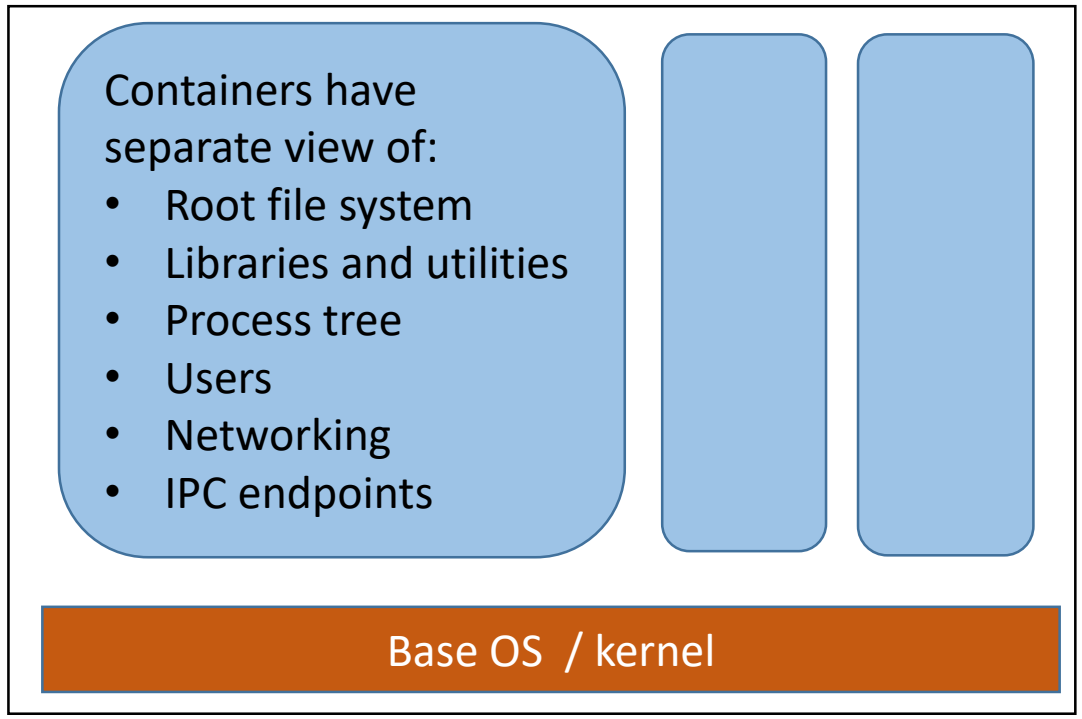
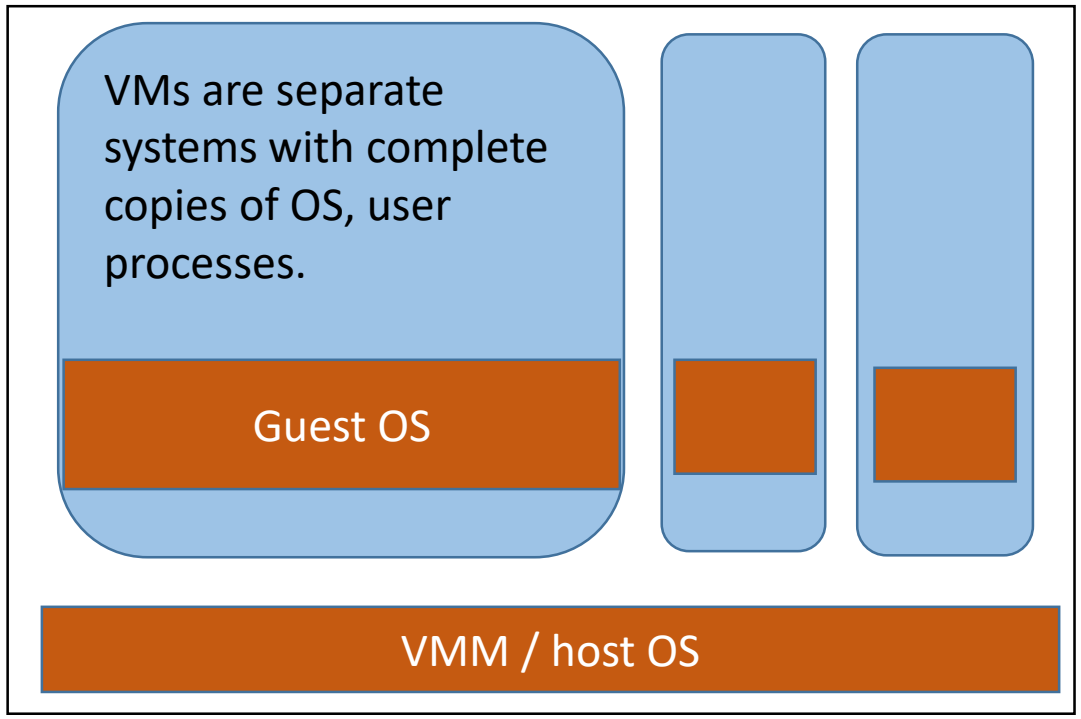
Containers: lightweight virtualization

- Containers share base OS, have different set of libraries, utilities, root filesystem, view of process tree, networking, and so on.
  - VMs have different copies of OS itself
  - Containers have lesser overhead than VMs, but also lesser isolation





Namespaces and Cgroups

- Two mechanisms in Linux kernel over which containers are built:
  - Namespaces: a way to provide isolated view of a certain global resource (e.g., root filesystem) to a set of processes. Processes within a namespace see only their slice of the global resource
  - Cgroups: a way to set resource limits on a group of processes
- Together, namespaces and cgroups allow us to isolate a set of processes into a bubble and set resource limits
- Container implementations like LXC, Docker leverage these mechanisms to build the container abstractions
  - LXC is general container while Docker optimized for single application
- Frameworks like Docker Swarm or Kubernetes help manage multiple containers across hosts, along with autoscaling, lifecycle management, and so on

Namespaces in operation (lwn.net) https://lwn.net/Articles/531114/

Documentation/cgroups/cgroups.txt https://lwn.net/Articles/524935/

Namespaces

- Group of processes that have an isolated/sliced view of a global resource
- Default namespace for all processes in Linux, system calls to create new namespaces and place processes in them
- Which resources can be sliced?
  1. Mount namespace: isolates the filesystem mount points seen by a group of processes. The mount() and umount() system calls only affect the processes in that namespace.
  2. PID namespace: isolates the PID numberspace seen by processes. E.g., first process in a new PID namespace gets a PID of 1.
  3. Network namespace: isolates network resources like IP addresses, routing tables, port numbers and so on. E.g., processes in different network namespaces can reuse the same port numbers.
  4. UTS namespace: isolates the hostname seen by processes.
  5. User namespace: isolates the UID/GID numberspace. E.g., a process can get UID=0 (i.e., act as root) in one namespace, while being unprivileged in another namespace. Mappings to be specified between UIDs in parent namespace and UIDs in new namespace.
  6. IPC namespace: isolates IPC endpoints like POSIX message queues.
- More powerful than chroot() which only isolates root filesystem

Namespaces API

- Three system calls related to namespaces:
  - clone() is used to create a new process and place it into a new namespace. More general version of fork().

```
childPID = clone(childFunc, childStack, flags, arg)
```

    Flags specify what should be shared with parent, and what should be created new for child (including virtual memory, file descriptors, namespaces etc.)
  - setns() lets a process join an existing namespace. Arguments specify which namespace, and which type.
  - unshare() creates a new namespace and places calling process into it. Flags indicate which namespace to create. Forking a process and calling unshare() is equivalent to clone().
- Once a process is in a namespace, it can open a shell and do other useful things in that namespace
- Forked children of a process belong to parent namespace by default

Namespace handles: how to refer to a namespace

- /proc/PID/ns of a process has information on which namespace a process belongs to. Symbolic links pointing to the inode of that namespace ("handle")

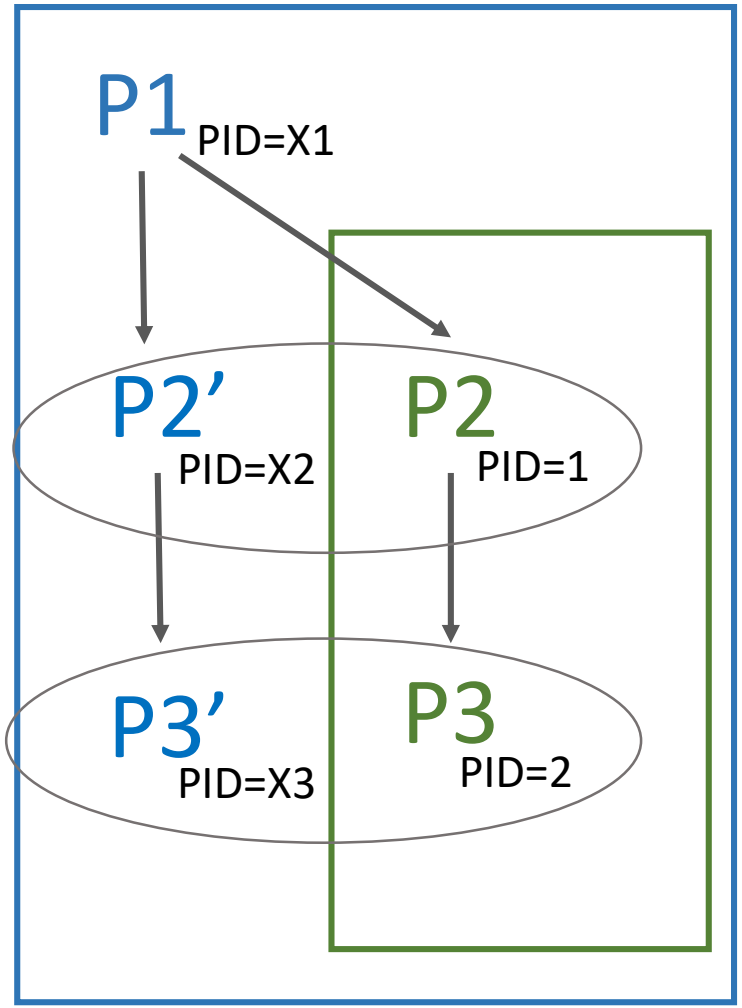
```
$ ls -l /proc/$$/ns          # $$ is replaced by shell's PID
total 0
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 ipc -> ipc:[4026531839]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 mnt -> mnt:[4026531840]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 net -> net:[4026531956]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 pid -> pid:[4026531836]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 user -> user:[4026531837]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 uts -> uts:[4026531838]
```

- Namespace handle can be used in system calls (e.g., argument to setns)
- Processes in same namespace will have same handle, new handle created when new namespace created

Namespaces in operation (lwn.net) https://lwn.net/Articles/531114/

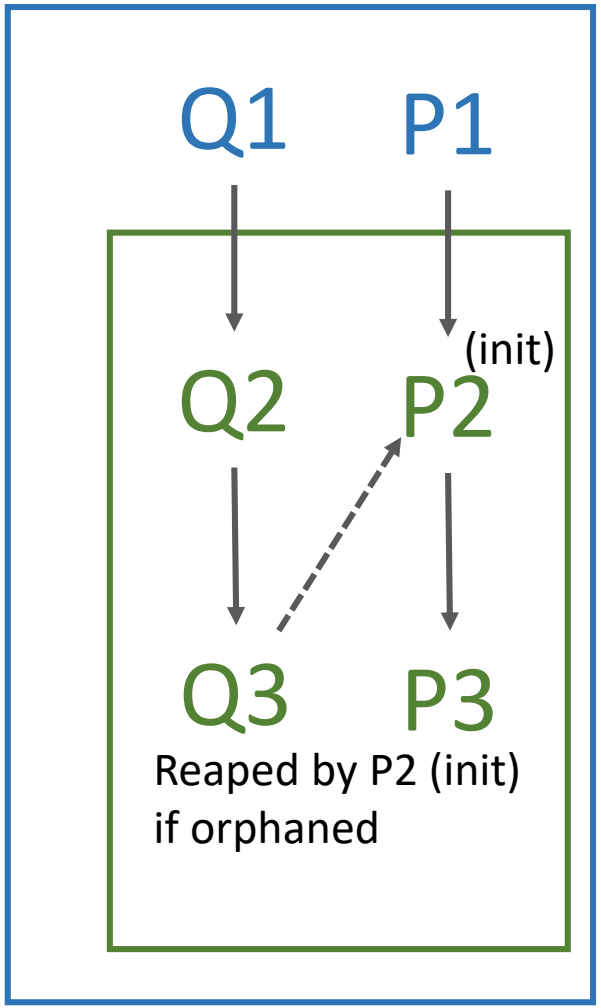
PID namespaces (1)

- The first process to be created in a new PID namespace will have PID=1 and will act as init process in that namespace
  - Will reap orphans in this namespace
- Processes in PID namespace get separate PID numberspace (child of init gets PID 2 onwards)
- A process can see all other processes in its own or nested namespaces, but not in its parent namespace
  - P2, P3 not aware of P1 (parent PID of P2 = 0)
  - P1 can see P2 and P3 in its namespace (with different PIDs)
  - P2=P2’ (just different PIDs in different namespaces)



PID namespaces (2)

- First process in a namespace acts as init and has special privileges
  - Other processes in namespace cannot kill it
  - If init dies, namespace terminated
  - However, parent process can kill it in parent namespace
- Who reaps whom?
  - Init process reaped by parent in parent namespace
  - Other child processes reaped by parent in same namespace
  - Any orphan process in namespace reaped by init of that namespace



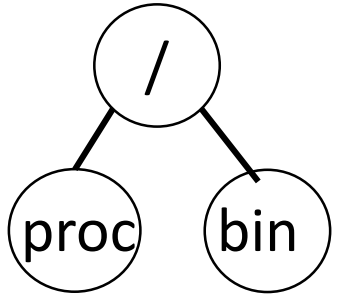
PID namespaces (3)

- Namespace related system calls have slightly different behavior with PID namespace alone
- clone() creates a new namespace for child as expected
- However, setns() and unshare() do not change PID namespace of calling process. Instead, the child processes will begin in a new PID namespace
- Why this difference? If namespace changes, PID returned by getpid() will also change. But many programs make assumption that getpid() returns same value throughout life of process.
  - getpid() returns the PID in the namespace the process resides in

Mount namespaces

- Root filesystem seen by a process is constructed from a set of mount points (mount() and umount() syscalls)
- New mount namespace can have new set of mount points
  - New view of root filesystem
- Mount point can be shared or private
  - Shared mount points propagated to all namespaces, private is not
  - If parent makes all its mount points private and clones child in new mount namespace, child starts with empty root filesystem
- Container frameworks use mount namespaces to create a custom root filesystem for each container using a base rootfs image

Mount namespaces and ps



- How does “ps” work?
  - Linux has a special procfs, in which kernel populates info on processes
  - Reading /proc/PID/.. does not read file from disk, but fetches info from OS
  - procfs mounted on root as a special type of filesystem
- P1 clones P2 to be in new PID namespace but uses old mount namespace. We open shell in new PID namespace and run ps. We will still see all processes of parent namespace. Why?
  - ps command is still using procfs of parent’s mount namespace
- How to make ps work correctly within a PID namespace?
  - Place P2 in new mount namespace, mount a new procfs at root
  - New procfs at new mount point is different from parent’s procfs
  - ps will not show only processes in this PID+mount namespace

Network namespaces (1)

- Network namespace can be created by cloning a process into a new namespace, or simply via commandline

```
# ip netns add netns1
```

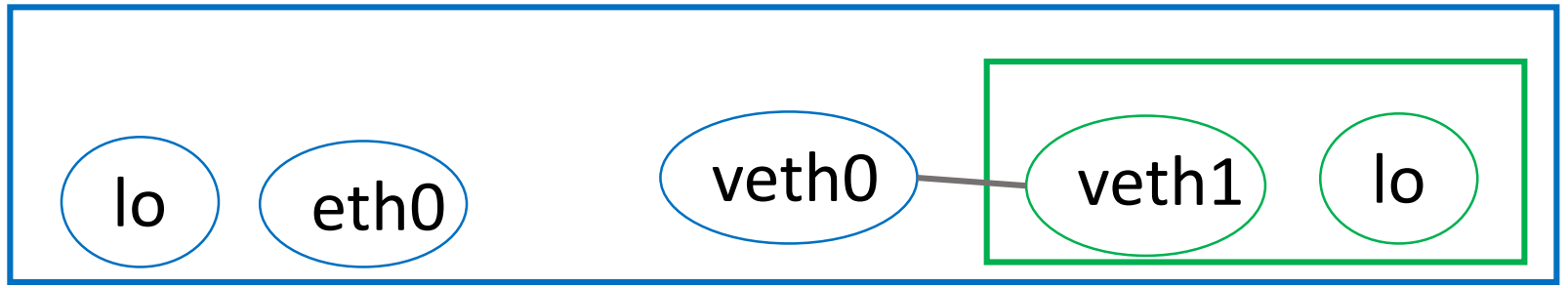
- List of network namespaces can be viewed at /var/run/netns, can use setns() to join existing namespace
- Command “ip netns exec” can be used to execute commands within network namespace, for example, to view all IP links:

```
# ip netns exec netns1 ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Namespaces in operation, part 7: Network namespaces https://lwn.net/Articles/580893/

Network namespaces (2)

- Any new network namespace only has loopback interface. How to communicate with rest of network?
- Create a virtual Ethernet link (veth pair) to connect parent namespace to new child namespace
  - Assign endpoints to two different namespaces
  - Assign IP addresses to both endpoints
  - Can communicate over this link to parent namespace
  - Can configure bridging/NAT to connect to wider internet



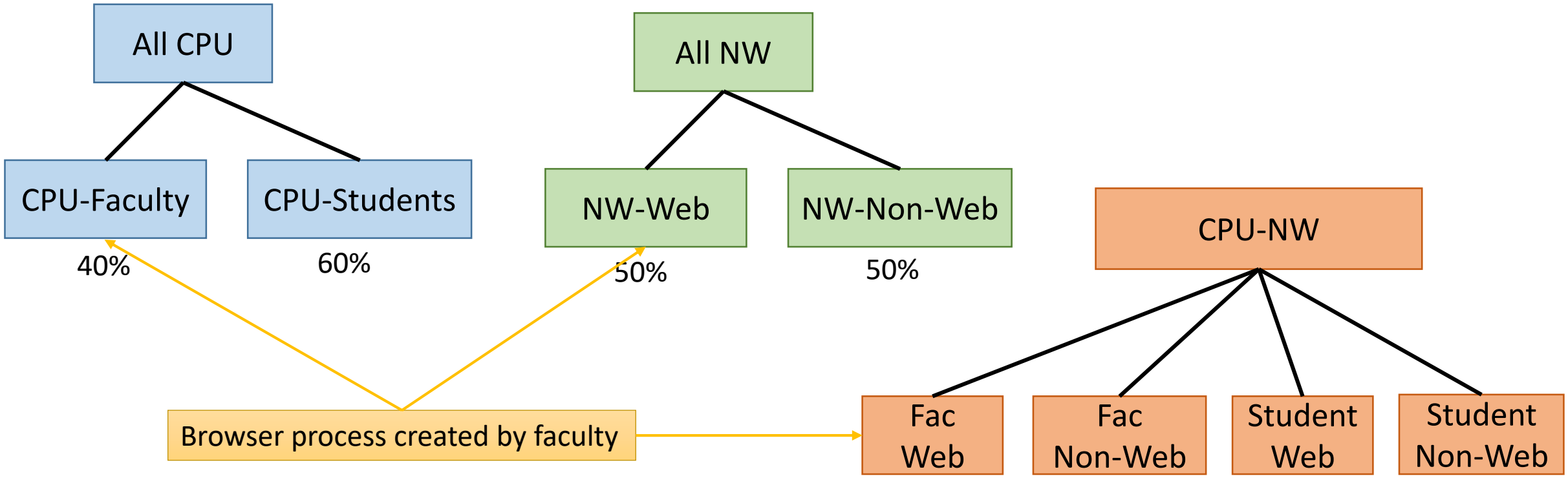
Namespaces in operation, part 7: Network namespaces https://lwn.net/Articles/580893/

The next building block: Cgroups

- Namespaces let us isolate processes into a slice with respect to many resources: mount points, PID/UID numberspace, network endpoints, etc.
- The next topic Cgroups will let us assign resource limits on a set of processes
  - Divide processes into groups and subgroups hierarchically
  - Assign resource limits for processes in each group/subgroup
- Which resources can be limited? CPU, memory, I/O, CPU sets (which process can execute on which CPU core), and so on
  - Specify what fraction of resource can be used by each group of processes

Cgroups hierarchies

- Can create separate hierarchies for each resource, or a combined hierarchy for multiple resources together



Creating cgroups

- No new system calls, managed via the filesystem
- A special cgroup filesystem mounted at /sys/fs/cgroup
- Create directories and sub-directories for different resources and different user classes
- Write “founding father” task PID to tasks file
  - All children of this task will be in same cgroup too

```
# echo browser_pid > /sys/fs/cgroup/<restype>/<userclass>/tasks
```

- Tasks can be assigned to leaf nodes in hierarchy
  - Tasks will belong to default cgroup of parent if not explicitly placed into any hierarchy

Documentation/cgroups/cgroups.txt https://lwn.net/Articles/524935/

How to create a container?

- Suppose you wish to run an application/shell in a container. How?
- Create separate namespaces for isolation
- Create and configure cgroups for resource limits
- Create root filesystem that is compatible with CPU's ISA and OS binary
  - All utilities, binaries, configuration files needed to run the application
- A process enters the namespaces, mounts rootfs, registers in cgroups, execs desired application or shell
  - Your application or shell is running in a "container"!
- Many tutorials online on how to create your own container

Container frameworks

- Existing container frameworks like LXC and Docker do the namespace/cgroup configuration automatically “under the hood”
- LXC container is a lightweight VM
  - Provides standard OS shell interface
  - Uses namespaces and cgroups under the hood
- Docker containers are optimized to run a single application
  - Docker config file specifies base root filesystem, along with utilities needed to run a specific application
  - Runs application in a container environment
  - Easy way to package an application and all its dependencies and run anywhere

Container orchestration frameworks

- Docker Swarm, Kubernetes – frameworks to manage multiple containers on multiple hosts
- Kubernetes – popular container orchestration framework
  - Runs over multiple physical machines ("nodes") each with multiple "pods“
  - A pod contains one or more containers within the same network namespace, with the same IP address
  - Pod is a tier of a multi-tier application (e.g., "frontend", "backend", "database", "webserver")
  - Kubernetes manages multiple nodes and their pods, e.g., instantiating pods on free nodes, auto-scaling pods when load increases, restarting pods when they crash, etc.

Summary

- Containers provide lightweight isolation with lower overhead
- Containers share same kernel binary, have different root filesystems (utilities, configurations) over the kernel
- Implemented using two Linux primitives
  - Namespaces for isolation
  - Cgroups for resource limits
- Frameworks like Docker, LXC, Kubernetes provide more functionality by building upon these primitives

Namespaces in operation (lwn.net) https://lwn.net/Articles/531114/

Documentation/cgroups/cgroups.txt https://lwn.net/Articles/524935/