Namespaces and Control Groups

Namespaces

- A namespace wraps a global system resource in an abstraction
  - Processes within the namespace believe that they have their own isolated instance of the resource
  - This can provide a group of processes with the illusion that they are the only processes on the system, and forms the basis of containers
- The namespace API consists of three system calls— clone(), unshare(), and setns()—and a number of /proc files

Defined Namespaces

- There are currently 6 namespaces:
  - mnt (mount points) clone flag: CLONE_NEWNS
  - pid (processes) clone flag: CLONE_NEWPID
  - net (network stack) clone flag: CLONE_NEWNET
  - ipc (System V IPC) clone flag: CLONE_NEWIPC
  - uts (hostname) clone flag: CLONE_NEWUTS
  - user (UIDs) clone flag: CLONE_NEWUSER
- There is an **initial, default namespace** for each of the 6 namespaces
- Creating a new namespace requires **CAP_SYS_ADMIN** for all but user_ns

The clone() system call

- *clone()* - creates a new process/thread
  - The newly created task can be attached to one or more new namespaces during its creation
  - Namespace selection is done with bit flags passed as an argument to the clone call

```
int clone(int  (*child_func)(void *),
              void  *child_stack,
              int  flags,           // namespace bits, etc.
              void  *arg);          //  function arg
```

The setns() system call

- The /proc file system retains namespace information for each task on the system
- This information is managed in a set of symbolic links found at */proc/<PID|TID>/ns*
- Opening one of these links will allow the associated namespace to persist even if there are no longer any *<PID|TID>s* in it
- setns() allows an unassociated *<PID|TID>* to be moved into such a namespace

The setns() system call (cont'd)

- If a namespace is constructed via a clone() call, then setns() can move any other calling *<PID|TID>* into that namespace

```
int setns(int fd,           // open link
          int nstype);  //  check ns bit or 0
```

The unshare() system call

- The unshare() system call supports namespace manipulation for an existing task
  - Unlike **clone()**, it's not used to create a task
  - It operates on the calling task
  - Creates one or more new namespaces
  - Extracts the calling task from its current namespace(s)
  - Places the calling task into the new namespace(s)

int unshare(int flags); // ns bits

The unshare() system call (cont'd)

- When a task wants to transition itself from one or more of its current namespaces, to corresponding new private namespaces.

```
#define _GNU_SOURCE
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
.
.
// move to private PID and mount namespaces
int my_flags = (CLONE_NEWPID | CLONE_NEWNS);
if( unshare(my_flags) == -1){
      perror("unshare() failed\n");
      exit(3);
}
```

The unshare command

- Unshares the indicated namespace(s) from the parent process and then *executes* the specified program from one or more new namespaces
- Provides functionality similar to the clone() system call
- Namespaces are designated with options

unshare [options] program [arguments]

unshare command

```
# ps
  PID TTY          TIME CMD
21078 pts/0    00:00:00 bash
26417 pts/0    00:00:00 ps
# ls -l /proc/21078/ns
total 0
lrwxrwxrwx. 1 root root 0 Sep 28 11:28 ipc -> ipc:[4026531839]
lrwxrwxrwx. 1 root root 0 Sep 28 11:28 mnt -> mnt:[4026531840]
lrwxrwxrwx. 1 root root 0 Sep 28 11:28 net -> net:[4026531957]
lrwxrwxrwx. 1 root root 0 Sep 28 11:28 pid -> pid:[4026531836]
lrwxrwxrwx. 1 root root 0 Sep 28 11:28 user -> user:[4026531837]
lrwxrwxrwx. 1 root root 0 Sep 28 11:28 uts -> uts:[4026531838]
# echo $$
21078
```

unshare command

```
# unshare --pid --fork bash  // create bash in a private PID ns
# ps
  PID TTY          TIME CMD
21078 pts/0    00:00:00 bash
26419 pts/0    00:00:00 unshare
26420 pts/0    00:00:00 bash
26444 pts/0    00:00:00 ps
# echo $$
1
# ls -l /proc/26420/ns
total 0
lrwxrwxrwx. 1 root root 0 Oct 13 19:41 ipc -> ipc:[4026531839]
lrwxrwxrwx. 1 root root 0 Oct 13 19:41 mnt -> mnt:[4026531840]
lrwxrwxrwx. 1 root root 0 Oct 13 19:41 net -> net:[4026531957]
lrwxrwxrwx. 1 root root 0 Oct 13 19:41 pid -> pid:[4026532563]
lrwxrwxrwx. 1 root root 0 Oct 13 19:41 user -> user:[4026531837]
lrwxrwxrwx. 1 root root 0 Oct 13 19:41 uts -> uts:[4026531838]
```

namespace inherited

```
# bash
# echo $$
26
# jobs
# ps
  PID TTY          TIME CMD
21078 pts/0    00:00:00 bash
26419 pts/0    00:00:00 unshare
26420 pts/0    00:00:00 bash
26474 pts/0    00:00:00 bash
26518 pts/0    00:00:00 ps
```

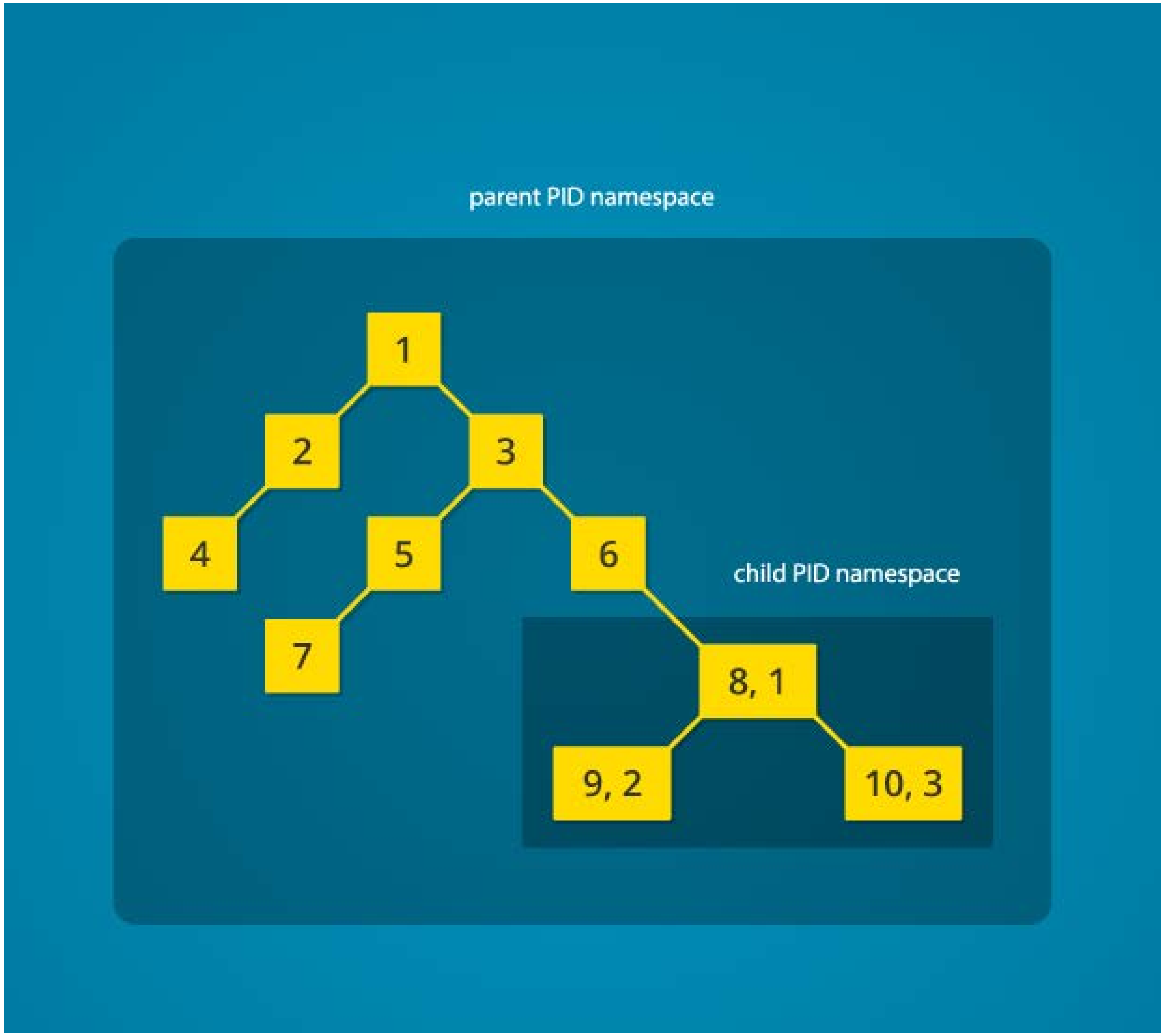
PID visibility

```
# ./mypid &
[1] 54
#
MY PID is 54
ps
  PID TTY          TIME CMD
21078 pts/0    00:00:00 bash
26419 pts/0    00:00:00 unshare
26420 pts/0    00:00:00 bash
26474 pts/0    00:00:00 bash
26586 pts/0    00:00:00 mypid
26587 pts/0    00:00:00 ps
# kill 26586
bash: kill: (26586) - No such process
# kill 54
[1]+  Terminated              ./mypid
#
```

PID namespace

- The global resource isolated by PID namespaces is the process ID number space
- This means that processes in different PID namespaces can have the same process ID
- the process IDs *within* a PID namespace are unique, and are assigned sequentially starting with PID 1
- A new PID namespace is created by calling ***clone()*** or ***unshare()*** with the CLONE_NEWPID flag

PID namespace (cont'd)



PID namespace (cont'd)

- Operations involving PIDs are limited to PIDs within the given namespace
  - A kill command can only operate within a namespace on the PIDs that are visible
    - No view from an inner namespace to an outer namespace
  - Outer namespaces have visibility to inner namespaces
    - Can use outer namespace PIDs to interact with inner-namespace processes/threads

PID namespace (cont'd)

```
# ./mypid
MY PID is 90
^Z
[1]+  Stopped                 ./mypid
# ps
  PID TTY          TIME CMD
21078 pts/0    00:00:00 bash
26419 pts/0    00:00:00 unshare
26420 pts/0    00:00:00 bash
26474 pts/0    00:00:00 bash
26831 pts/0    00:00:00 mypid
26832 pts/0    00:00:00 ps
# mount -t proc proc /proc
# ps
  PID TTY          TIME CMD
    1 pts/0    00:00:00 bash
   26 pts/0    00:00:00 bash
   90 pts/0    00:00:00 mypid
   93 pts/0    00:00:00 ps
```

User namespaces

- User namespaces allow per-namespace mappings of user and group IDs
  - A process's user and group IDs inside a user namespace can be different from its IDs outside of the namespace
  - A process can have a nonzero user ID outside a namespace while at the same time having a user ID of zero inside the namespace
  - Unprivileged processes outside the namespace can create root privileged process within the namespace

User namespaces (cont’d)

- User namespaces are created by specifying the CLONE_NEWUSER flag when calling ***clone()*** or ***unshare()***
  - A mapping can then be made from within the new user namespace
  - The initial UID and GID mappings are set for the /proc/sys/kernel/overflowuid values of 65534
  - A one-time write can then be made to /proc/<PID>/uid_map (or gid_map)

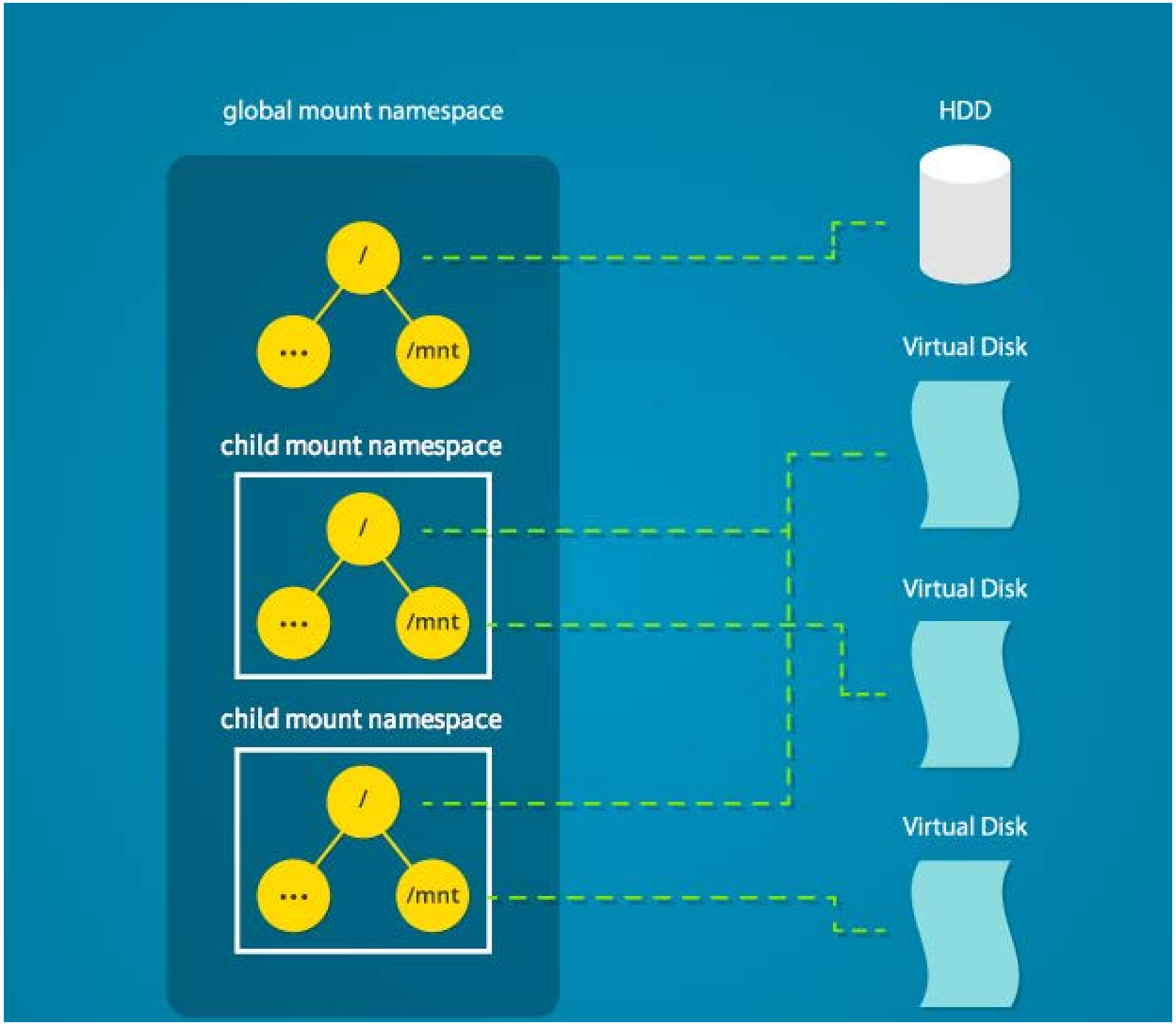
echo '0 1000 1' > /proc/$$/uid_map

  - Maps the inside UID = 0 from an outside UID = 1000

Mount namespace

- Processes under different namespaces can change the mountpoints without affecting each other
- Creating separate mount namespace has an effect similar to doing a ***chroot()***, but much more comprehensive
  - All mountpoints can be controlled, not just the root
  - Can hide areas of the underlying system

Mount namespace (cont'd)



Mount namespace (cont'd)

- Initially, the child process sees the exact same mountpoints as its parent process would
- Being under a new mount namespace, the child process can mount or unmount whatever endpoints it wants to
  - the change will affect neither its parent's namespace, nor any other mount namespace in the entire system

Mount namespace (cont'd)

- A common approach is to start a special **"init"** process with the CLONE_NEWNS flag
  - This process can then set up the mount namespace
  - Typically this would involve changing the "/", "/proc", "/dev" or other mountpoints as desired
  - These changes would not affect other processes in the system
  - Processes in this new mount namespace have only the view of the file hierarchy provided by the local mounts

Network namespaces

- Network namespaces partition the use of the network
  - Devices
  - Addresses
  - Ports
  - Routes
  - Firewall rules
- Essentially virtualizing the network within a single running kernel instance

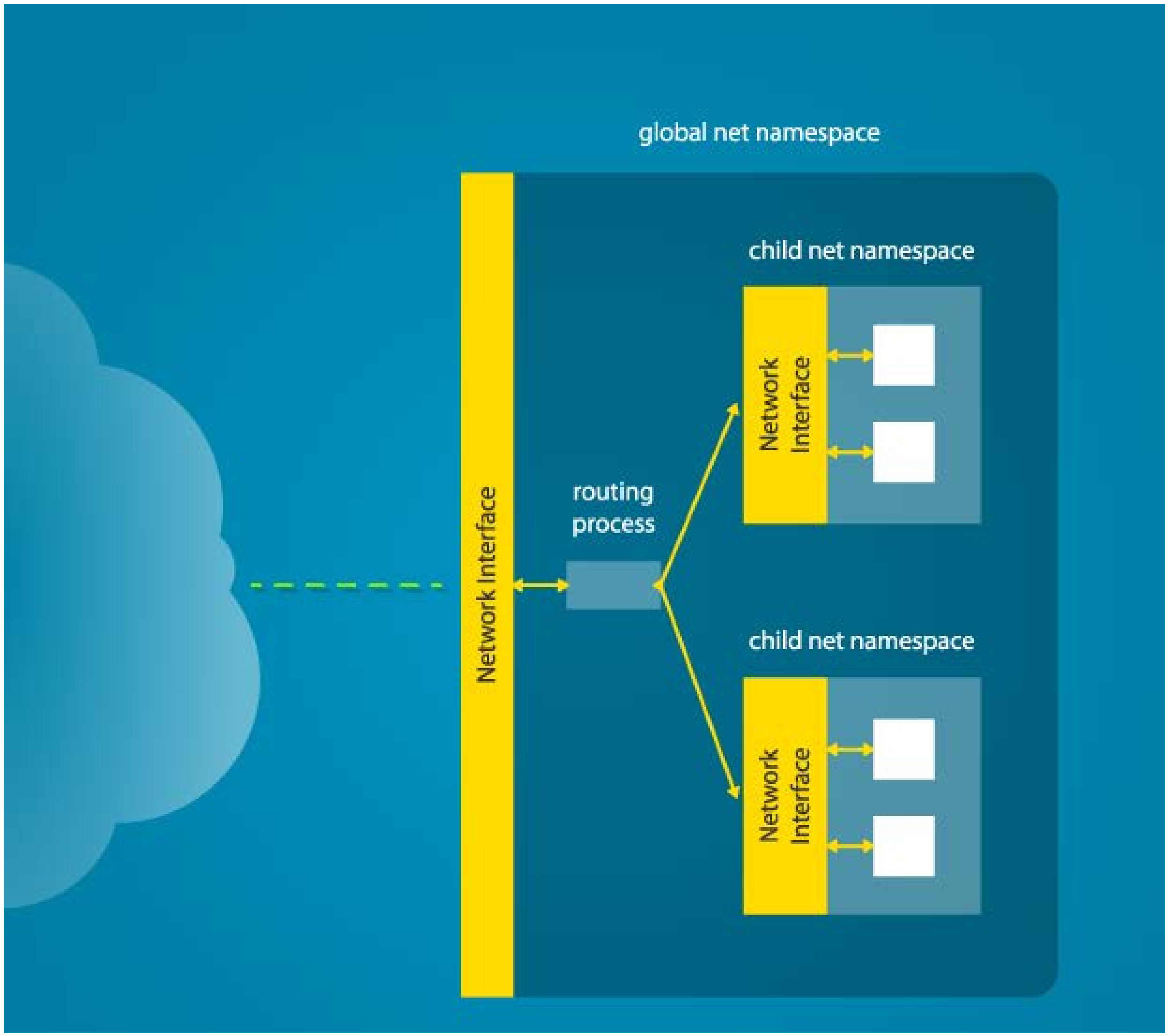
Network namespaces (cont'd)

- Network namespaces are created by passing a flag to the clone() system call: CLONE_NEWNET
  - it is convenient to use the **ip** networking configuration tool to set up and work with network namespaces

    ip netns add netns1

  - This command creates a new network namespace called netns1

Network namespaces (cont'd)



Network namespaces (cont’d)

- When the ip tool creates a network namespace, it will create a bind mount for it under /var/run/netns
  - The namespace will now persist even if no processes are currently alive within it
  - **“ip netns exec”** can now run management tasks

```
# 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
```

Network namespaces (cont'd)

- For communication between the inside and outside namespaces, virtual ethernet devices need to be created and configured

```
# ip link add veth0 type veth peer name veth1
# ip link set veth1 netns netns1
# ip netns exec netns1 ifconfig veth1 10.1.1.1/24 up
# ifconfig veth0 10.1.1.2/24 up
```

Network namespaces (cont'd)

- Communication in both directions is now possible

ping 10.1.1.1 PING 10.1.1.1 (10.1.1.1) 56(84) bytes of data. 64 bytes from 10.1.1.1: icmp_seq=1 ttl=64 time=0.087 ms ...

ip netns exec netns1 ping 10.1.1.2 PING 10.1.1.2 (10.1.1.2) 56(84) bytes of data. 64 bytes from 10.1.1.2: icmp_seq=1 ttl=64 time=0.054 ms ...

- Namespaces do not share routing tables or firewall rules, so these must be configured

UTS namespace

- The UTS namespace manages:
  - sysname
  - nodename (hostname)
  - release
  - version
  - machine
  - domainname

UTS namespace (cont'd)

- Since hostname and domainname are used in several network configuration steps, the UTS namespace allows isolation

```
# uname -n
  myoldhostname  // outer namespace
# unshare -u /bin/bash
# hostname mynewhostname
# uname -n
  mynewhostname  // inner namespace
```

IPC namespace

- IPC namespace isolates the System V inter-process communication within a namespace
  - Semaphores
  - Shared memory
  - Message queues
- These resources use a numeric ket-to-ID naming convention which can be localized to a namespace
- POSIX Message Queues are also included

cgroups (Control Groups)

- A *cgroup* associates a set of tasks with a set of parameters for one or more subsystems
  - A *subsystem* is a module that makes use of the task grouping to treat groups of tasks in particular ways
  - A subsystem is typically a "resource controller" that schedules a resource or applies per-cgroup limits
- **Namespaces** provide a per process resource **isolation** solution.
- **Cgroups**, on the other hand, provide a resource **management** solution (handling groups).

cgroups (cont'd)

- The implementation of cgroups is based around the VFS (Virtual File System)
  - A new file system of type "cgroup" (VFS)
  - Addition of *procfs* entries:
    - For each process: */proc/pid/cgroup*
    - System-wide: */proc/cgroups*
- All cgroups actions are performed via filesystem actions
  - (create/remove directory, reading/writing to files in it, mounting/mount options)
- Usually, cgroups are mounted on ***/sys/fs/cgroup***

cgroups (cont'd)

```
# ls -l /sys/fs/cgroup
total 0
dr-xr-xr-x. 4  0 Sep  2 11:22 blkio
lrwxrwxrwx. 1 11 Aug 26 11:59 cpu -> cpu,cpuacct
lrwxrwxrwx. 1 11 Aug 26 11:59 cpuacct -> cpu,cpuacct
dr-xr-xr-x. 4  0 Sep  2 11:22 cpu,cpuacct
dr-xr-xr-x. 2  0 Sep  2 11:22 cpuset
dr-xr-xr-x. 4  0 Sep  2 11:22 devices
dr-xr-xr-x. 2  0 Sep  2 11:22 freezer
dr-xr-xr-x. 2  0 Sep  2 11:22 hugetlb
dr-xr-xr-x. 4  0 Sep  2 11:22 memory
lrwxrwxrwx. 1 16 Aug 26 11:59 net_cls -> net_cls,net_prio
dr-xr-xr-x. 2  0 Sep  2 11:22 net_cls,net_prio
lrwxrwxrwx. 1 16 Aug 26 11:59 net_prio -> net_cls,net_prio
dr-xr-xr-x. 2  0 Sep  2 11:22 perf_event
dr-xr-xr-x. 4  0 Aug 31 10:46 systemd
```

- There are 10 cgroup subsystems (controllers)
  - **cpuset_subsys** - CPU Management Controller.
  - **freezer_subsys** – Checkpointing and Scheduling Controller.
  - **mem_cgroup_subsys** - Memory Resource Controller.
  - **blkio_subsys** - Block IO Controller.
  - **net_cls_subsys - ** Network Classifier Controller.
  - **net_prio_subsys** - Network Priority Controller.
  - **devices_subsys** - Device Whitelist Controller.
  - **perf_subsys (perf_event)** - Event Management Controller.
  - **hugetlb_subsys -** HugeTLB Controller.
  - **cpuacct_subsys** - CPU Accounting Controller.

cgroups (cont’d)

- Each of these subsystems allows for specific configuration of various system resources
- The file system is used in conjunction with the mount command to create hierarchies
- For a given subsystem, a top level directory will contain the default settings
  - Using the mkdir command in this top level directory will produce a populated subdirectory that can be tuned for specific needs

cgroups (cont'd)

- All top level directories have 5 common files
  - tasks
  - cgroup.procs
  - cgroup.event_control
  - notify_on_release
  - release_agent
- Subdirectories contain all but the release_agent

cgroups (cont'd)

- Documentation for the various configuration parameters can be found at:

https://www.kernel.org/doc/Documentation/cgroups

- When a parameter is identified for change, then use the echo command to write to the pathname for that parameter
- We can then use the echo command to place a PID of interest into the tasks file in our modified subsystem

cgroups example

```
# pwd
/sys/fs/cgroup/blkio
# ls
blkio.io_merged                      blkio.throttle.io_serviced
blkio.io_merged_recursive            blkio.throttle.read_bps_device
blkio.io_queued                      blkio.throttle.read_iops_device
blkio.io_queued_recursive            blkio.throttle.write_bps_device
blkio.io_service_bytes
blkio.throttle.write_iops_device
blkio.io_service_bytes_recursive  blkio.time
blkio.io_serviced                    blkio.time_recursive
blkio.io_serviced_recursive          blkio.weight
blkio.io_service_time                blkio.weight_device
blkio.io_service_time_recursive   cgroup.clone_children
blkio.io_wait_time                   cgroup.procs
blkio.io_wait_time_recursive         cgroup.sane_behavior
blkio.leaf_weight                    notify_on_release
blkio.leaf_weight_device             release_agent
blkio.reset_stats                    system.slice
blkio.sectors                        tasks
blkio.sectors_recursive              user.slice
blkio.throttle.io_service_bytes   cgroup.event_control
```

cgroups example (cont'd)

```
[root@localhost blkio]# pwd
/sys/fs/cgroup/blkio
[root@localhost blkio]# mkdir test1
[root@localhost blkio]# ls
blkio.io_merged                     blkio.throttle.io_serviced
blkio.io_merged_recursive           blkio.throttle.read_bps_device
blkio.io_queued                     blkio.throttle.read_iops_device
blkio.io_queued_recursive           blkio.throttle.write_bps_device
blkio.io_service_bytes              blkio.throttle.write_iops_device
blkio.io_service_bytes_recursive    blkio.time
blkio.io_serviced                   blkio.time_recursive
blkio.io_serviced_recursive         blkio.weight
blkio.io_service_time               blkio.weight_device
blkio.io_service_time_recursive     cgroup.clone_children
blkio.io_wait_time                  cgroup.procs
blkio.io_wait_time_recursive        cgroup.sane_behavior
blkio.leaf_weight                   notify_on_release
blkio.leaf_weight_device            release_agent
blkio.reset_stats                   system.slice
blkio.sectors                       tasks
blkio.sectors_recursive             test1
blkio.throttle.io_service_bytes     user.slice
```

cgroups example (cont’d)

```
root@localhost blkio]# cd test1
[root@localhost test1]# ls
blkio.io_merged                      blkio.sectors_recursive
blkio.io_merged_recursive            blkio.throttle.io_service_bytes
blkio.io_queued                      blkio.throttle.io_serviced
blkio.io_queued_recursive            blkio.throttle.read_bps_device
blkio.io_service_bytes               blkio.throttle.read_iops_device
blkio.io_service_bytes_recursive     blkio.throttle.write_bps_device
blkio.io_serviced                    blkio.throttle.write_iops_device
blkio.io_serviced_recursive          blkio.time
blkio.io_service_time                blkio.time_recursive
blkio.io_service_time_recursive      blkio.weight
blkio.io_wait_time                   blkio.weight_device
blkio.io_wait_time_recursive         cgroup.clone_children
blkio.leaf_weight                    cgroup.procs
blkio.leaf_weight_device             notify_on_release
blkio.reset_stats                    tasks
blkio.sectors
```

cgroups example (cont'd)

```
 echo "<major>:<minor>  <rate_bytes_per_second>"
           > /cgrp/blkio.throttle.read_bps_device

# ls -l /dev/sd*
brw-rw----. 1 root disk 8, 0 Aug 26 11:59 /dev/sda
brw-rw----. 1 root disk 8, 1 Aug 26 11:59 /dev/sda1
brw-rw----. 1 root disk 8, 2 Aug 26 11:59 /dev/sda2
# echo "8:0 5000000" > blkio.throttle.read_bps_device
# cat blkio.throttle.read_bps_device
8:0 5000000

# cgexec –g blkio:test1 <command> <arg>
```