Resource Virtualization with Containers

Tanu Malik

School of Computing, DePaul University

Visiting Faculty, CSE, IIT, Delhi

Network namespaces

- A process (or group of processes) that no longer has access to all the host system’s “native” network interfaces
  - Similar to a process that has executed the chroot() system call no longer has access to the full filesystem.
- E.g. A virtual network
  - virtual Ethernet interfaces
  - virtual Ethernet links.

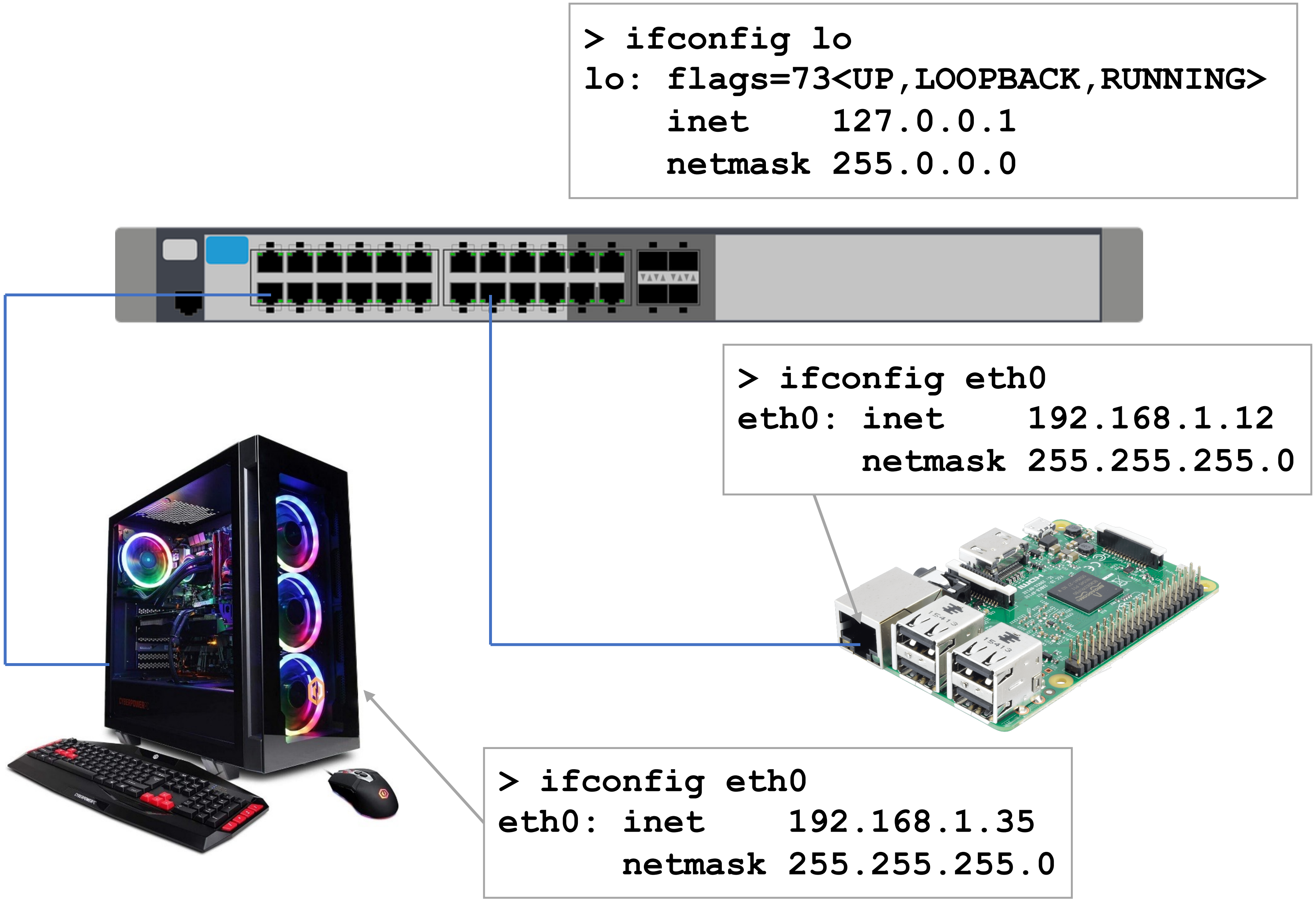
Host network stack

OSI 5-layer model of the Internet

Application (e.g. video streaming)
Transport (e.g. TCP, UDP)
Network (e.g. routing)
Link (e.g. scheduling)
Physical (e.g. OFDM)

Network interface

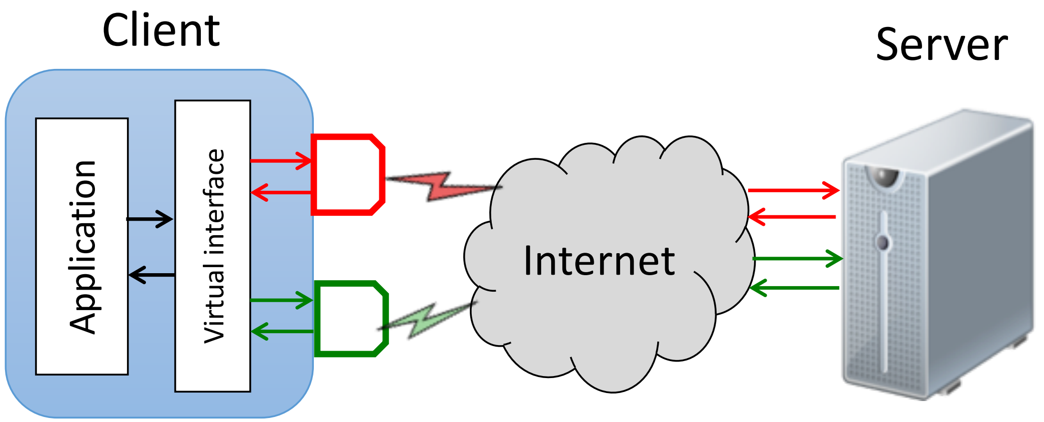
- is a hardware component, typically a circuit board or chip, which is installed on a computer so it can connect to a network
- Unique, unchangeable MAC addresses, also known as physical network addresses, are assigned to NICs.



Slide credit: CSE

Network namespaces

- A network namespace is a logical copy of the network stack from the host system.
  - Each namespace has its own IP addresses, network interfaces, routing tables, etc.
  - The default or global namespace is the one in which the host system physical interfaces exist.
- A virtual network interface (VIF) is an abstract virtualized representation of a computer network interface that may or may not correspond directly to a network interface controller.



Uses of network namespaces

- Isolate processes from the network
- Secure network applications:
  - A process with a socket connection clone()s into a new network namespace
  - Child inherits socket file descriptor but establish other network connections
  - Instead of clone()ing, a networked process can send a socket fd to an isolated process via a UNIX socket
- Create virtual network devices, e.g. containers or virtual machines that appear as separate devices on the network

Network namespaces

- Network namespace management: ip-netns
- Network namespaces enable isolation of network resources

```
ip netns add ns1
```

  - Creates a new network napespace
- By default, a process inherits its network namespace from its parent.
  - Initially all the processes share the same default network namespace from the init process.

- Creates a named bind mount:

```
/var/run/netns/ns1
```

- This allows the network namespace to persist without processes
- Allows setup and manipulation of the namespace before processes are launched

Network namespaces have no communication

- Even local loopback must be explicitly enabled!

Execute command in namespace

```
ip netns exec ns1 bash
ip link set dev lo up
```

Enable namespace's loopback interface

Validate: `ip netns exec ns1 ip address`

Test: `ip netns exec ns1 ping 127.0.0.1`

Can also run command directly, e.g.:

```
ip netns exec ns1 ip link set dev lo up
```

Network Namespaces

- We can create virtual network interfaces to connect container to host

ip link add veth0_1 type veth peer name veth1_0

- Establishes two virtual ethernet ports, connected by a virtual cable

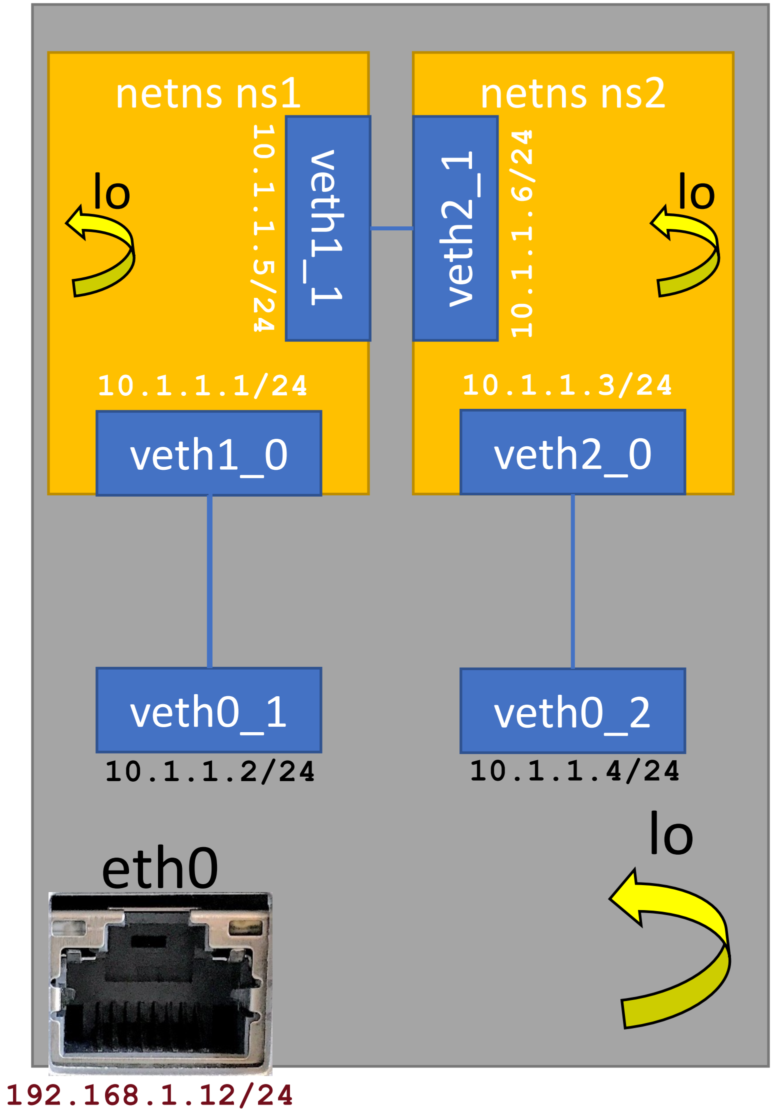
ip link set veth1_0 netns ns1

- assign the virtual device to your namespace

ip netns exec ns1 ifconfig veth1_0 10.1.1.1/24 up

ifconfig veth0_1 10.1.1.2/24 up

- Can similarly connect two containers
- All veth interfaces are on the same subnet, allowing communication between both containers and the host
- This seems inefficient …Why?



Network Namespaces

- We can create virtual network interfaces to connect container to host

ip link add veth0_1 type veth peer name veth1_0

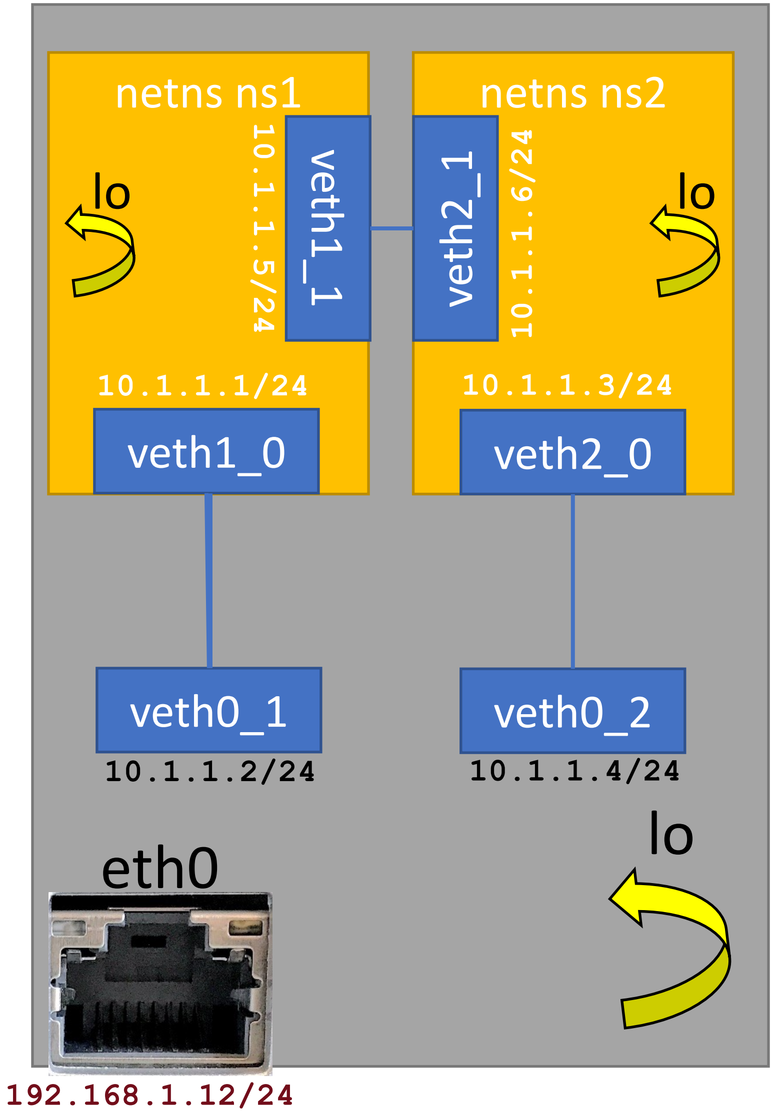
- Establishes two virtual ethernet ports, connected by a virtual cable

ip link set veth1_0 netns ns1

ip netns exec ns1 ifconfig veth1_0 10.1.1.1/24 up

ifconfig veth0_1 10.1.1.2/24 up

- Can similarly connect two containers
- All veth interfaces are on the same subnet, allowing communication between both containers and the host
- This seems inefficient … for n containers, we need $2*\binom{n+1}{2}$ virtual interfaces
- Is there a better way?
- Question: if we have several physical devices, how do we connect them?

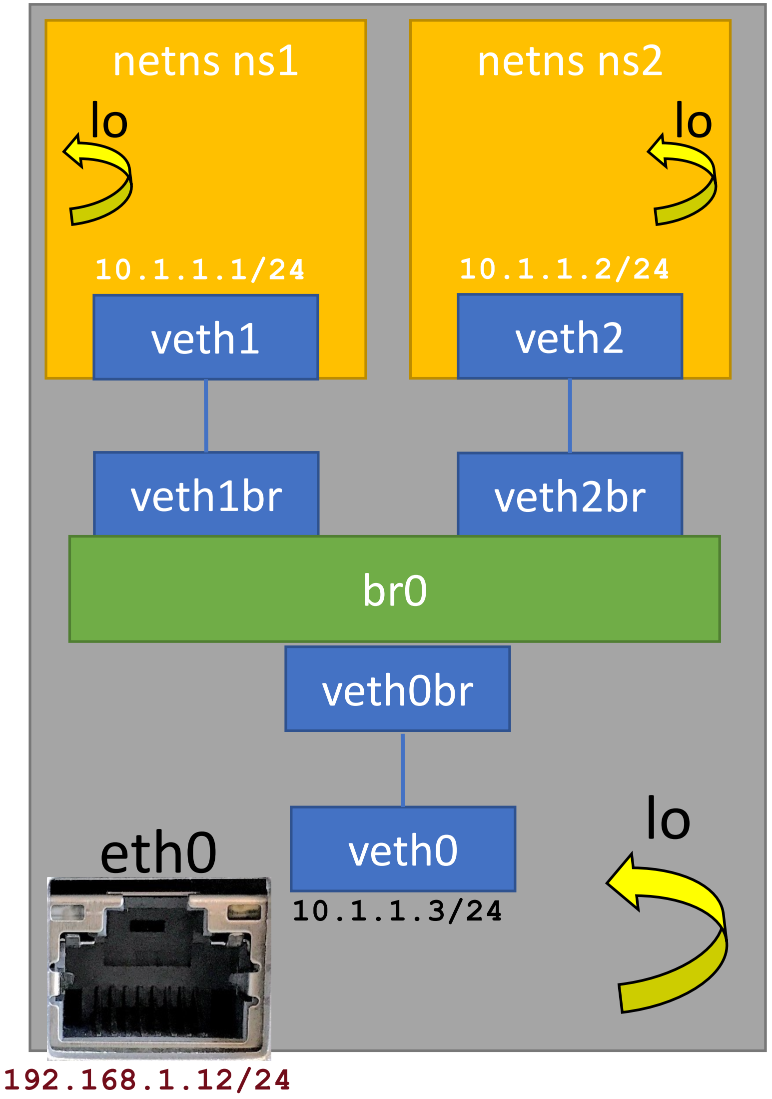


Network namespace bridges

- Answer: we use a switch to connect devices!
- A ***veth*** is like a virtual ethernet port
- A ***bridge*** is like a ***virtual switch***

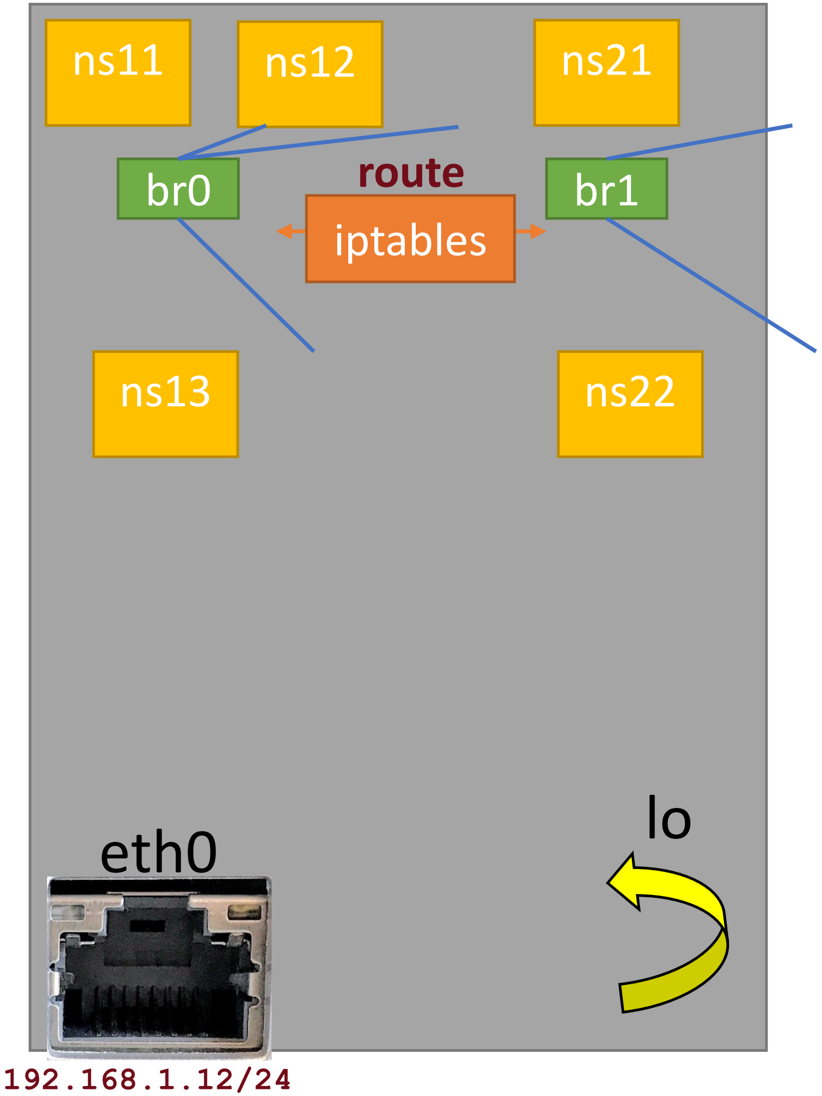
```
ip link add name br0 type bridge
ip link set br0 up
ip link set veth1br master br0
```

- Now for n containers, we need 2(n+1) veths, 1 bridge



Network namespaces

- Q: How can we create multiple, isolated networks of containers?
- A: Use multiple ***bridges***
- Q: How can we enable communication between these networks?
- A: Connect them via **route(8)** rules
- Use **iptables(8)** rules to restrict traffic between networks based on port, IP, etc.

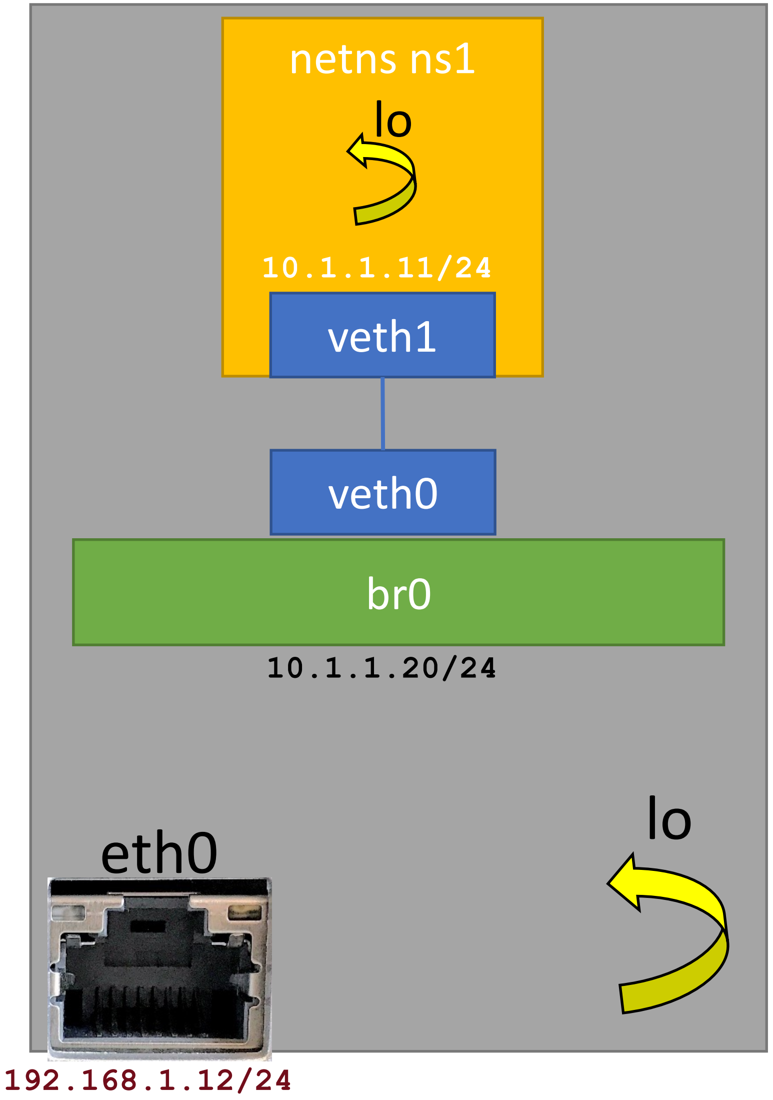


Connecting to outside

How can a container reach the outside world?

Host network address translation (NAT) with a veth as a gateway

- Add a route from ns1 to outside networks using veth0 as the gateway
```
ip netns exec ns1 \
ip route add default via 10.1.1.10
```
- Enable IP traffic forwarding
```
cat /proc/sys/net/ipv4/ip_forward
```
- Enable NAT so traffic from the ns1 subnet appears to come from the host subnet
```
iptables --table nat -A POSTROUTING \
-s 10.1.1.0/24 -j MASQUERADE
```
- Allow incoming and outgoing traffic to be forwarded over veth0
```
iptables -A FORWARD -i veth0 -j ACCEPT
iptables -A FORWARD -o veth0 -j ACCEPT
```



Connecting from outside

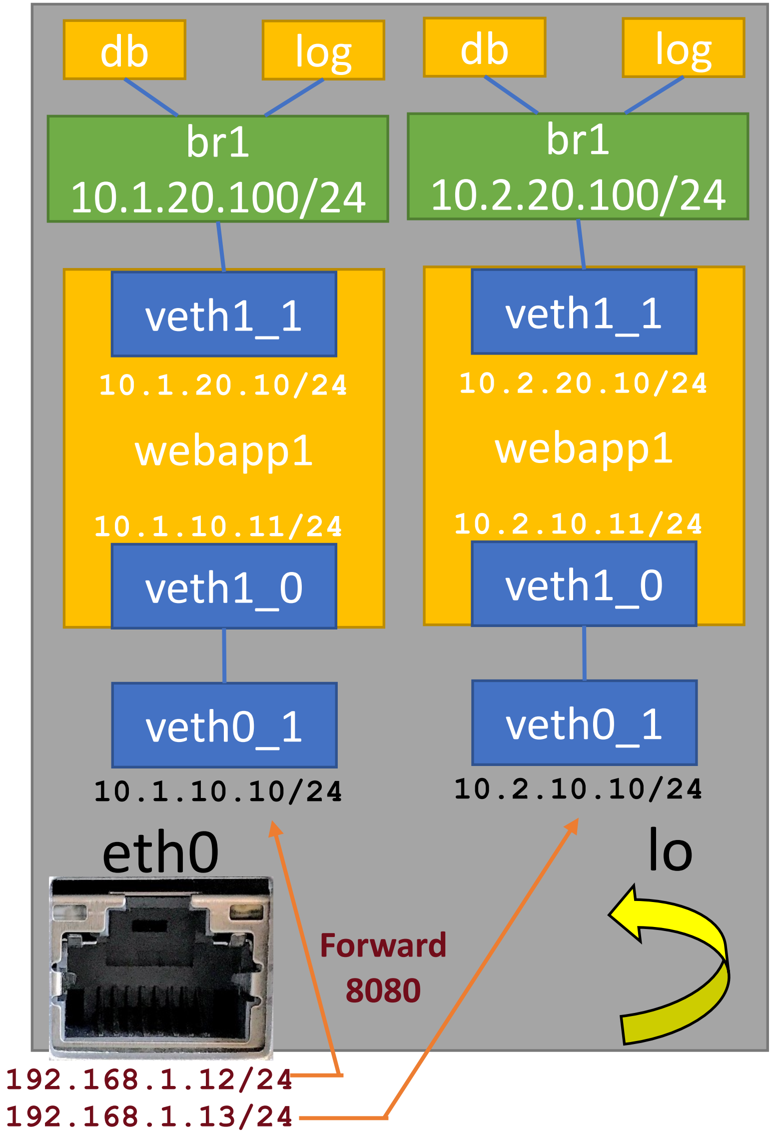
What if the container hosts a service that needs to be accessible from the outside world?

Port forwarding

- **iptables** can be used to forward inbound traffic on a specified port to a container
- The physical network interface can be provided multiple IP addresses
- Use port forwarding rules to forward traffic to different containers based on requested IP
- Useful for multiple containers providing services on the same port

Complex Network Topologies

- Putting this all together enables composition of complex container networks
- Consider a container running a web application on port 8080
- The web application uses a database server and log server
- A second web application, on the same port, is added
- We can assign a second address to eth0
- Then forward it with **iptables** to the second application


db
log
db
log
br1
10.1.20.100/24
br1
10.2.20.100/24
veth1_1
10.1.20.10/24
webapp1
10.1.10.11/24
veth1_0
veth1_1
10.2.20.10/24
webapp1
10.2.10.11/24
veth1_0
veth0_1
10.1.10.10/24
veth0_1
10.2.10.10/24
eth0
lo
Forward
8080
192.168.1.12/24
192.168.1.13/24

Cgroups

cgroups

- cgroups provides a mechanism for managing resources of a group of processes
- System Resources: CPU time, memory, disk, and network bandwidth
- ls -l /sys/fs/cgroup/systemd/

How are cgroups used?

- Consider a datacenter with
  - >100,000 servers
  - Many thousands of services
  - Want to limit failure domains

Sample workload of a famous website

- Core workload
  - Web requests
- Non-core services
  - Metric collection
  - Cron jobs
  - Chef
  - atop (logging mode)
- Ad-hoc querying/debugging
  - tcpdump
  - atop

Limits on the workload

- Core workload → Essentially unlimited
  - Web requests
- Non-core services → Memory limit: 1GiB, IO write: 1MBps
  - Metric collection
  - Cron jobs
  - Chef
  - atop (logging mode)
- Ad-hoc querying/debugging → Mem limit: 2GiB Max tasks: 1000
  - tcpdump
  - atop

Cgroups

- Two principle components:
  - A mechanism for hierarchically grouping processes
  - A set of controllers (kernel components) that manage, control, or monitor processes in cgroups
- Interface is via a pseudo-filesystem
- Cgroup manipulation takes form of filesystem operations, which might be done:
  - Via shell commands
  - Programmatically
  - Via management daemon (e.g., systemd)
  - Via your container framework’s tools (e.g., LXC, Docker)

What do cgroups allow us to do

- Limit resource usage of group
  - E.g., limit % of CPU available to group;
  - limit amount of memory that group can use
- Prioritize group for resource allocation
  - E.g., favor the group for network bandwidth
- Resource accounting
  - Measure resources used by processes
- Freeze a group
  - Freeze, restore, and checkpoint a group

Terminology

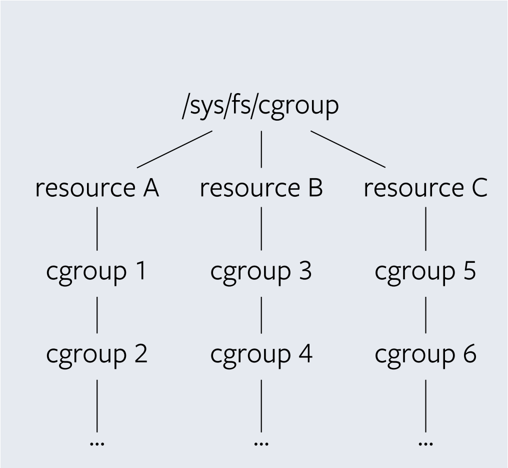
- Control group: a group of processes that are bound together for purpose of resource management
- (Resource) controller: kernel component that controls or monitors processes in a cgroup
  - E.g., memory controller limits memory usage; cpu controller limits CPU usage
- Cgroups are arranged in a hierarchy
  - Each cgroup can have zero or more child cgroups
  - Child cgroups inherit control settings from parent

cgroupsv1

- cgroupv1 has a hierarchy per-resource, for example:
  - % ls /sys/fs/cgroup
  - cpu/ cpuacct/ cpuset/ devices/ freezer/ memory/ net_cls/ pids/
- Each resource hierarchy contains cgroups for this resource:
  - % find /sys/fs/cgroup/pids -type d
  - /sys/fs/cgroup/pids/background.slice
    /sys/fs/cgroup/pids/background.slice/async.slice
    /sys/fs/cgroup/pids/workload.slice

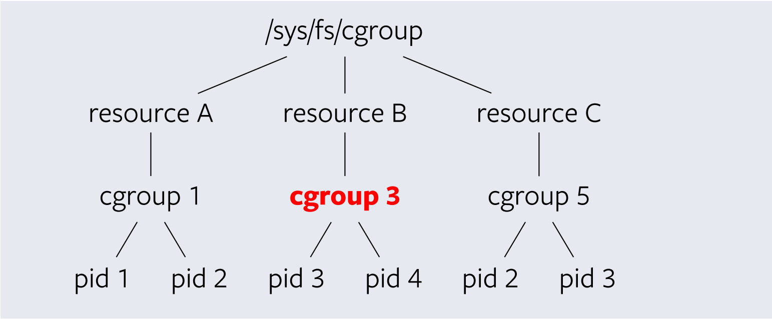
cgroupsv1

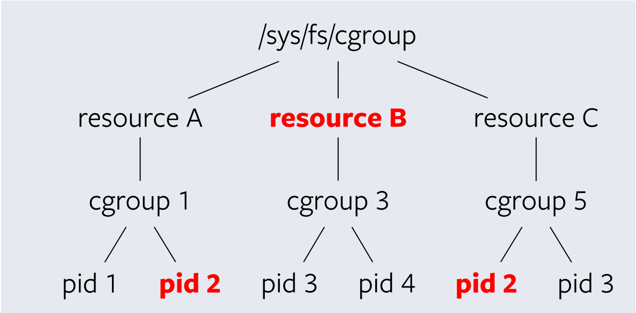
- Separate hierarchy/cgroups for each resource
- Even if they have the same name, cgroups for each resource are distinct
- cgroups can be nested inside each other



cgroupsv1

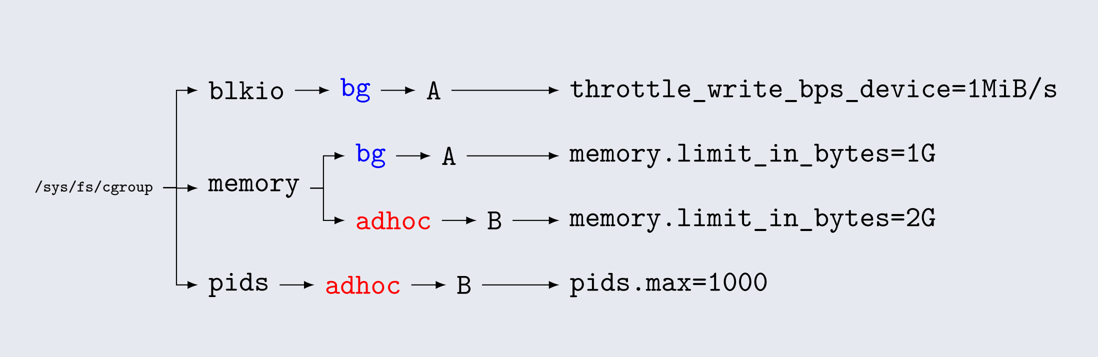
- Limits and accounting are performed per-cgroup
- If resource B is "memory", you can set memory.limit_in_bytes in cgroup 3





- One PID is in exactly one cgroup per resource
- PID 2 explicitly assigned in separate cgroups for resource A and C
- Not assigned for resource B, so in the root cgroup

cgroupsv1

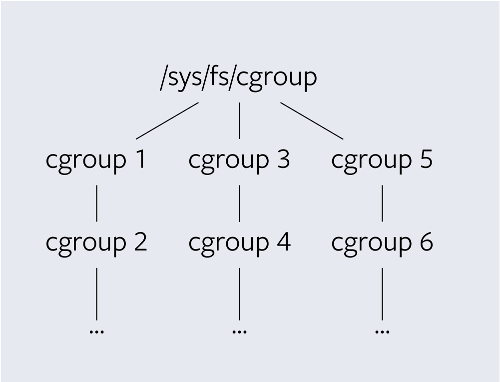

/sys/fs/cgroup
blkio
bg
A
throttle_write_bps_device=1MiB/s
memory
bg
A
memory.limit_in_bytes=1G
adhoc
B
memory.limit_in_bytes=2G
pids
adhoc
B
pids.max=1000

cgroups

- cgroupv2 has a unified hierarchy, for example:
  - % ls /sys/fs/cgroup
  - background.slice/ workload.slice/
- Each cgroup can support multiple resource domains:
  - % ls /sys/fs/cgroup/background.slice
  - async.slice/ foo.mount/ cgroup.subtree_control
  - memory.high memory.max pids.current pids.max

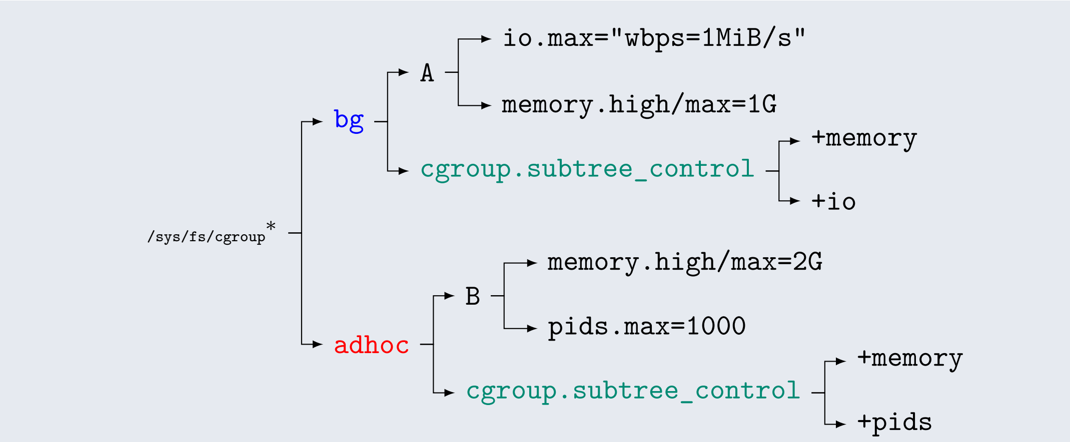
cgroupsv2

- cgroups are “global” now — not limited to one resource
- Resources are now opt-in for cgroups



cgroups v2 Vs v1

- Unified hierarchy — resources apply to cgroups now
- Granularity at TGID (PID), not TID level
- Focus on simplicity/clarity over ultimate flexibility



Determining the between v1 and v2

- You may be on a distro that uses cgroups v1 by default; if so, you need to reboot....
  - Because we can’t simultaneously use a controller in both v1 and v2
  - If this shows a value > 1, then you need to reboot:
    - $ grep -c cgroup /proc/mounts # Count cgroup mounts
- Use kernel boot parameter, cgroup_no_v1:
  - cgroup_no_v1=all ⇒ disable all v1 controllers

Filesystem interface

- Cgroup filesystem directory structure defines cgroups + cgroup hierarchy
  - I.e., use mkdir(2) / rmdir(2) (or equivalent shell commands) to create cgroups
- Each subdirectory contains automatically created files
  - Some files are used to manage the cgroup itself
  - Other files are controller-specific
- Files in cgroup are used to:
  - Define/display membership of cgroup
  - Control behavior of processes in cgroup
  - Expose information about processes in cgroup (e.g., resource usage stats)

Example

- cgd-demo.c