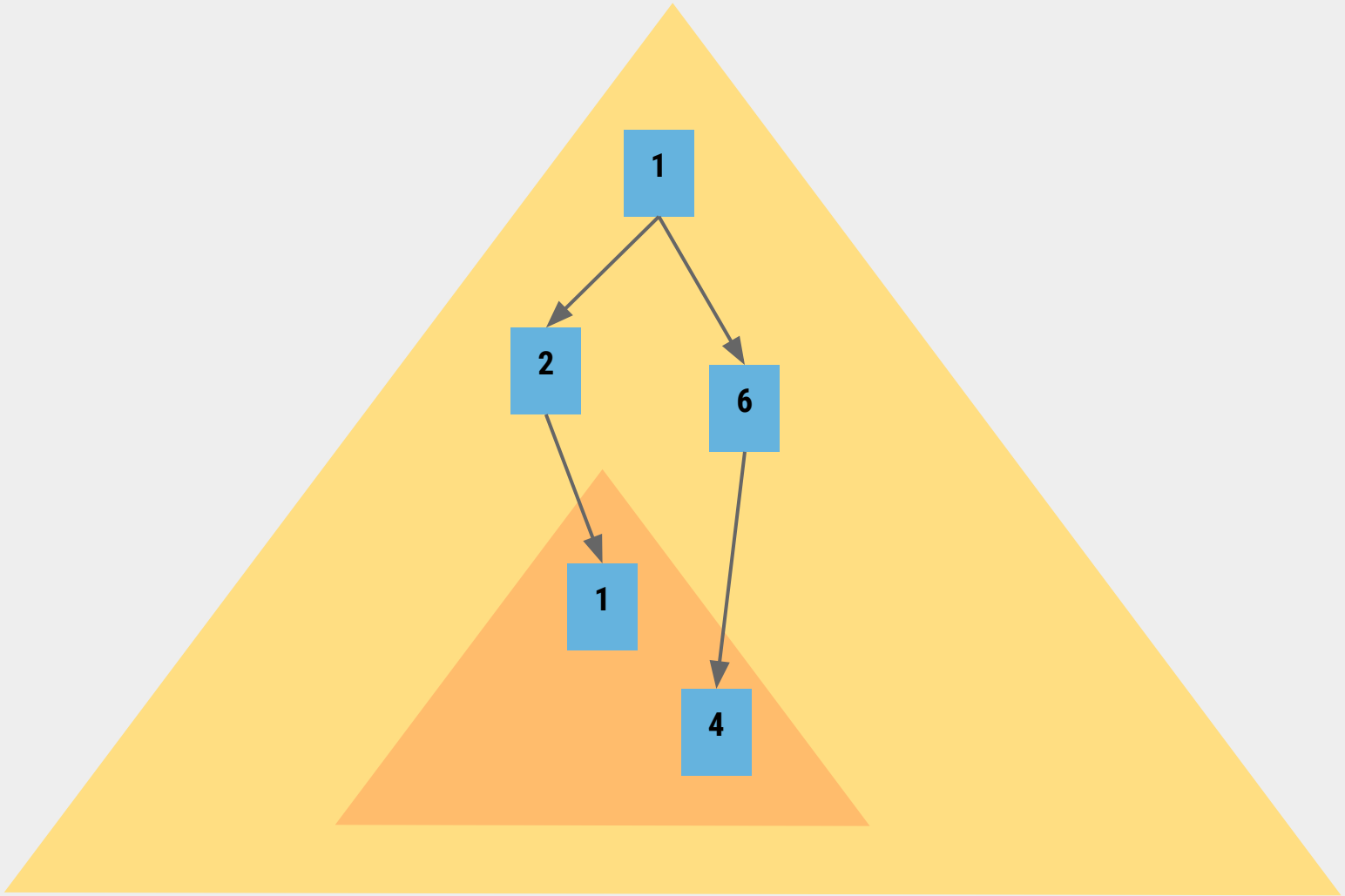
`rkt enter ...`

- uses setns() to enter an existing namespace

Joining an existing namespace

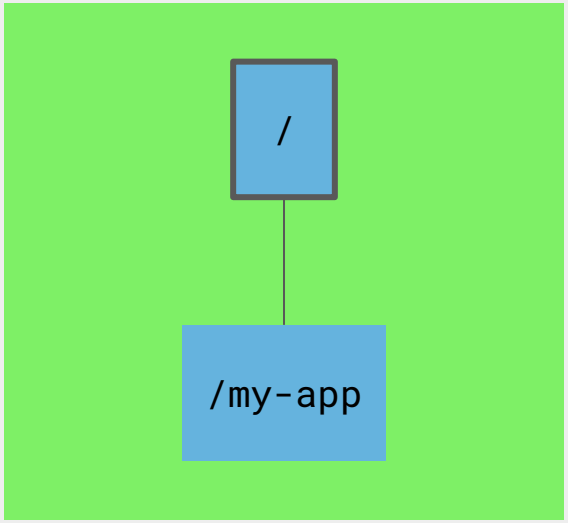


Joining an existing namespace

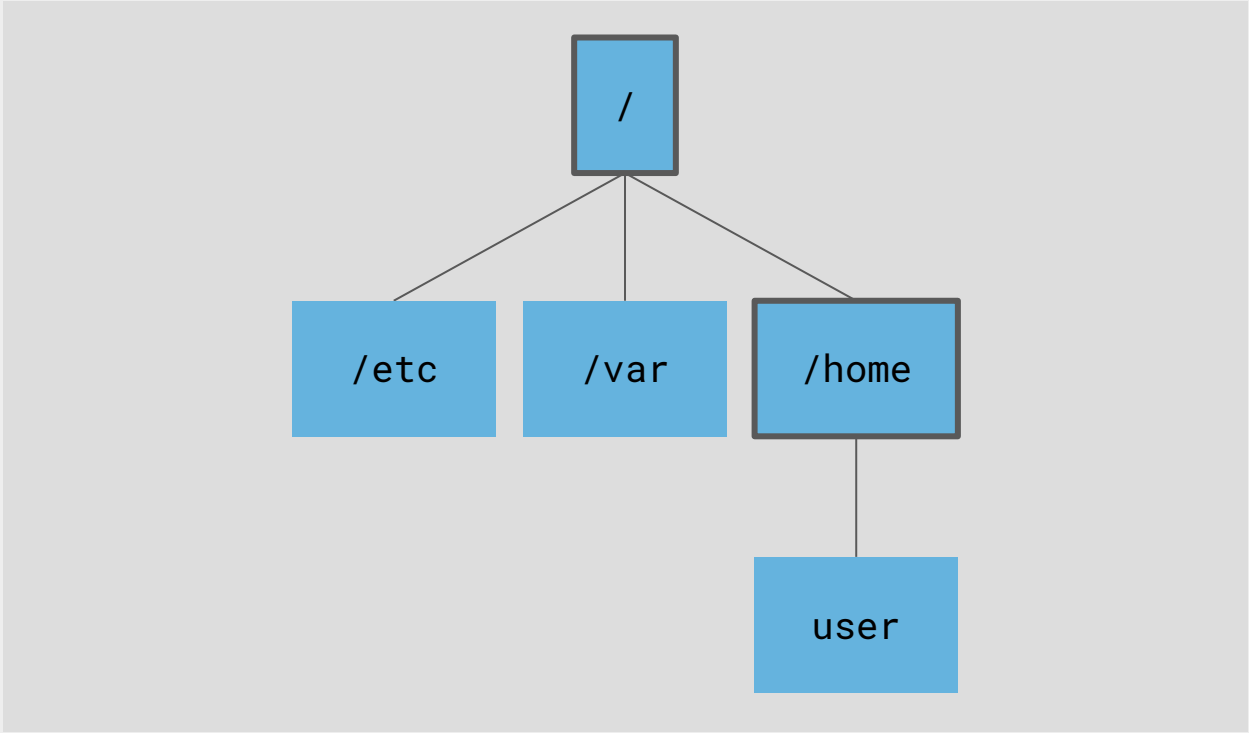


Mount namespaces

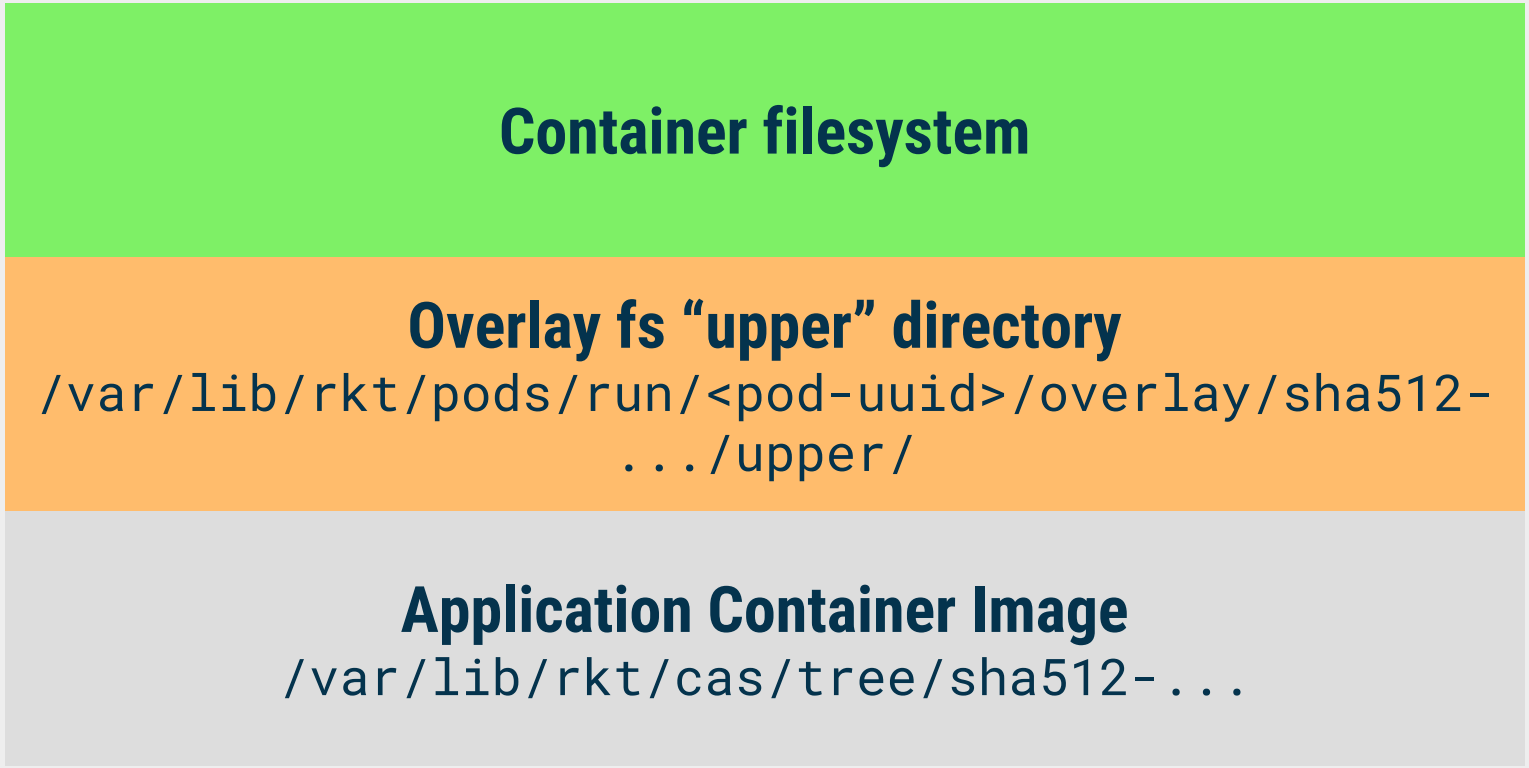
container



host

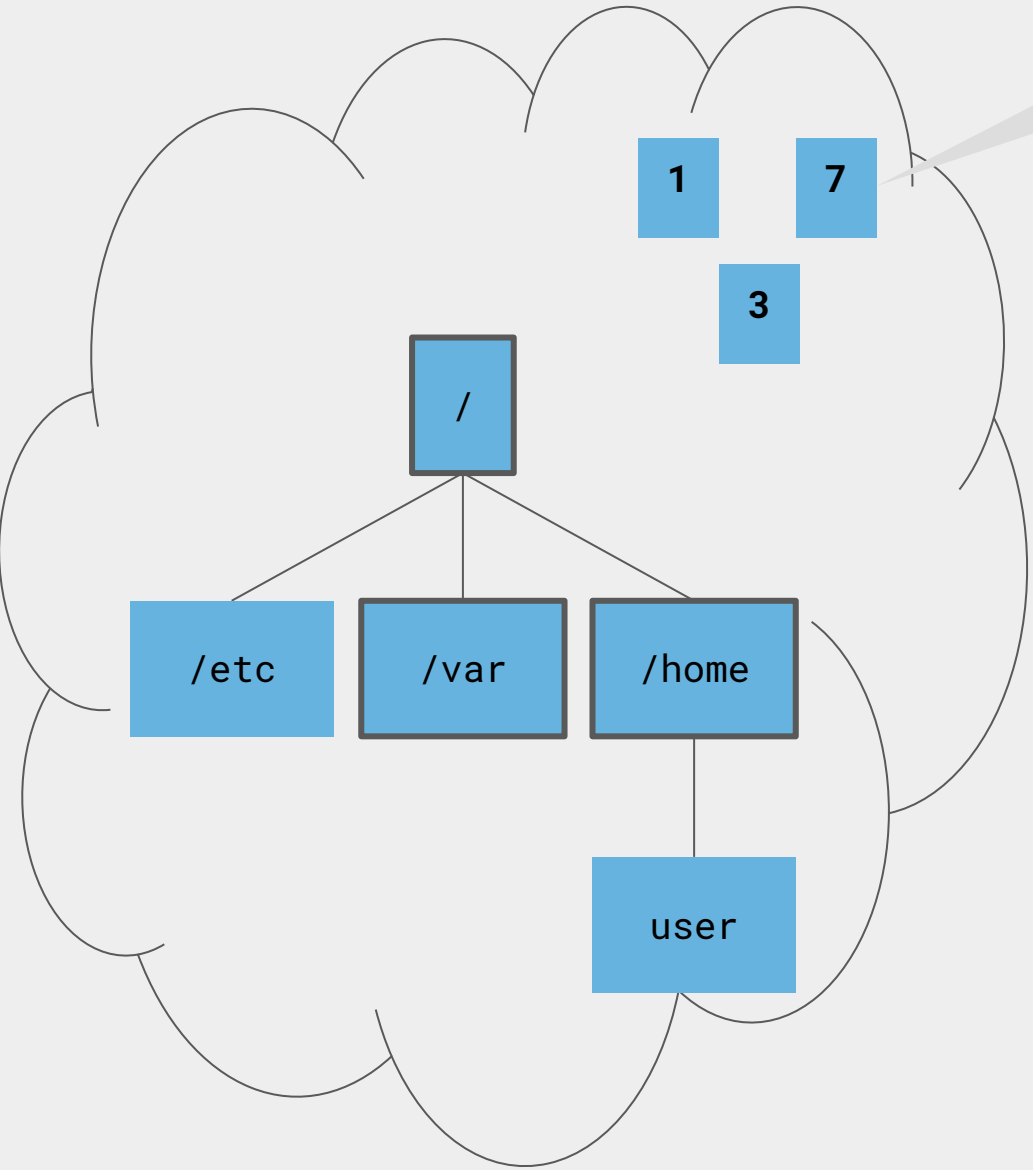


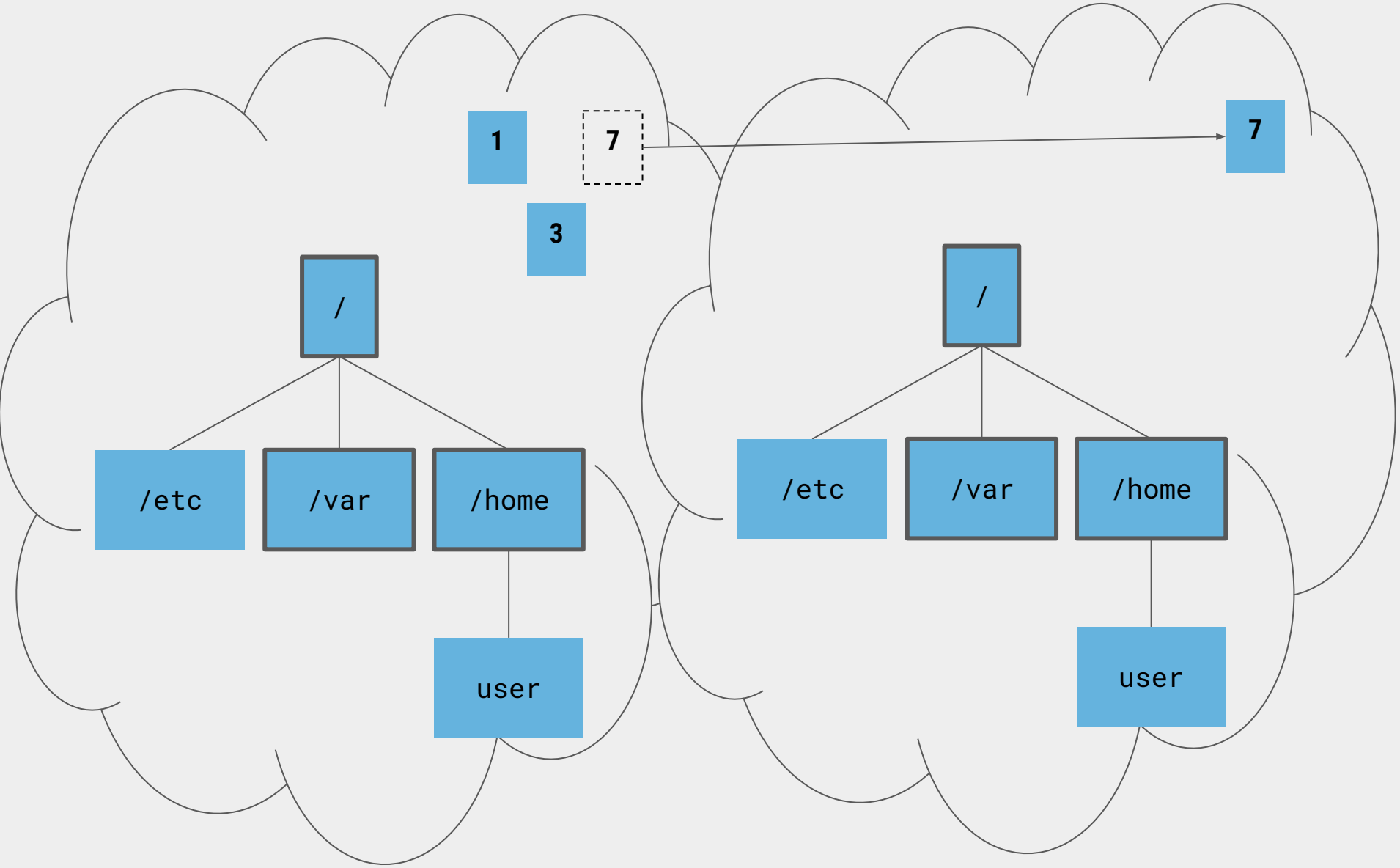
Storing the container data (Copy-on-write)



rkt directories

```
/var/lib/rkt
 ├─ cas
 │   └─ tree
 │       ├─ deps-sha512-19bf...
 │       └─ deps-sha512-a5c2...
 └─ pods
    └─ run
        └─ e0ccc8d8
            └─ overlay/sha512-19bf.../upper
            └─ stage1/rootfs/
```


unshare(..., CLONE_NEWNS);
1
7
3
/
/etc
/var
/home
user


1
7
3
7
/
/etc
/var
/home
user
/
/etc
/var
/home
user

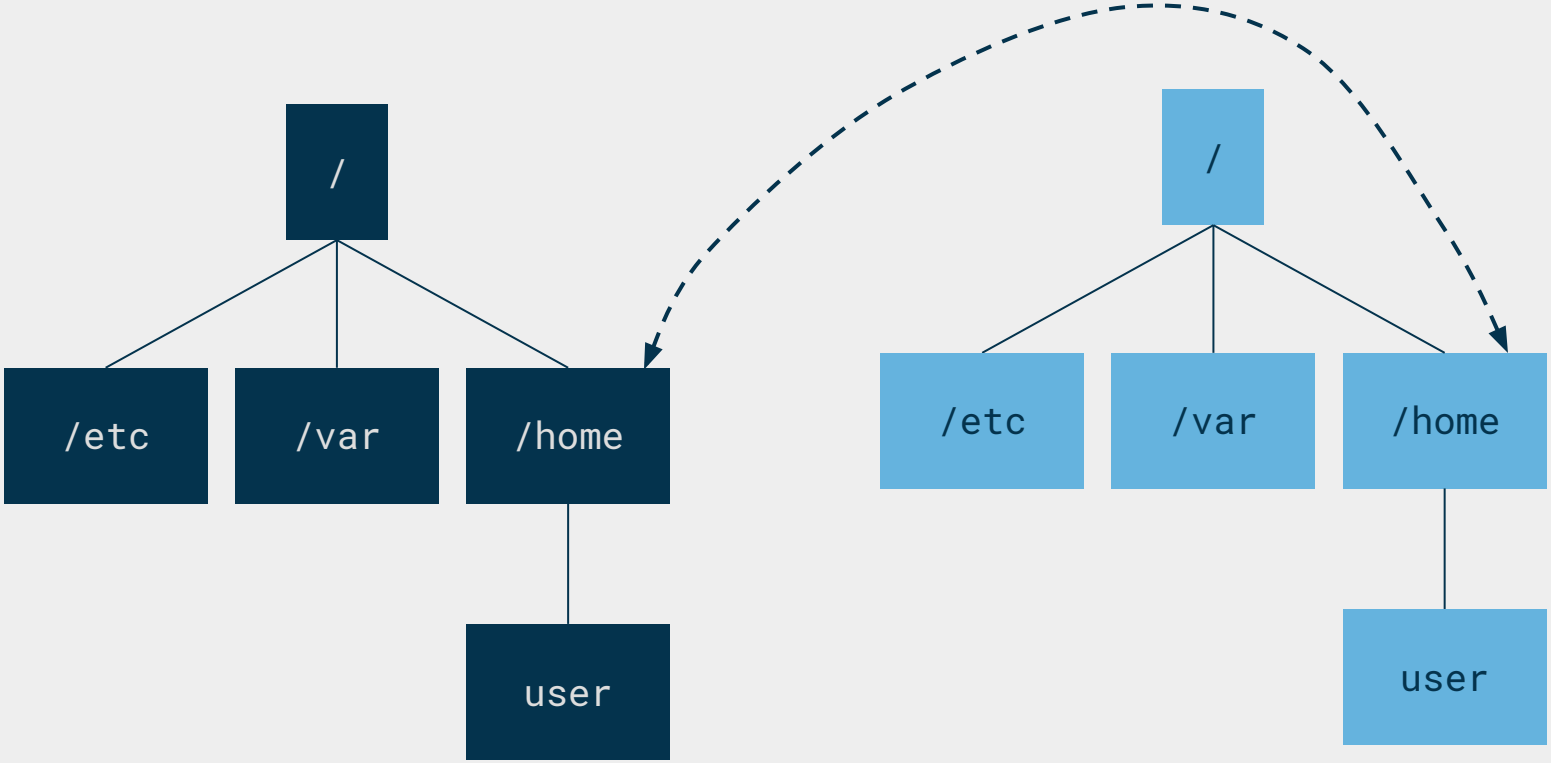
Changing root with MS_MOVE

$ROOTFS = /var/lib/rkt/pods/run/e0ccc8d8.../stage1/rootfs

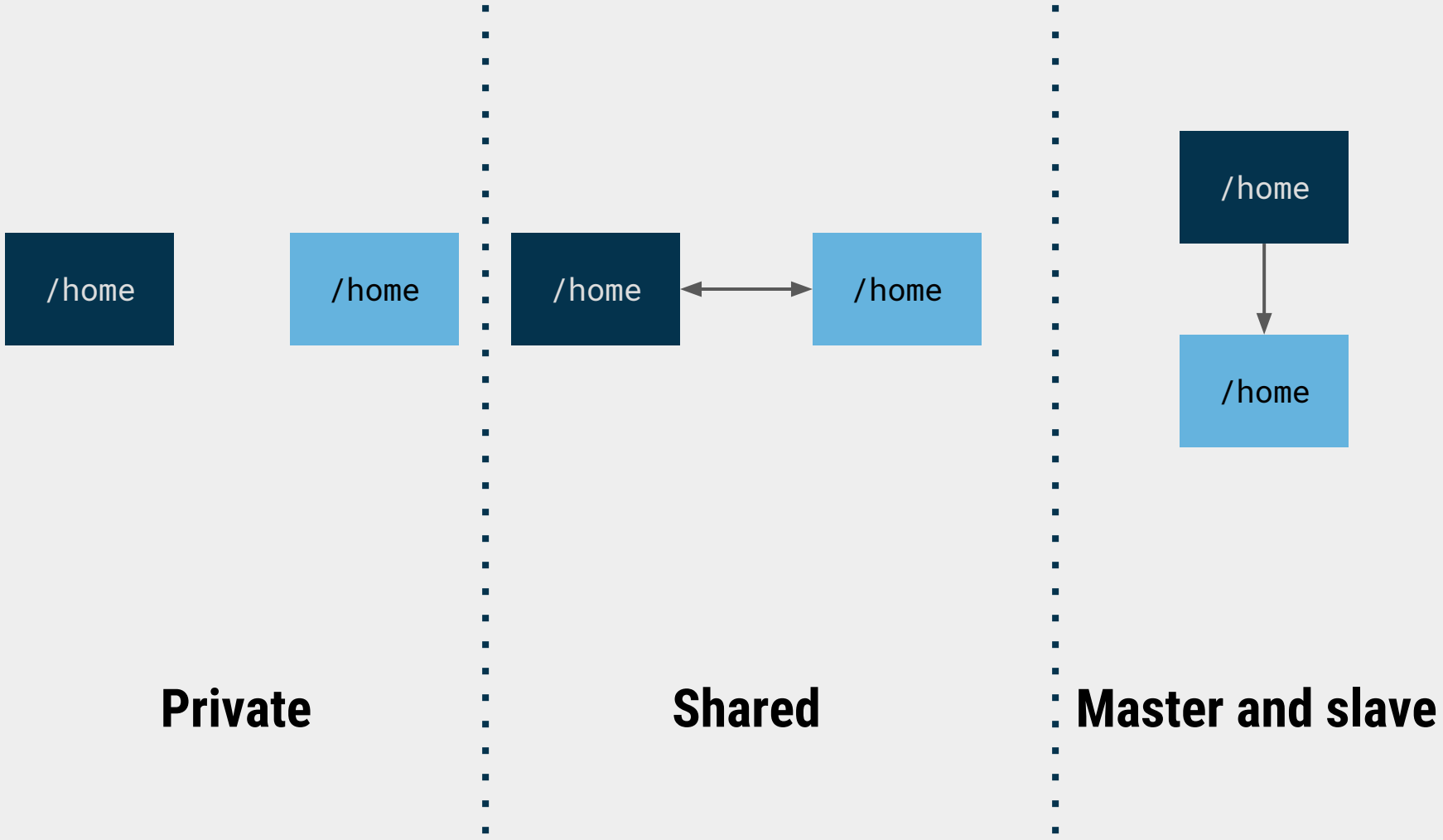
Mount propagation events

Relationship between the two mounts:

- shared
- master / slave
- private



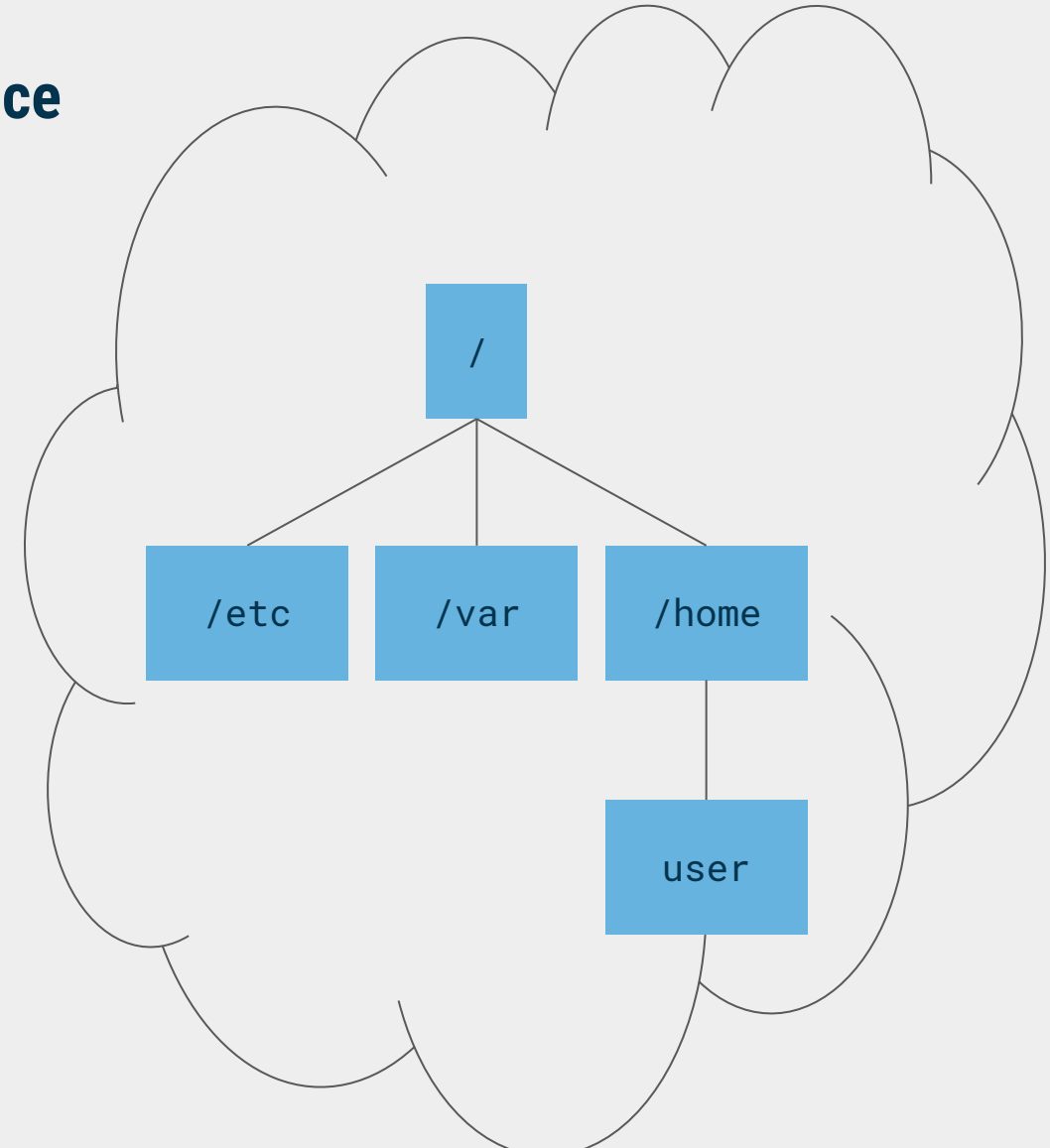
Mount propagation events



How rkt uses mount propagation events

- **/ in the container namespace is recursively set as slave:**

```
mount(NULL, "/", NULL,
      MS_SLAVE|MS_REC, NULL)
```

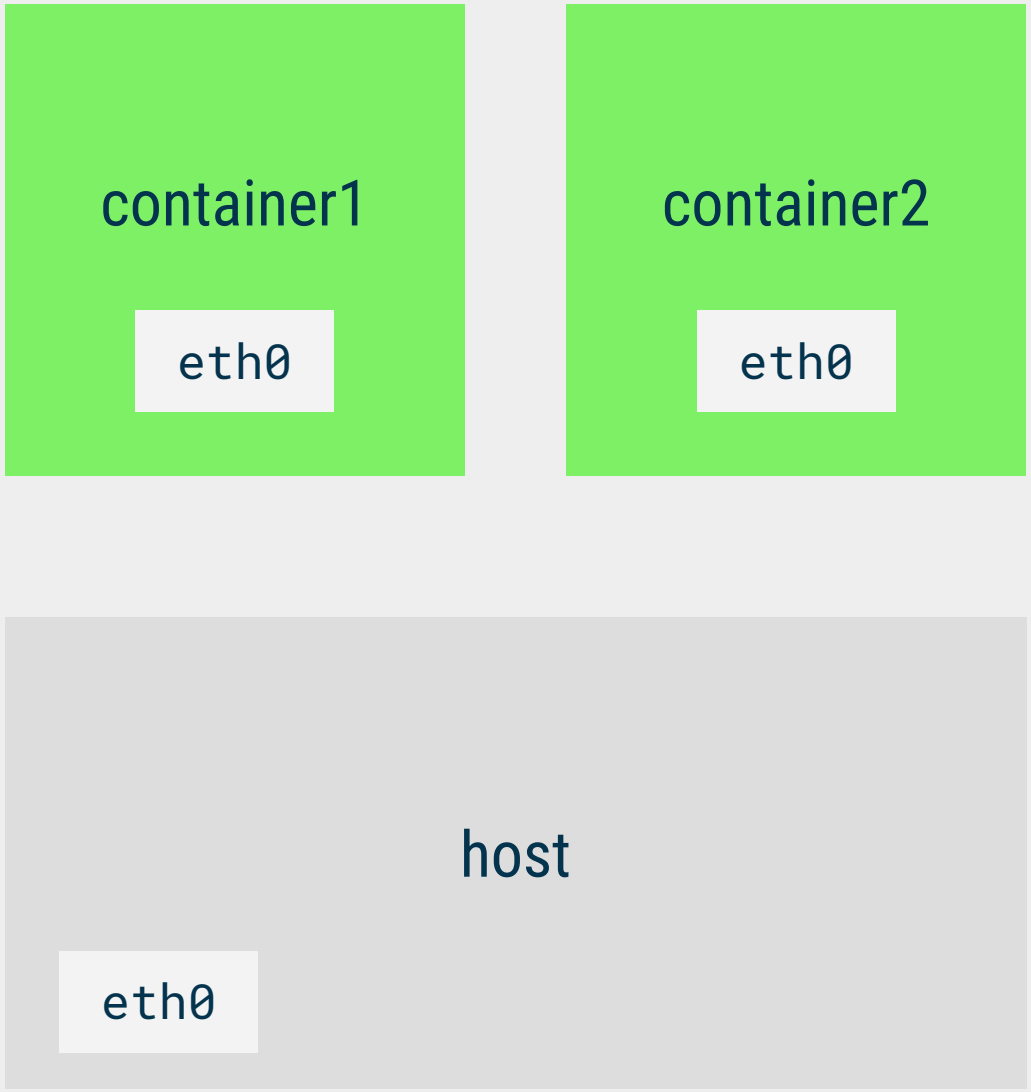


Network namespace

Network isolation

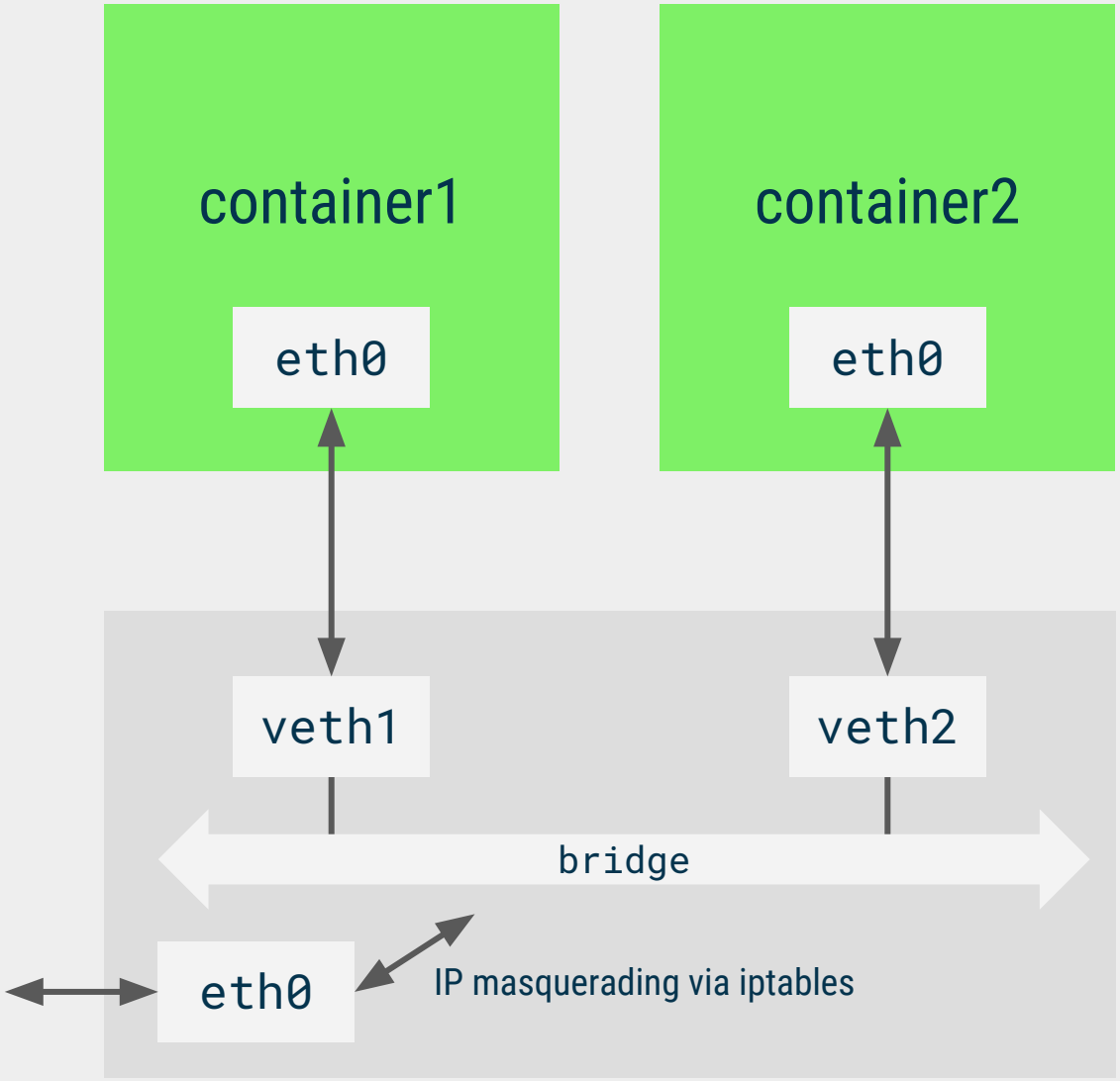
Goal:

- **each container has their own network interfaces**
- **Cannot see the network traffic outside the container (e.g. tcpdump)**



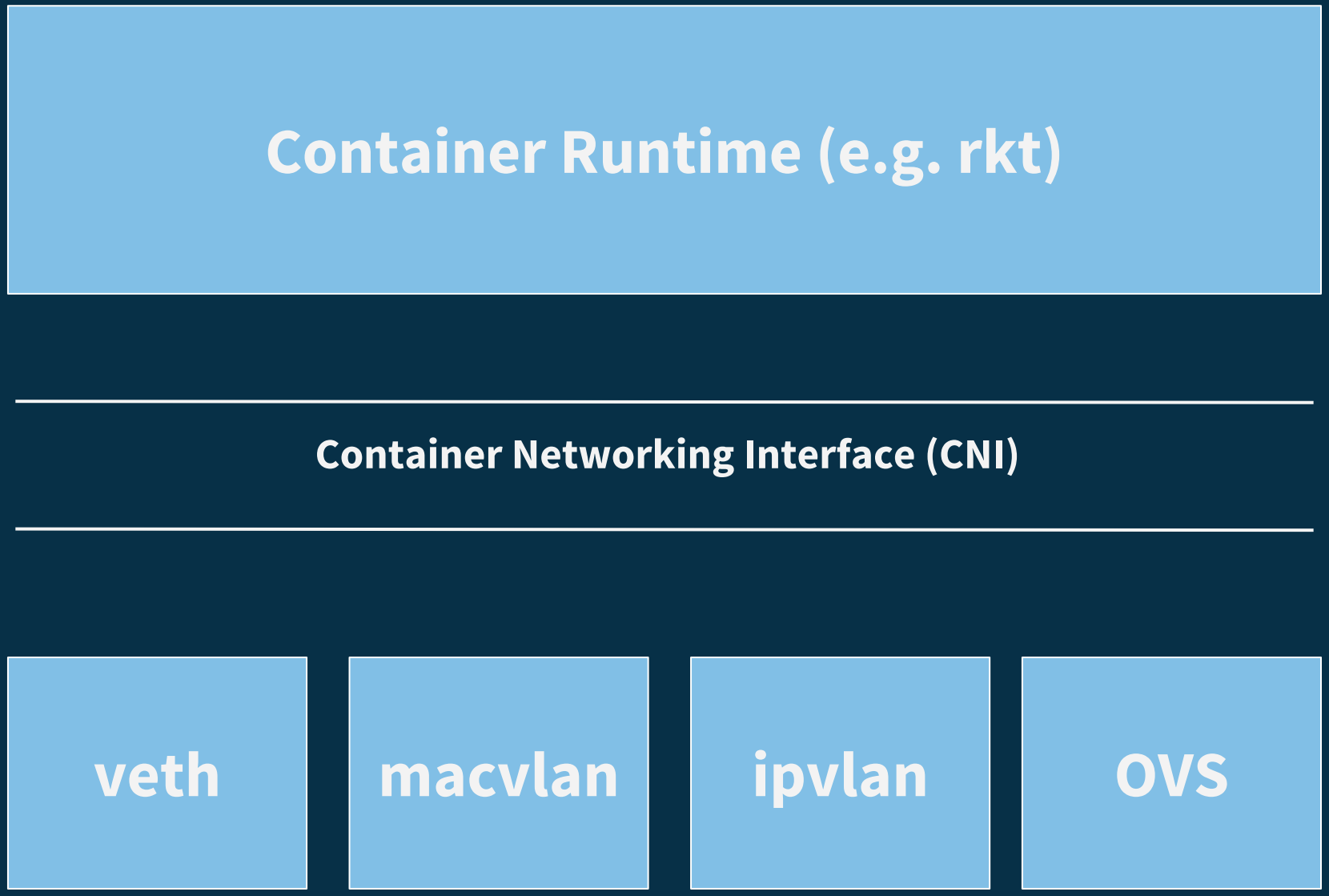
Network tooling

- Linux can create pairs of virtual net interfaces
- Can be linked in a bridge



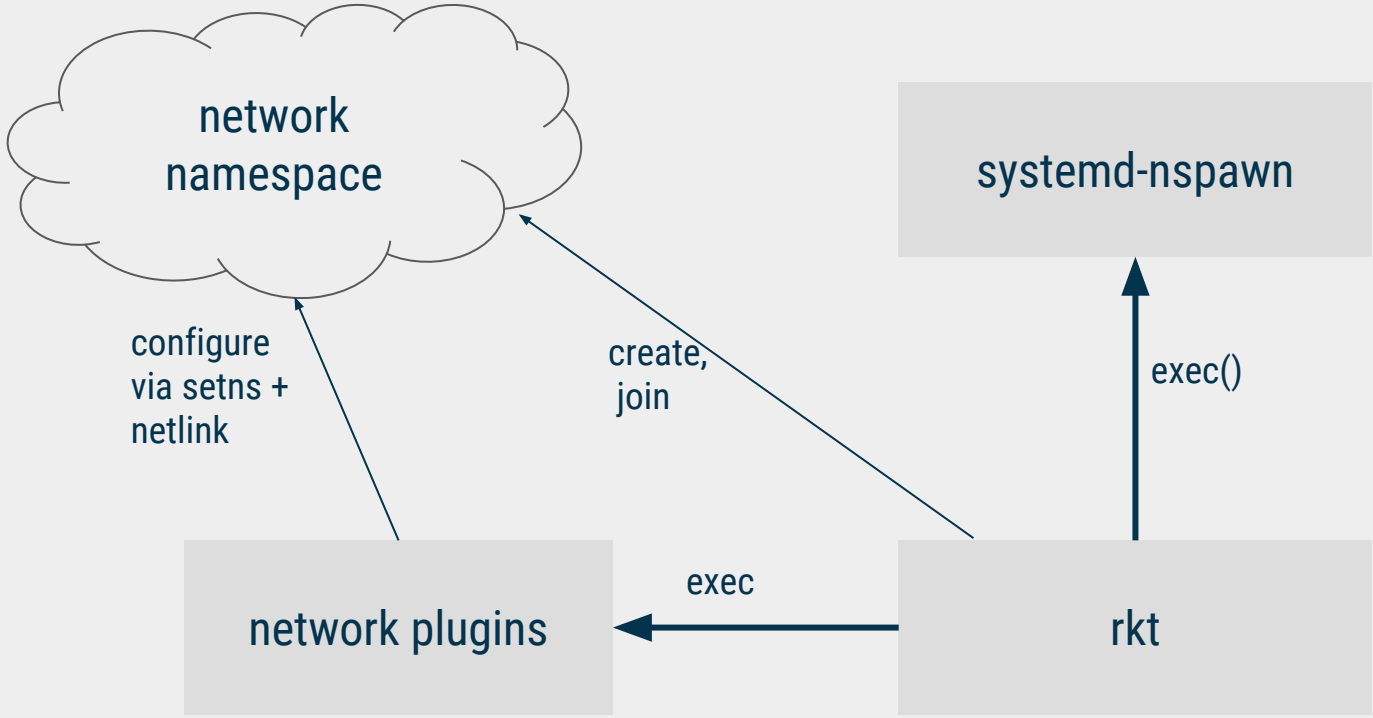
rkt networking

- plugin based
- Container Network Interface (CNI)
  - rkt
  - Kubernetes
  - Calico


Container Runtime (e.g. rkt)
Container Networking Interface (CNI)
veth
macvlan
ipvlan
OVS

How does rkt do it?

- rkt uses the network plugins implemented by the Container Network Interface (CNI, https://github.com/appc/cni)



`/var/lib/rkt/pods/run/$POD_UUID/netns`

User namespaces

History of Linux namespaces

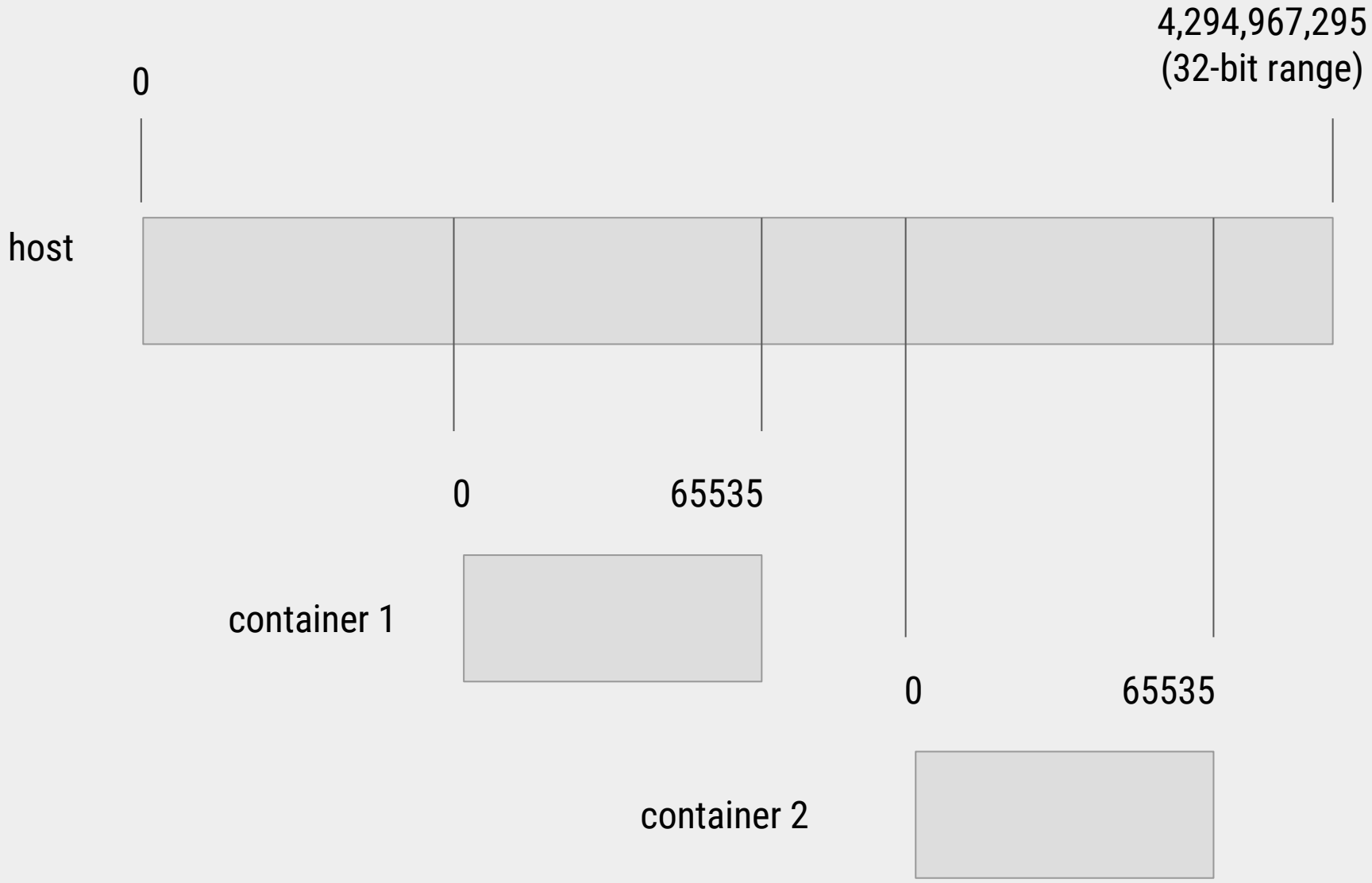
- ✓ 1991: Linux
- ✓ 2002: namespaces in Linux 2.4.19
- ✓ 2008: LXC
- ✓ 2011: systemd-nspawn
- ✓ 2013: **user namespaces** in Linux 3.8
- ✓ 2013: Docker
- ✓ 2014: rkt

… development still active

Why user namespaces?

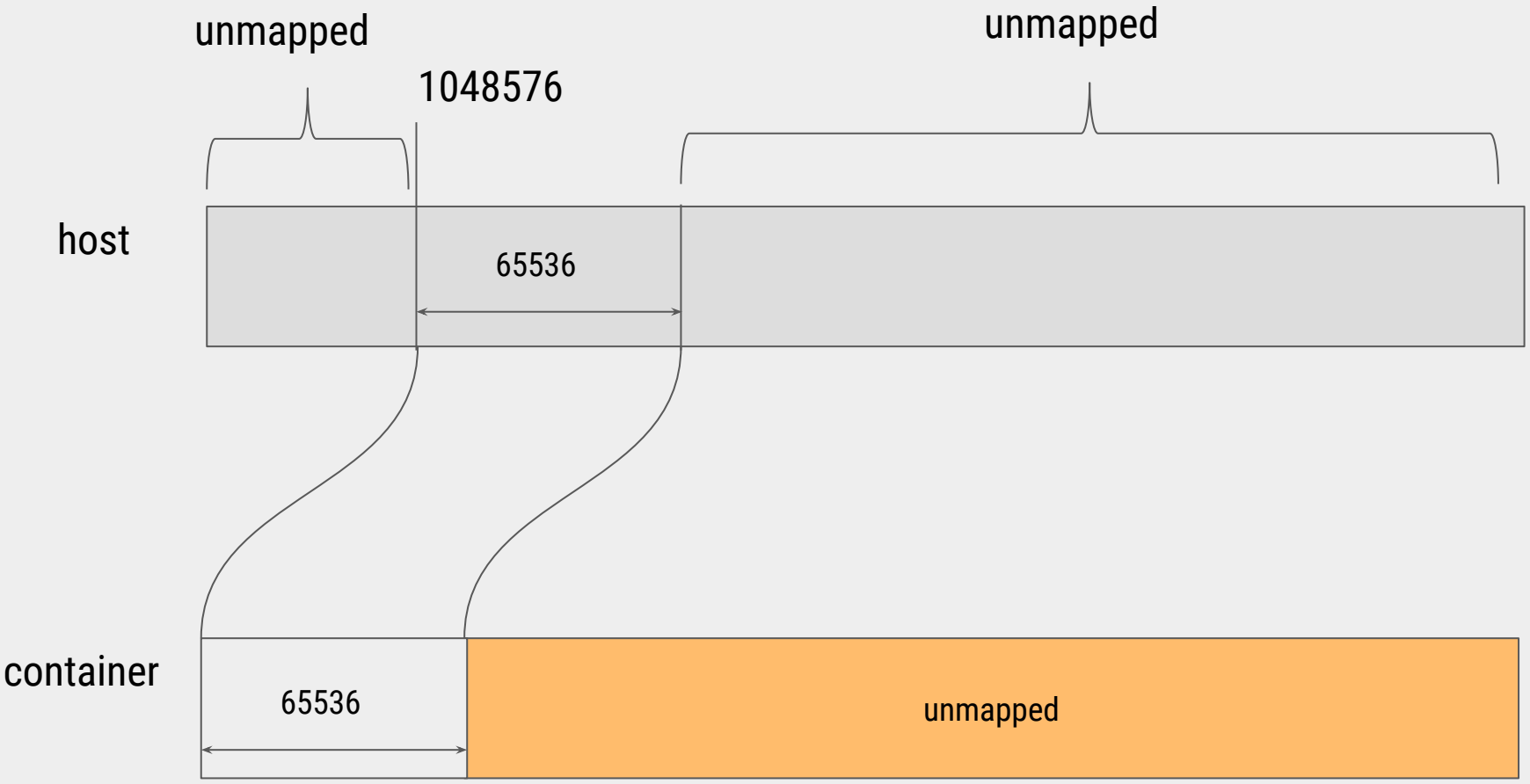
- Better isolation
- Run applications which would need more capabilities
- Per user limits
- Future:
  - Unprivileged containers: possibility to have container without root

User ID ranges

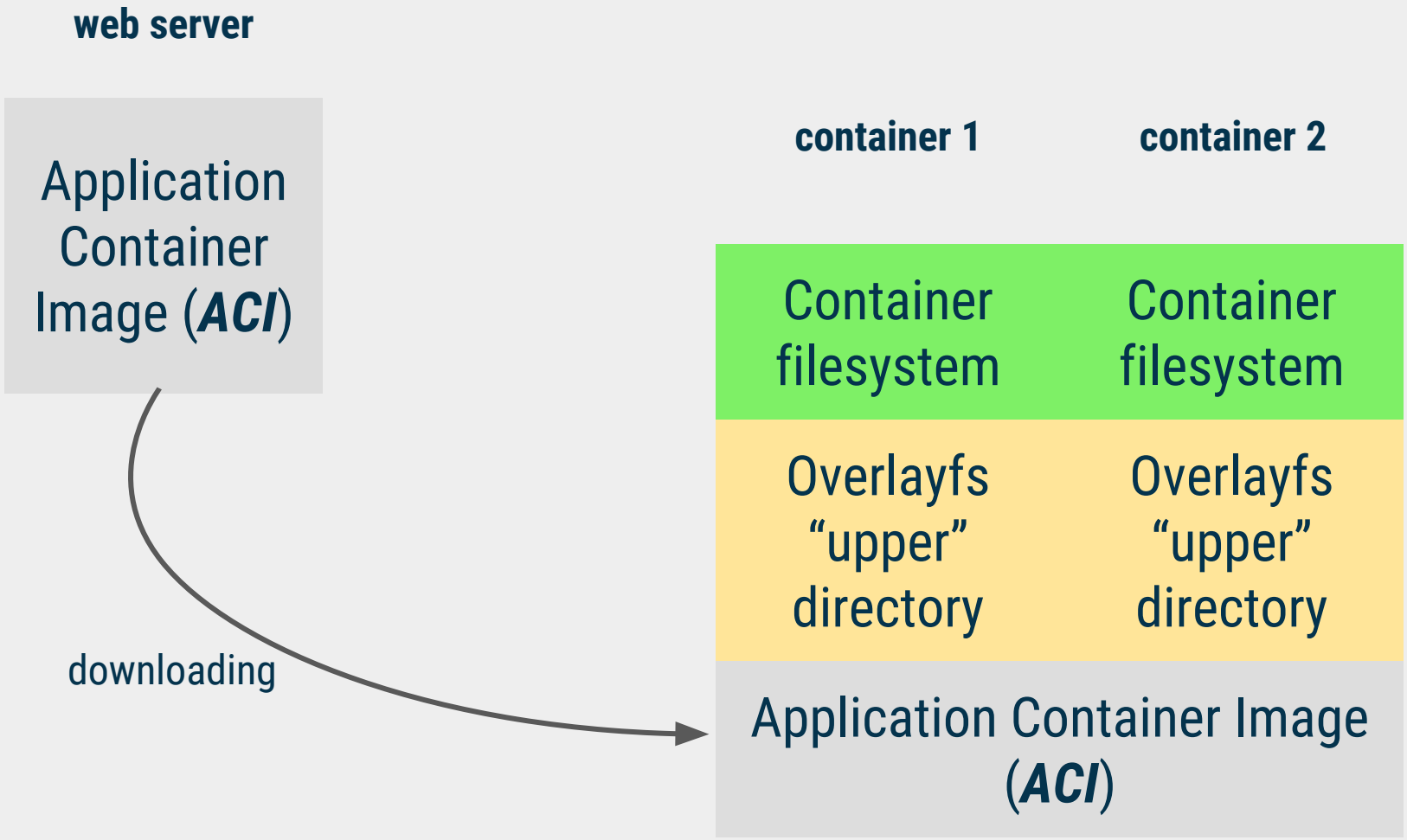


User ID mapping

```
/proc/$PID/uid_map: “0  1048576  65536”
```



Problems with container images

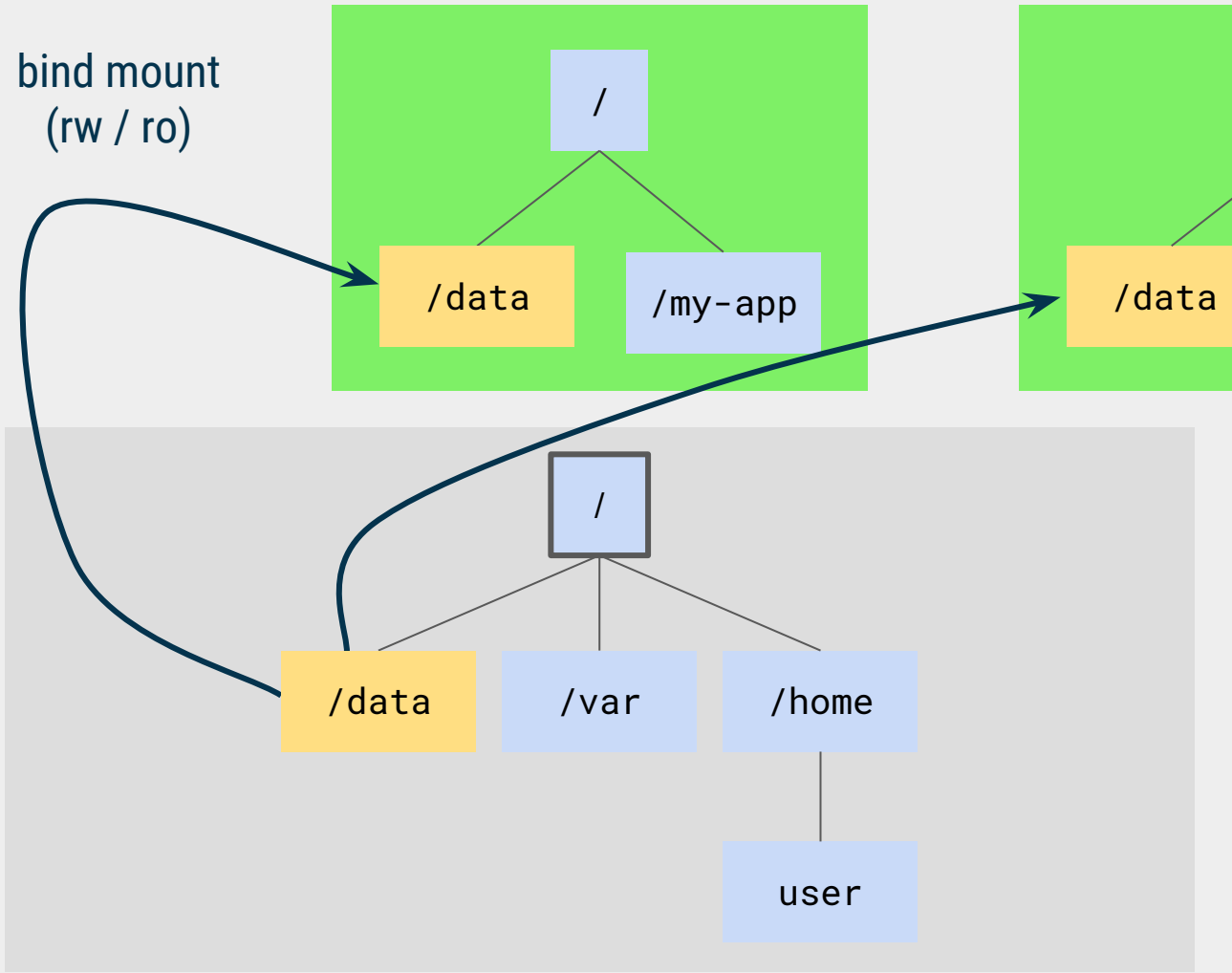


Problems with container images

- Files UID / GID
- rkt currently only supports user namespaces without overlayfs
  - Performance loss: no COW from overlayfs
  - “chown -R” for every file in each container

Problems with volumes

- mounted in several containers
- No UID translation
- Dynamic UID maps



User namespace and filesystem problem

- Possible solution: add options to mount() to apply a UID mapping
- rkt would use it when mounting:
  - the overlay rootfs
  - volumes

Isolators

Isolators in rkt

- specified in an image manifest
- limiting capabilities or resources

```
"isolators": [
    {
        "name": "resource/cpu",
        "value": {
            "request": "250m",
            "limit": "500m"
        }
    },
    {
        "name": "resource/memory",
        "value": {
            "request": "1G",
            "limit": "2G"
        }
    },
    {
        "name": "os/linux/capabilities-retain-set",
        "value": {
            "set": ["CAP_NET_BIND_SERVICE"]
        }
    }
],
```

Isolators in rkt

Currently implemented

- capabilities
- cpu
- memory

Possible additions

- block-bandwidth
- block-iops
- network-bandwidth
- disk-space

cgroups

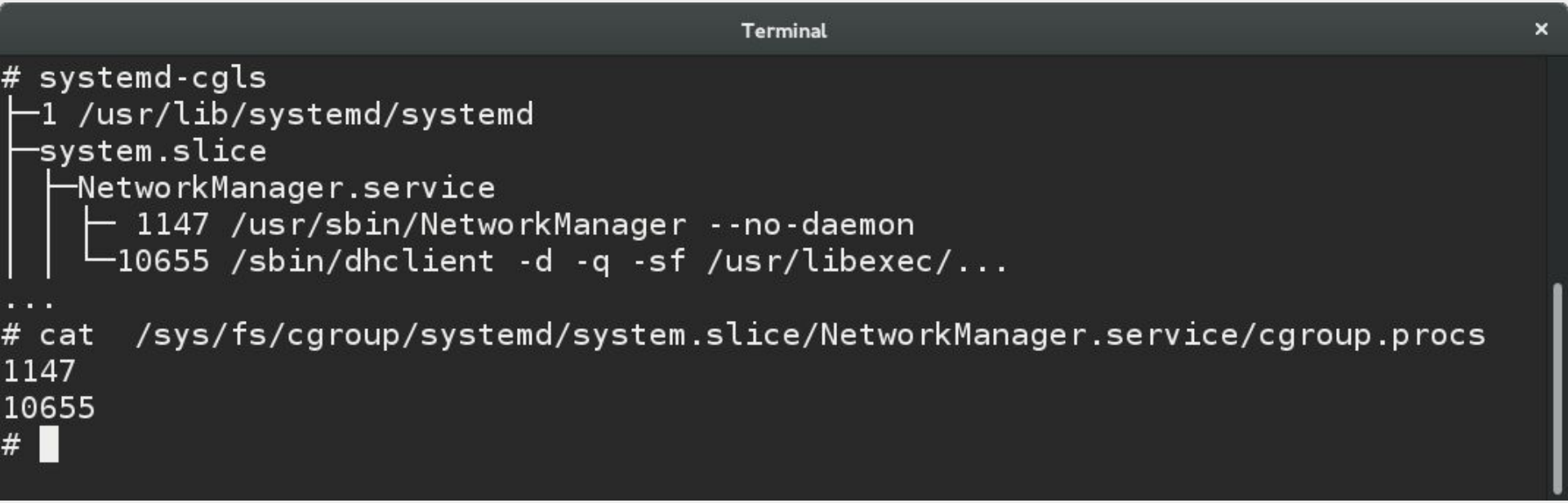
What's a control group (cgroup)

- group processes together
- organised in trees
- applying limits to them as a group

cgroup API

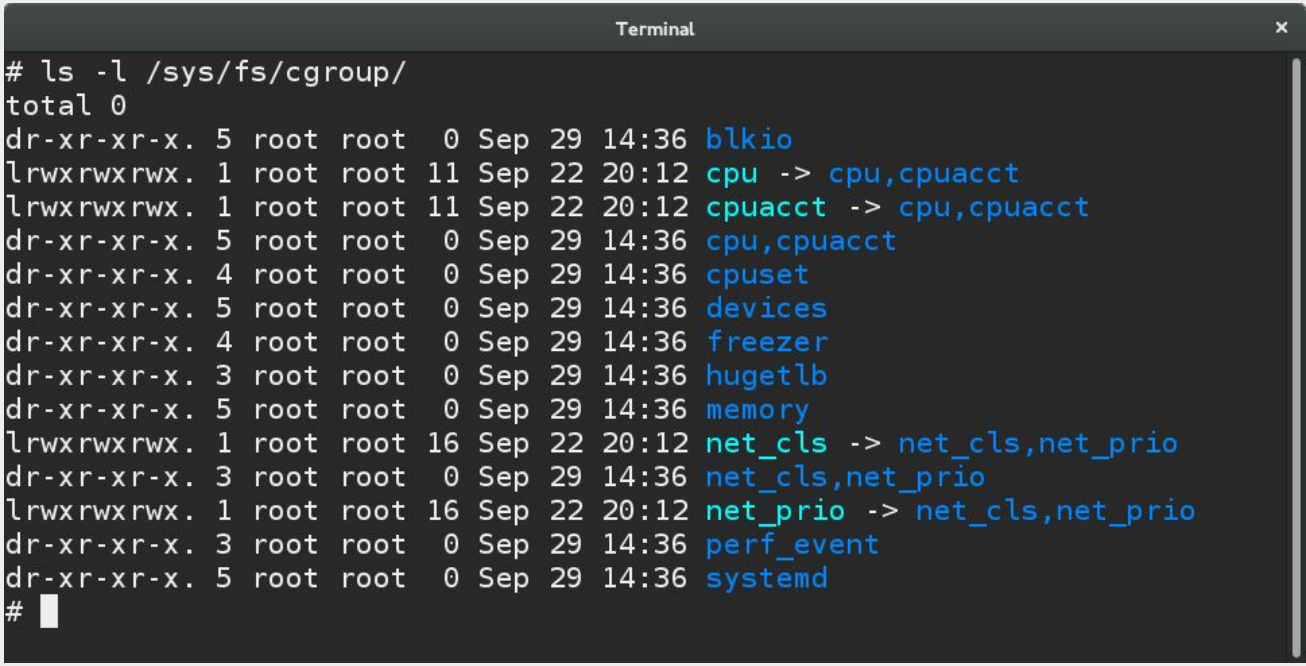
```
/sys/fs/cgroup/*/

/proc/cgroups

/proc/$PID/cgroup
```

cgroups

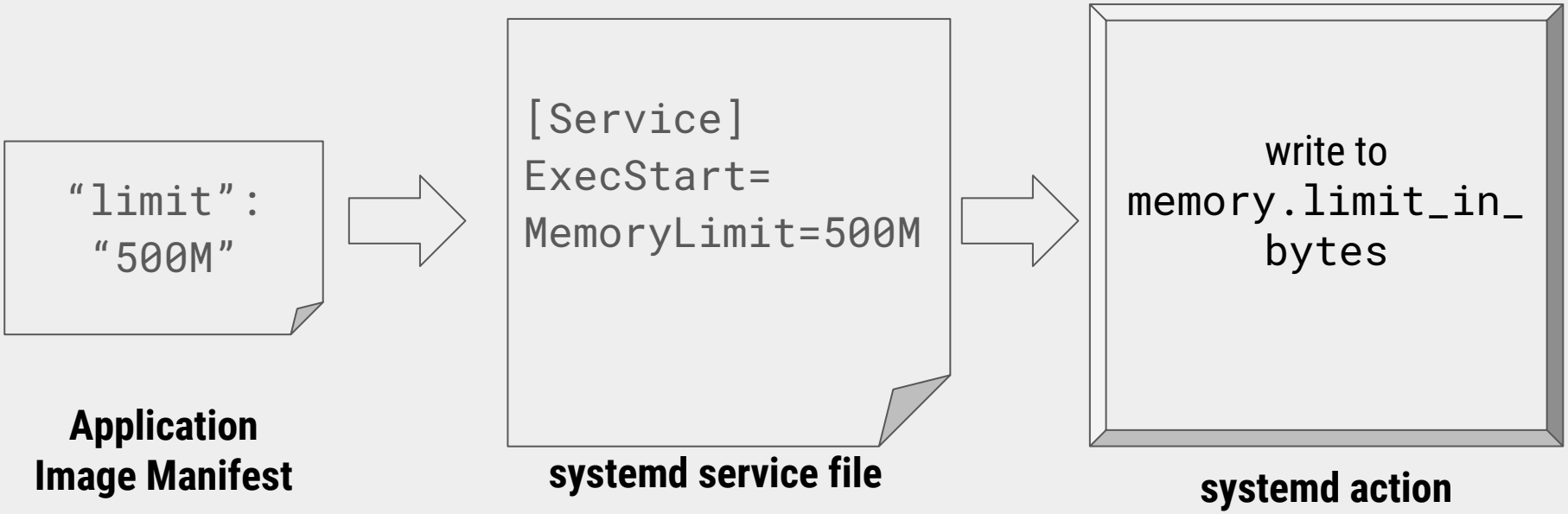

Terminal
systemd-cgls
├─1 /usr/lib/systemd/systemd
├─system.slice
│ ├─NetworkManager.service
│ │ ├─ 1147 /usr/sbin/NetworkManager --no-daemon
│ │ └─10655 /sbin/dhclient -d -q -sf /usr/libexec/...
...
cat /sys/fs/cgroup/systemd/system.slice/NetworkManager.service/cgroup.procs
1147
10655
#

List of cgroup controllers

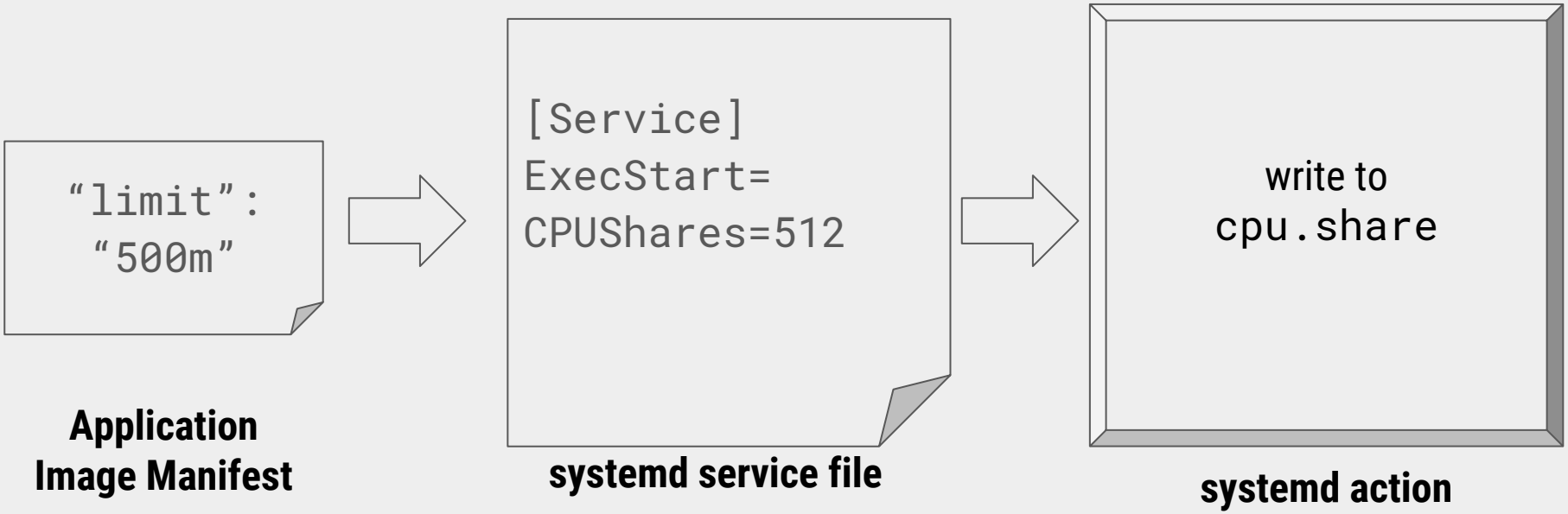
```
/sys/fs/cgroup/
  ├─ cpu
  ├─ devices
  ├─ freezer
  ├─ memory
  ├─ ...
  └─ systemd
```



Memory isolator



CPU isolator



Application Image Manifest

systemd service file

systemd action

Unified cgroup hierarchy

- Multiple hierarchies:
  - one cgroup mount point for each controller (memory, cpu...)
  - flexible but complex
  - cannot remount with a different set of controllers
  - difficult to give to containers in a safe way
- Unified hierarchy:
  - cgroup filesystem mounted only one time
  - soon to be stable in Linux (mount option "`__DEVEL__sane_behavior`" being removed)
  - initial implementation in systemd-v226 (September 2015)
  - no support in rkt yet

Questions?

Join us!

github.com/coreos/rkt

@coreosfest

coreos.com/fest

May 9 & 10, 2016 | Berlin, Germany

- Early bird tickets
- Sponsorships are still available
- Submit a talk before February 29th!