Proceedings of NetDev 1.1: The Technical Conference on Linux Networking (February 10th-12th 2016. Seville, Spain)

Namespaces and cgroups, the basis of Linux containers

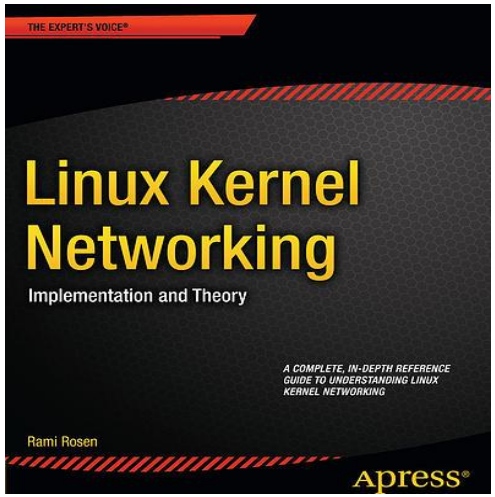
Rami Rosen

About Myself:

I am a working for Intel for various projects, primarily Kernel networking.

My website: http://ramirose.wix.com/ramirosen

I am the author of a book titled **"Linux Kernel Networking"** by Apress, 648 pages, 2014:



Agenda:

Overview of the cgroup subsystem and the namespace subsystem

cgroups

The PIDs cgroup controller

cgroup v2

namespaces

Backup

- The **namespace subsystem** and the **cgroup subsystem** are the basis of lightweight process virtualization.
- They form the basis of Linux containers.
- Can be used also for setting a testing environment or as a resource management/resource isolation setup and for accounting.
- We will talk mainly about the kernel implementation with some userspace usage examples.

lightweight process virtualization: A process which gives the user an illusion that he runs a Linux operating system. You can run many such processes on a machine, and all such processes in fact share a **single Linux kernel** which runs on the machine.

This is opposed to hypervisor solutions, like Xen or KVM, where you run **another instance of the kernel.**

The idea is not really a new paradigm - we have Solaris Zones and BSD jails already several years ago.

It seems that Hypervisor-based VMs like KVM are here to stay (at least for the next several years). There is an ecosystem of cloud infrastructure around solutions like KVMs.

Advantages of Hypervisor-based VMs (like KVM) :

- You can create VMs of other operating systems (windows, BSDs).
- Security
  - Though there were cases of security vulnerabilities which were found and required installing patches to handle them (like VENOM).

Containers versus Hypervisor-based VMs

Containers – advantages:

- **Lightweight:** occupies less resources (like memory) significantly then a hypervisor.
- **Density:** you can install many more containers on a given host then KVM based VMs.
- **Elasticity:** startup time and shutdown time is much shorter, almost instantaneous. Creation of a container has the overhead of creating a Linux process, which can be of the order of milliseconds, while creating a VM based on XEN/KVM can take seconds.

The lightness of the containers in fact provides both their density and their elasticity.

The cgroup subsystem: background

The cgroup (control groups) subsystem is a **Resource Management and Resource Accounting/Tracking** solution, providing a generic process-grouping framework.

- It handles resources such as memory, cpu, network, and more; mostly needed in both ends of the spectrum (servers and embedded).
- Development was started by engineers at Google in **2006** under the name "**process containers**".
- Merged into kernel 2.6.24 (2008).
- cgroup core has 3 maintainers, and each cgroup controller has its own maintainers.

cpu, *memory* and *io* are the most important resources that cgroup deals with. For networking, it's mostly about allowing network layer to match packets to cgroups. The actual control part still happens on the network side through *iptables* and *tc*.

cgroup subsystem implementation

No new system call was needed in order to support cgroups.

A new file system (VFS), "cgroup" (also referred sometimes as *cgroupfs*).

The implementation of the cgroup subsystem required a few, simple hooks into the rest of the kernel, **none in performance-critical paths:**

– In boot phase (***init/main.c*)** to perform various initializations.

– In process creation and termination methods, ***fork()*** and ***exit().***

– Minimal additions to the process descriptor (*struct task_struct*)

– Add *procfs* entries:

- For each process: */proc/pid/cgroup.*
- System-wide: */proc/cgroups*

cgroup subsystem implementation - contd

- The cgroups subsystem (v1 and v2) is composed of 2 ingredients:
  - **cgroup core.**
    - Mostly in *kernel/cgroup.c* (~6k lines of code).
      - The same code serves both cgroup V1 and cgroup V2.
  - **cgroup controllers.**
    - Currently there are 12 cgroup v1 controllers and 3 cgroup v2 controllers (*memory*, *io*, and *pids*) and there are other v2 controllers which are under work in progress.
    - For both v1 and v2, these controllers are represented by *cgroup_susbsys* objects.

Mounting cgroups

In order to use the cgroups filesystem (browse it/attach tasks to cgroups, etc.) it must be mounted, as any other filesystem. The cgroup filesystem can be mounted on any path on the filesystem. **Systemd** and various container projects uses */sys/fs/cgroup* as a mounting point.

When mounting, we can specify with mount options (-o) which cgroup controllers will be used (can be a bitmask of controllers, and also all controllers):

Example: mounting the *net_prio* cgroup controller:

mount -t cgroup -onet_prio none /sys/fs/cgroup/net_prio

Kernel 4.4 cgroup v1 – 12 controllers

Name	Kernel Object name	Module
blkio	***io_cgrp_subsys***	block/blk-cgroup.c
cpuacct	***cpuacct_cgrp_subsys***	kernel/sched/cpuacct.c
cpu	***cpu_cgrp_subsys***	kernel/sched/core.c
cpuset	***cpuset_cgrp_subsys***	kernel/cpuset.c
devices	***devices_cgrp_subsys***	security/device_cgroup.c
freezer	***freezer_cgrp_subsys***	kernel/cgroup_freezer.c
hugetlb	***hugetlb_cgrp_subsys***	mm/hugetlb_cgroup.c
memory	***memory_cgrp_subsys***	mm/memcontrol.c
net_cls	***net_cls_cgrp_subsys***	net/core/netclassid_cgroup.c
net_prio	***net_prio_cgrp_subsys***	net/core/netprio_cgroup.c
perf_event	***perf_event_cgrp_subsys***	kernel/events/core.c
pids	***pids_cgrp_subsys***	kernel/cgroup_pids.c

Memory controller interface files

tasks

cgroup.clone_children
cgroup.event_control
cgroup.procs
memory.move_charge_at_immigrate
memory.numa_stat
memory.oom_control
memory.kmem.limit_in_bytes
memory.kmem.max_usage_in_bytes
memory.kmem.slabinfo
memory.kmem.tcp.failcnt
memory.kmem.tcp.limit_in_bytes
memory.kmem.tcp.max_usage_in_bytes
memory.kmem.tcp.usage_in_bytes
memory.kmem.usage_in_bytes
memory.limit_in_bytes

release_agent

Note: the release_agent and sane_behavior appear only on root

notify_on_release
cgroup.sane_behavior
memory.memsw.usage_in_bytes
memory.memsw.max_usage_in_bytes
memory.memsw.failcnt
memory.memsw.limit_in_bytes
memory.pressure_level
memory.soft_limit_in_bytes
memory.stat
memory.swappiness
memory.usage_in_bytes
memory.use_hierarchy
memory.failcnt
memory.force_empty
memory.kmem.failcnt

Example 1: memcg (memory control groups)

mkdir /sys/fs/cgroup/memory/group0

- *The tasks entry that is created under **group0** is empty (*processes are called **tasks** in cgroup terminology).

echo 0 > /sys/fs/cgroup/memory/group0/tasks

- *The pid of the current bash shell process is moved from the memory controller in which it resides into **group0** memory controller group.*

echo 40M > /sys/fs/cgroup/memory/group0/memory.limit_in_bytes

You can disable the **out of memory killer** with memcg:

echo 1 > /sys/fs/cgroup/memory/*group0*/memory.oom_control

memcg (memory control groups) - contd

With containers, the same can be done from the host:

lxc-cgroup -n myfedora memory.limit_in_bytes 40M

cat /sys/fs/cgroup/memory/lxc/myfedora/memory.limit_in_bytes

41943040

Or for assigning the processors 0 and 1 to the container:

lxc-cgroup -n myfedora cpuset.cpus "0,1"

cghost:/$cat /sys/fs/cgroup/cpuset/lxc/myfedora/cpuset.cpus

0-1

docker run -i --cpuset=0,2 -t fedora /bin/bash

cat   /sys/fs/cgroup/cpuset/system.slice/docker-64bit_ID.scope/cpuset.cpus

0,2

Example 2: release_agent in memcg

The release agent mechanism is invoked when the last process of a cgroup terminates.

- The cgroup ***release_agent*** entry should be set to a path to an executable/script to be invoked when the last process in a group terminates.

- The cgroup ***notify_on_release*** entry should be set so that ***release_agent*** will be invoked.

echo 1 > /sys/fs/cgroup/memory/notify_on_release

The release_agent can be set also via a mount option; systemd, for example, use this mechanism. For example in Fedora 21, mount shows:

*cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,**release_agent=/usr/lib/systemd/systemd-cgroups-agent**,name=systemd)*

Example 3: devices control group

Also referred to as : *devcg* (devices control group)

- It is more of an access controller than a resource controller.

● devices cgroup provides enforcing restrictions on reading, writing and creating (mknod) operations on device files.

● 3 control files: **devices.allow**, **devices.deny**, **devices.list**.

– **devices.allow** can be considered as devices whitelist

– **devices.deny** can be considered as devices blacklist.

– **devices.list** available devices.

● Each entry in these files consist of 4 fields:

– **type**: can be a (all), c (char device), or b (block device).

● All means all types of devices, and all major and minor numbers.

– **Major number.**

– **Minor number.**

– **Access:** composition of 'r' (read), 'w' (write) and 'm' (mknod).

devices control group – example (contd)

/dev/null major number is 1 and minor number is 3 (see *Documentation/devices.txt*)

mkdir /sys/fs/cgroup/devices/group0

By default, for a new group, you have full permissions:

cat /sys/fs/cgroup/devices/group0/devices.list

*a *:* rwm*

echo 'c 1:3 rmw' > /sys/fs/cgroup/devices/group0/devices.deny

This denies rmw access from ***/dev/null*** device.

echo 0 > /sys/fs/cgroup/devices/group0/tasks # Attaches the current shell to group0

echo "test" > /dev/null

bash: /dev/null: Operation not permitted

PIDs cgroup controller

- An anti-fork-bomb solution by a new cgroup controller.
- Adds a cgroup controller to enable **limiting the number of processes** that can be forked inside a cgroup.
- Implementation of the prlimit(2)/RLIMIT_NPROC but to a cgroup and not to a process.

- The PIDs space is a limited resource space, about 4 million pids system-wide.
- PID_MAX_LIMIT=**4,194,304** (0x400000) see: *include/linux/threads.h*.
  - With nowadays RAM capacities, all of them can be used up by a single container, making the whole system unusable. The PIDs cgroup controller can help avoiding this by limiting the number of processes per cgroup.
- Developed by Aleksa Sarai and integrated into the kernel since **v4.3**
  - A simple and short module, ~300 lines of code only, *kernel/cgroup_pids.c*

PIDs cgroup controller - contd

The PID cgroup controller has two interface files:

- pids.max (Read/Write)
  - The maximum number of processes for the cgroup.
  - Does not exist for the root cgroup directory.
  - For subgroups, the value is “max”, which is about 4,000,000.
- pids.current (Read only)
  - The number of processes currently in the cgroup and its children
    - Also of processes of a child on which PIDs controller is not enabled.
    - Does not include zombies.
    - Does not exist for the root cgroup directory
- *When the number of processes in a group exceeds its pids.max, you will get this error: fork: Resource temporarily unavailable.*

cgroup v2 - background

- cgroup v1 is the existing cgroup kernel implementation (also referred to as **legacy cgroup** in cgroup/LKML/other mailing lists).
- “Unified Hierarchy” development features are available over three years with special mount option (__**DEVEL__sane_behavior**).
- From kernel 4.4 (January 2016), cgroup V2 is an official part of the kernel with special filesystem type.
- *Systemd* has initial support to cgroup v2 since v226, September 2015.
- Also *cgmanager* added initial support for cgroup v2 (by Serge Hallyn)
- **Why cgroup v2 ?**
  - A lot of chaos in cgroupv1.

cgroup v2 - background

- **no consistency** across cgroup controllers; for example:
- When creating a **new** controller, in several controllers the attribute values are inherited from the parent (like *net_prio* and *net_cls*), and in several others the attribute values are the defaults, regardless of the parent.
- For *cpuset*, changes in the parent are propagated to its descendants, whereas with all controllers they are not (clone_children should be set for this),
  - With **cgroup v1**, some controllers have controller-specific interface files in the root cgroup while others don't have.
  - **cgroup v2** establishes a strict and consistent interfaces

- In cgroup v2, there is only one hierarchy, "**the unified hierarchy**". Each process can belong to only a single cgroup. cgroup-v2.txt in kernel Documentation describes in detail cgroup v1 inconsistencies:

tree -L 1 /sys/fs/cgroup/ on Fedora 23: (cgroup v1 – multiple hierarchies)

```
├── blkio
├── -> cpu,cpuacct
├── cpuacct -> cpu,cpuacct
├── cpu,cpuacct
├── cpuset
├── devices
├── freezer
├── hugetlb
├── memory
├── net_cls -> net_cls,net_prio
├── net_cls,net_prio
├── net_prio -> net_cls,net_prio
├── perf_event
└── systemd
```

Unlike cgroup v1, cgroup v2 has only a **single** hierarchy and is strict about hierarchical behavior.

- Enabling/Disabling of a controller is done always **by the cgroup parent** rather than by the cgroup itself (*subtree_control*).
- Interface files - semantics
  - When a controller implements an absolute resource limit, the interface files should be named **"min"** and **"max“** respectively (for example, ***pids.max*** for the PIDs controller)
  - When a controller implements best effort resource limit, the interface files should be named **"low“** and **"high"** respectively.
    - used for example, in the memory controller.
  - A special token "max" is used for these interface files (write/read), representing infinity.
- Mounting cgroup v2 is done by:
  - *mount -t cgroup2 none $MOUNT_POINT*
  - The mount point can be anywhere in the filesystem.

cgroup v2 controllers

- As opposed to cgroups v1, there are **no cgroup mount options** in cgroup v2**.**
- When the system boots, **both** cgroup v1 filesystem and cgroup v2 filesystem are registered, so you can work with a mixture of cgroup v1 and cgroup v2 controllers.
- You **cannot** use the same type of controller simultaneously both in cgroup v1 and cgroup v2.

The root cgroup object

- After mounting /cgroup2 with *mount -t cgroup2 none /cgroup2*, a **root cgroup** object is created.

There is a single root cgroup object, and it does not have any resource control interface files.

- The following three cgroup core interface files are created under the /cgroup2 mount point:

```
/cgroup2/
├── cgroup.controllers
├── cgroup.procs
└── cgroup.subtree_control
```

Next we will describe these three cgroup core interface files.

The root cgroup object – contd.

- ***cgroup.controllers*** *(*A read-only file).
  - Shows the supported cgroup controllers. In cgroup v2, we have currently support for the **memory, io** and **pids** cgroup controllers only. All v2 controllers which are not bound to a v1 hierarchy are automatically bound to the v2 hierarchy and show up at the root, so reading *cgroup.controllers* will give *io memory pids*
- ***cgroup.procs*** *(*A read-write file)
  - The list of PIDs of processes in the group; contains the PIDs of all processes in the system after mount (zombie processes do not appear in "cgroup.procs" and thus can't be moved to another cgroup).

The root cgroup object – contd.

- ***cgroup.subtree_control*** *(*A read-write file.)
  - This entry is empty after mount, as no controller is enabled by default.
  - Enabling cgroup v2 controller is done by writing to ***cgroup.subtree_control***.
    - For example, enabling the memory controller is done by:
      - ***echo "+memory" > /cgroup2/ cgroup.subtree_control***
  - Disabling the memory controller can be done, for example, by:
    - ***echo "-memory" > /cgroup2/ cgroup.subtree_control***

Creating a subgroup

- Creating a cgroup is done by ***mkdir*** (like in cgroup v1), for example:
  - *mkdir /cgroup2/group1*
- After running this command, four cgroup core "*cgroup.*" prefixed entries are created and also several interface files for the cgroup controllers enabled in the parent, as we will immediately see.

```
group1/
├── cgroup.controllers
├── cgroup.procs
├── cgroup.events
└── cgroup.subtree_control
```

The set of these 4 cgroup interface files is the v2 hierarchy itself and is referred to internally as **"the default hierarchy"**.

All cgroup core interface files are prefixed with "cgroup."

Next we will see the entries which are created when running "*mkdir /cgroup2/group1*".

subgroup interface files

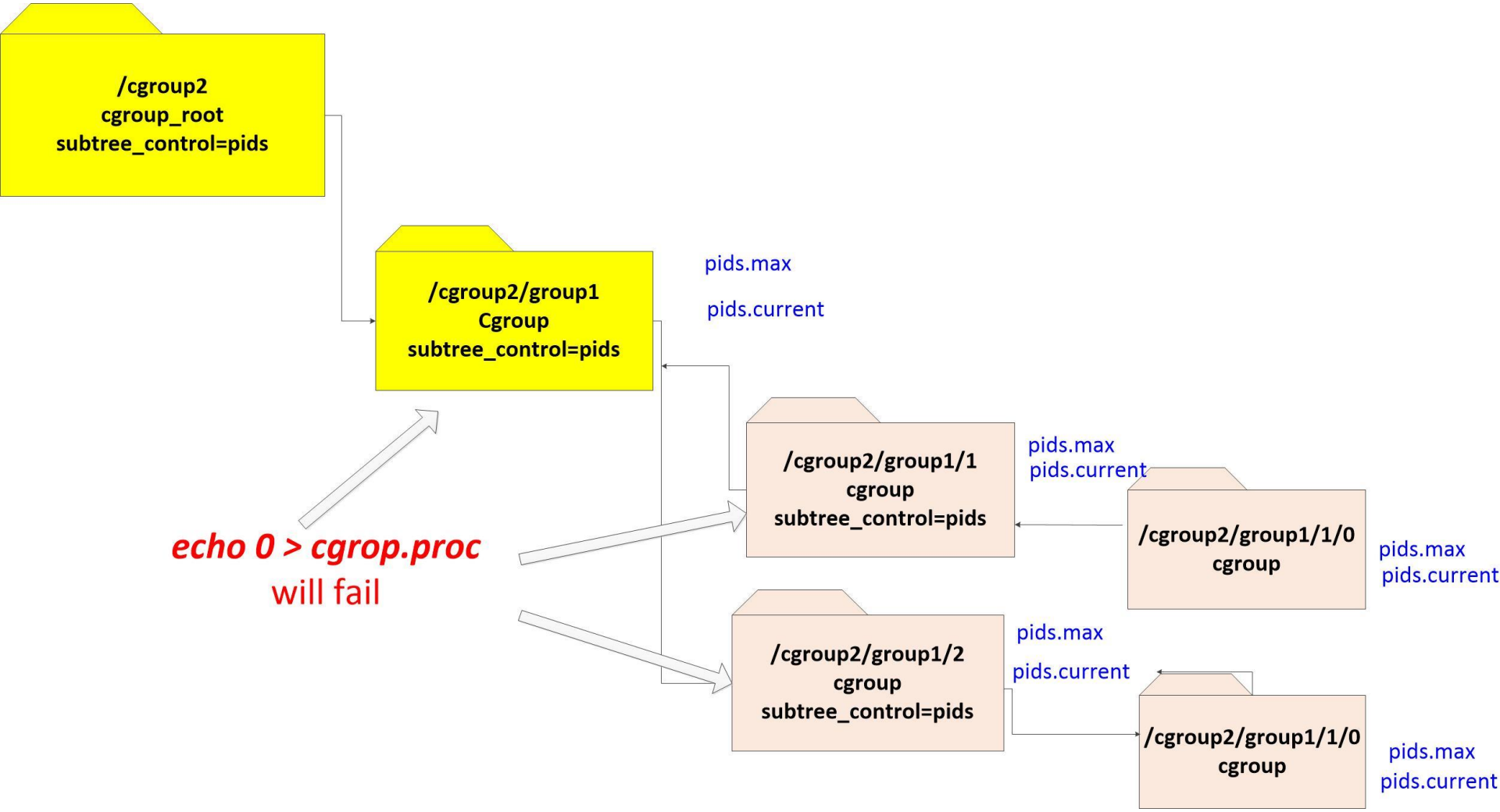
- ***/cgroup2/group1/cgroup.procs***
  - The list of PIDs of processes in this group; empty upon creation.
- ***/cgroup2/group1/cgroup.controllers***
  - The subgroup enabled controllers. For subgroups, this will show the controllers that were enabled in the parent by writing to *subtree_control*. When changing the *subtree_control* in the parent, changes are propagated to the child *cgroup.contorllers*
- ***/cgroup2/group1/cgroup.subtree_control***
  - Initialized to be empty for the child group. Also here, you can enable only controllers which appear in the *cgroups.controllers* of this cgroup (group1).
- ***/cgroup2/group1/cgroup.events***
  - *Contains only one field, "populated"; **0** means **no live process in this cgroup and its descendants**, **1** otherwise. Upon creation of a subgroup, populated is 0*
  - monitoring changes of *populated* from userspace - with ***poll()***, ***inotify()*** and ***dnotify()*** , as opposed to *call_usermodehelper() in cgroup v1.*

No Internal Process rule

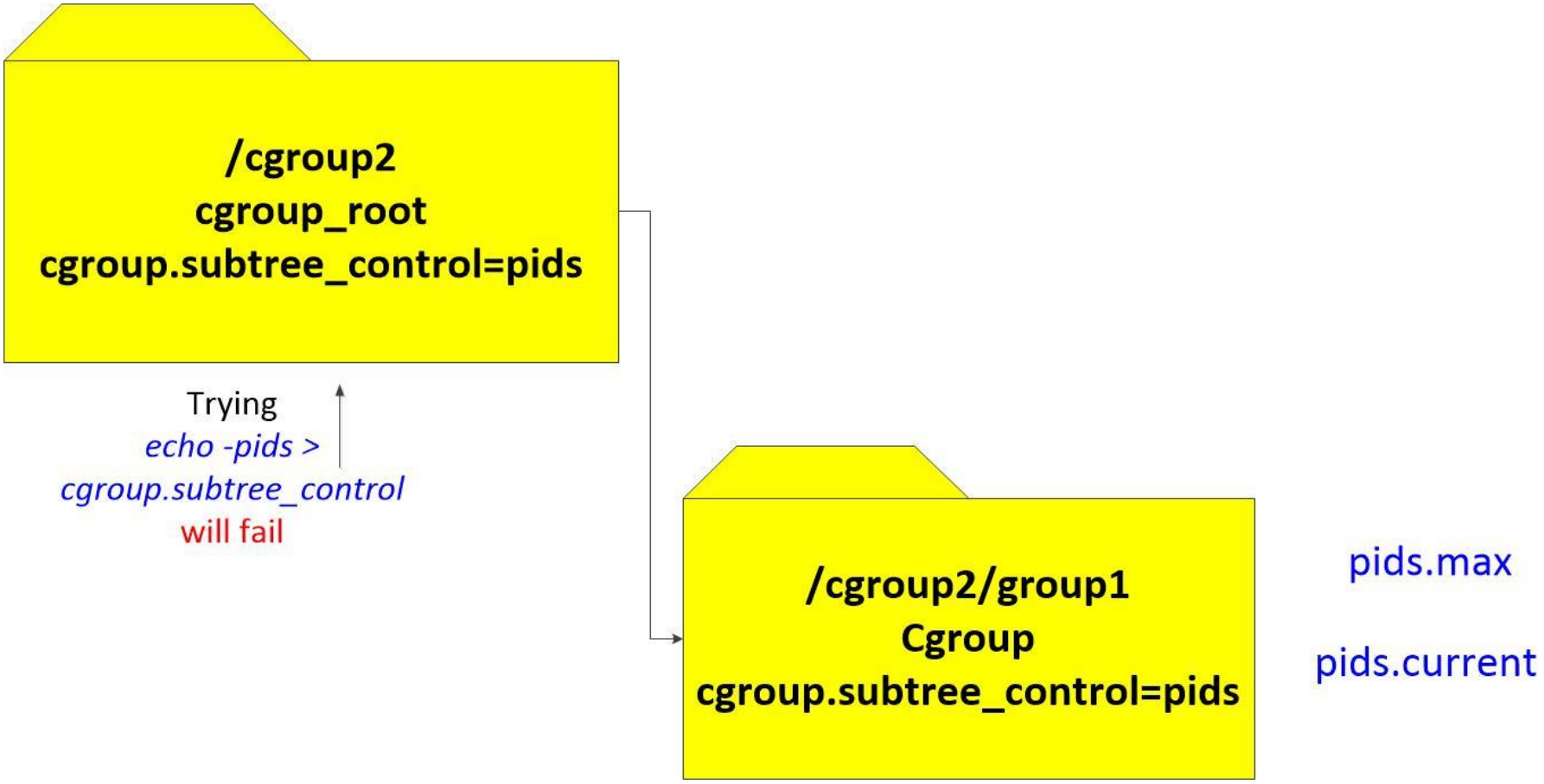
You can attach processes only to leaves.

You **cannot** attach a process to an internal subgroup if it has any controller enabled.

- Controllers can be enabled by either writing to *subtree_control* of the parent or implicitly via controllers dependency.
- The idea is that only processes of the leaves can compte on resources. This scheme is more well organized.
- The only exception for this is the cgroup root object.
  - This is opposed to cgroup v1, which allowed threads to be in any cgroups
  - See “2-4-3. No Internal Process Constraint” in cgroup-v2.txt.


/cgroup2
cgroup_root
subtree_control=pids
/cgroup2/group1
Cgroup
subtree_control=pids
pids.max
pids.current
/cgroup2/group1/1
cgroup
subtree_control=pids
pids.max
pids.current
/cgroup2/group1/1/0
cgroup
pids.max
pids.current
echo 0 > cgrop.proc
will fail
/cgroup2/group1/2
cgroup
subtree_control=pids
pids.max
pids.current
/cgroup2/group1/1/0
cgroup
pids.max
pids.current

A controller can't be disabled if one or more descendants have it enabled.



cgroup2 example

multi-destination migration as a result of *subtree_control* enabling:

Run this sequence:

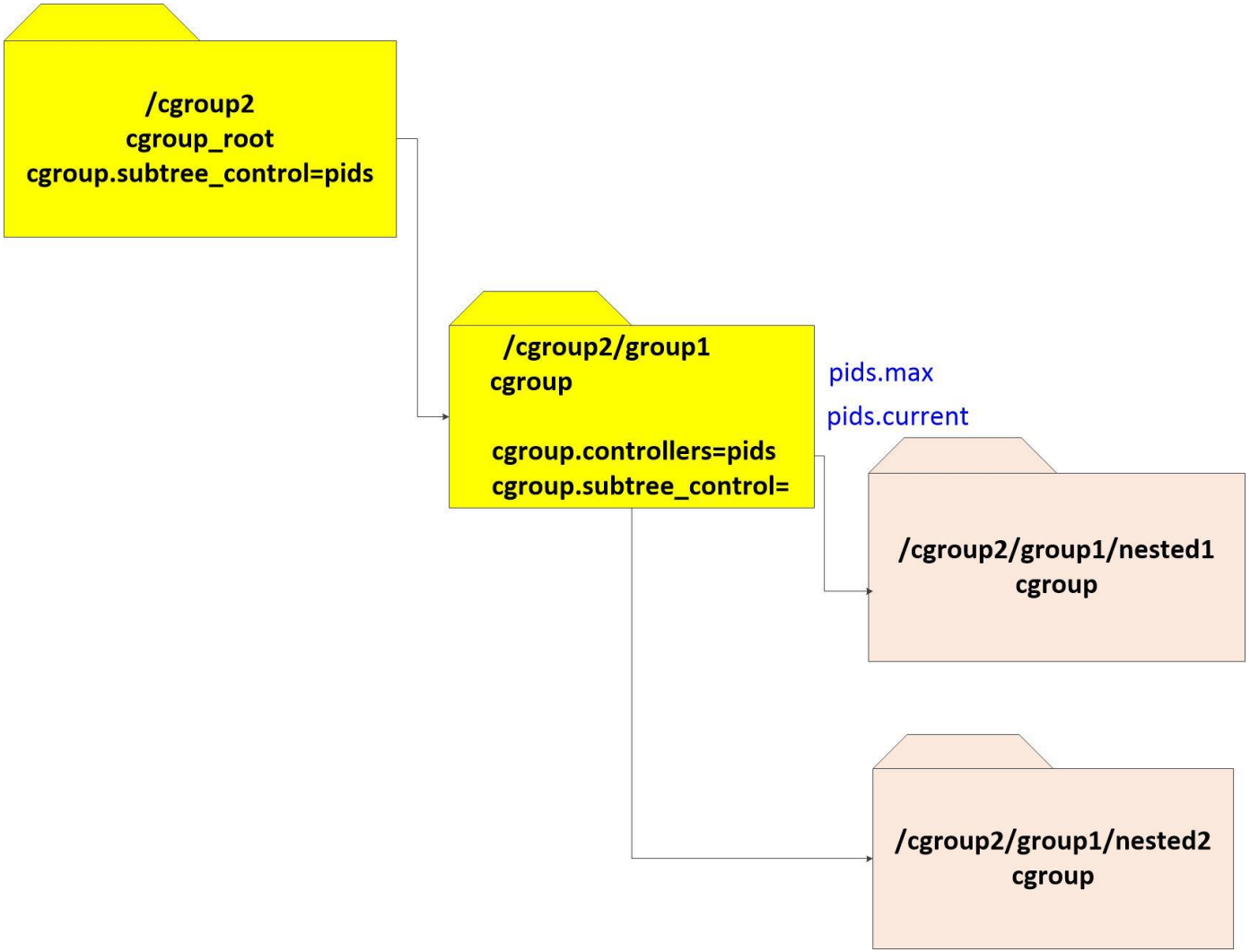
mount –t cgroup2 nodev /cgroup2

mkdir /cgroup2/group1

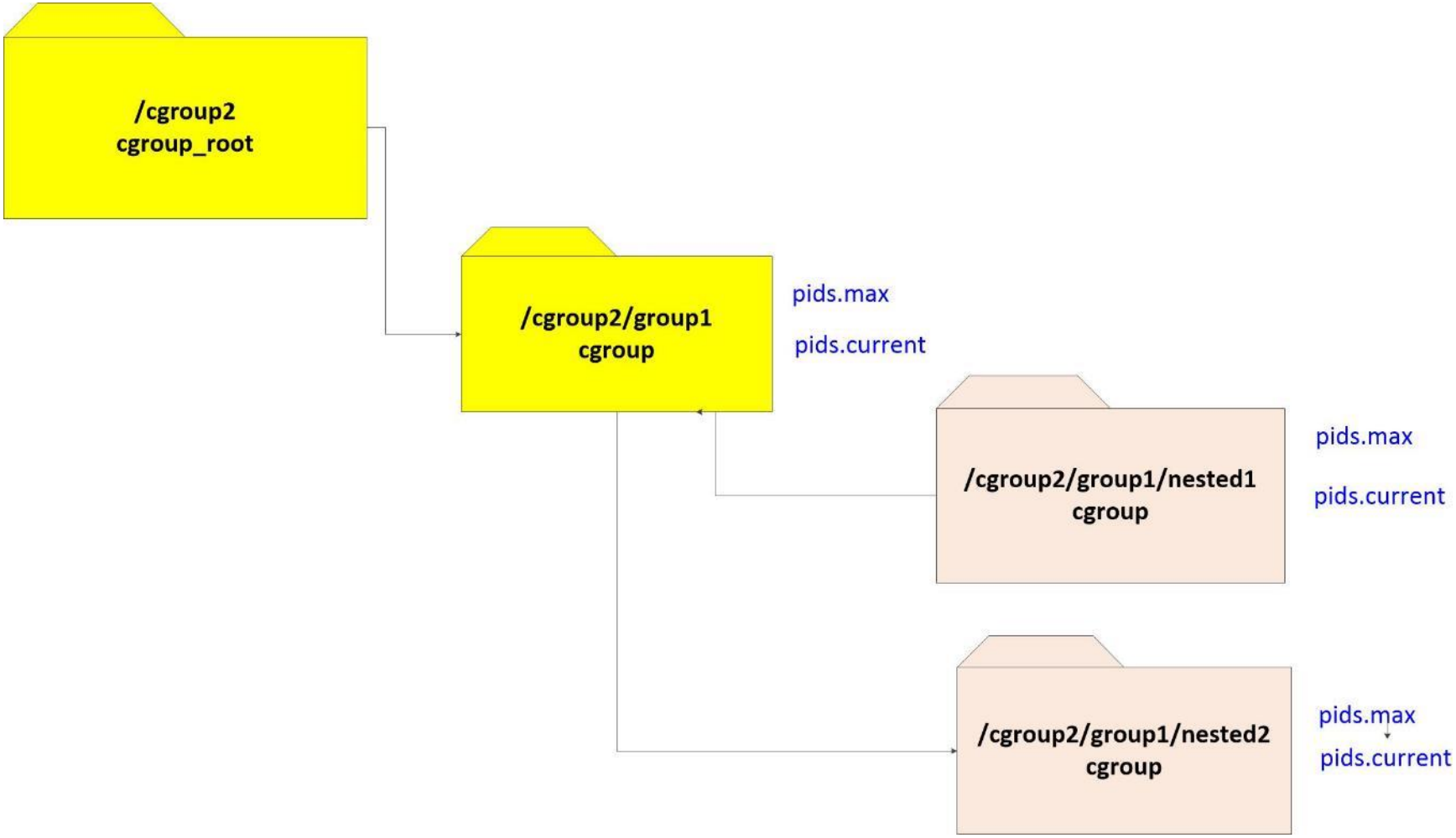
mkdir /cgroup2/group1/nested1

mkdir /cgroup2/group1/nested2

echo +pids > cgroup2/cgroup.subtree_controll


/cgroup2
cgroup_root
cgroup.subtree_control=pids
/cgroup2/group1
cgroup
cgroup.controllers=pids
cgroup.subtree_control=
pids.max
pids.current
/cgroup2/group1/nested1
cgroup
/cgroup2/group1/nested2
cgroup

echo +pids > cgroup2/group1/cgroup.subtree_controller



- What happens if there were processes in *nested1* and *nested2*

  before running:

  echo +pids > cgroup2/group1/cgroup.subtree_controller?

  - An inner *cgroup_subsys_state* (css) object is created for that group.
  - The processes in *nested1* and *nested2* should be migrated to this css.
    - With PIDs controller, attaching a process to a cgroup will never fail
    - For other controllers, there are cases when the attaching a process to a cgroup will fail, for example, when the CLONE_IO is set (for cgroup v1 as well as cgroup v2).
    - Migrating the processes should also handle implicit controllers.

Migrating processes and threads

- **cgroup v2 is process-granular.**
- Every process in the system belongs to **one and only one** cgroup.
- All threads of a process belong to the **same** cgroup.
- A process can be migrated into a cgroup by writing its PID to the target cgroup's *cgroup.procs* file.
- Writing the PID of any thread of a process to *cgroup.procs* of a destination cgroup migrates **all** the threads of the process into the destination cgroup (including the main process).
- This is opposed to **cgroup v1 thread granularity**, which allowed different threads of a process to belong to different cgroups.
- When forking other processes from inside a process, migrating of a parent process to another cgroup does not affect the existing child processes, and migrating of a child process does not affect the parent process.

cgroup v2- match subgroup by path

Finding matches based on the cgroup name in cgroup v2 (which is done by the --path parameter) is based on getting the cgroup to which the process holding the socket belongs. Practically, this mechanism is not possible in cgroup v1 as a process can belong to more than one cgroup. In cgroup v2 we have only the match by path capability. An example for a rule for matching traffic which originates from sockets created in a group called “test”, or its subgroups, can be

iptables -A OUTPUT -m cgroup --path test -j LOG

This is based to extension of the *xt_cgroup* netfilter match module (adding a new revision) , *net/netfilter/xt_cgroup.c.* The *xt_cgroup* module can be used both for cgroup v1 and cgroup v2.

Create a cgroupv2 group named **test** and move the current shell to it:

mkdir /cgroup2/test

echo 0 > /cgroup2/test/cgroup.procs

Now every socket created in this shell will have a pointer to the cgroup subgroup in which it was created, namely the “test” group.

A ***sock_cgroup_data*** object, which contains per-socket cgroup information:

was added to the ***sock*** object. It includes:

- A pointer to the *cgroup* in which the socket is created.
  - assigned when the socket is created
- prioidx
- classid
- is_data – 0 indicates a cgroup pointer , 1 indicates prioidx or classid.
  - *Note: once net_prio or net_class will be used, that pointer in the socket will no longer point to the cgroup, but to the priority or classid.*

cgroup v2- match cgroup example -contd

rr:/$echo 5 > /sys/fs/cgroup/net_prio/net_cls.classid

rr:/$mkdir /sys/fs/cgroup/net_prio/group1

/sys/fs/cgroup/net_prio/group1/net_cls.classid will be 5 as it is inherited.

echo $$ > /sys/fs/cgroup/net_prio/group1/tasks

iptables -A OUTPUT -m cgroup --cgroup 5 -j LOG

This will trigger again logging the packets to syslog.

Namespaces

Development took over a decade: Namespaces implementation started in about **2002**. There are currently 6 namespaces in Linux:

- **mnt** (mount points, filesystems)
- **pid** (processes)
- **net** (network stack)
- **ipc** (System V IPC)
- **uts** (hostname)
- **user** (UIDs)

A process can be created in Linux by the ***fork()***, ***clone()*** **or** ***vclone()*** system calls. In order to support namespaces, 6 flags (CLONE_NEW*) were added. These flags (or a combination of them) can be used in ***clone()*** or ***unshare()*** system calls to create a namespace.

Namespaces clone flags

Clone flag	Kernel Version	Required capability
CLONE_NEWNS	2.4.19	CAP_SYS_ADMIN
CLONE_NEWUTS	2.6.19	CAP_SYS_ADMIN
CLONE_NEWIPC	2.6.19	CAP_SYS_ADMIN
CLONE_NEWPID	2.6.24	CAP_SYS_ADMIN
CLONE_NEWNET	2.6.29	CAP_SYS_ADMIN
CLONE_NEWUSER	3.8	No capability is required

Namespaces system calls

Namespaces API consists of these 3 system calls:

● ***clone()*** **-** creates a **new process** and a **new namespace**; the newly created process is attached to the new namespace.

– The process creation and process termination methods, ***fork()*** and ***exit()***, were patched to handle the new namespace CLONE_NEW* flags.

● ***unshare()*** **– gets only a single parameter, flags.** Does **not** create a new process; creates **a new namespace** and **attaches** the **calling process** to it.

– ***unshare()*** was added in 2005.

see “new system call, unshare” : http://lwn.net/Articles/135266/

● ***setns()*** **-** a new system call, for attaching the calling process to an existing namespace; prototype: *int setns(int fd, int nstype);*

UTS namespace

UTS namespace provides a way to get information about the system with commands like ***uname*** or ***hostname***.

UTS namespace was the most simple one to implement.

There is a member in the process descriptor called nsproxy.

A member named **uts_ns** (uts_namespace object) was added to it.

The uts_ns object includes an object (new_utsname struct) with 6 members:

- sysname
- nodename
- release
- version
- machine
- domainname

In order to implement the UTS namespace, usage of global variables was replaces by accessing the corresponding members in the UTS namespace via the new *nsproxy* object of the process descriptor.

See for example the implementation of ***gethostname() , sethostname()*** *and* ***uname()*** syscalls.

For **IPC namespaces**, the same principle as in UTS namespace.

For **Mount namespace**, a member named **mnt_ns** (mnt_namespace object) was added to the nsproxy.

- In the new mount namespace, all previous mounts will be visible; and from now on, mounts/unmounts in that mount namespace are invisible to the rest of the system.

- mounts/unmounts in the global namespace are visible in that namespace.

PID namespaces

● Added a member named **pid_ns** (pid_namespace object) to the *nsproxy* struct.

● Processes in different PID namespaces can have the same process ID.

● When creating the first process in a new PID namespace, its PID is 1.

● Behavior like the “init” process:

– When a process dies, all its orphaned children will now have the process with PID 1 as their parent (**child reaping**).

– Sending **SIGKILL** signal does not kill process 1, regardless of in which namespace the command was issued (initial namespace or other pid namespace).

● pid namespaces can be nested, up to 32 nesting levels. (MAX_PID_NS_LEVEL)

- A usecase for PID namespaces: the CRIU project

User Namespaces

- Added a member named ***user_ns*** (user_namespace object) to the credentials object (struct ***sched***).
- Also each namespace includes a pointer to a *user_namespace* object.

Each process will have distinct set of UIDs, GIDs and capabilities.

User namespace enables non root user to create a process in which it will be root.

Anatomy of a user namespaces vulnerability:

By Michael Kerrisk, March 2013

An article about CVE 2013-1858 - exploitable security

vulnerability

http://lwn.net/Articles/543273/

Network Namespaces

● A network namespace is logically another copy of the network stack,

with its own routing tables, firewall rules, and network devices.

● The network namespace is represented by a huge **struct net.** (defined in *include/net/net_namespace.h*).

Method in the stack which change the state were adjusted to have the net object as a parameter and set its members accordigly.

struct **net** includes all network stack ingredients, like:

- – Loopback device.
- – SNMP stats. (netns_mib)
- – All network tables: routing, neighboring, etc.
- – All sockets
- – /procfs and /sysfs entries.

At a given moment -

- **A network device belongs to exactly one network namespace.**
- **A socket belongs to exactly one network namespace.**

The default initial network namespace, **init_net** (instance of struct net), includes the loopback device and all the physical devices, the networking tables, etc.

- Each **newly** created network namespace includes only the loopback device.

Example

Create two namespaces, called "myns1" and "myns2":

- ***ip netns add myns1***
- ***ip netns add myns2***
  - This triggers creation of */var/run/netns/myns1,/var/run/netns/myns2* empty **folders** and invoking the ***unshare()*** system call with CLONE_NEWNET.

You delete a namespace by:

- ***ip netns del myns1***

– This unmounts and removes /var/run/netns/myns1

You can **move** a network interface (eth0) to myns1 network namespace:

- ***ip link set eth0 netns myns1***

There are other subcommands for monitoring namespaces, running a command in a namespace/in all namespaces, listing the network namespaces **which were created with the ip netns command and more:**

ip netns monitor, ip netns exec myns1 bash, ip -all netns exec ip link, ip netns list

Applications which usually look for configuration files under /etc (like /etc/hosts or /etc/resolv.conf), will first look under /etc/netns/NAME/, and only if nothing is available there, will look under /etc.

cgroup namespace

- Provides a new namespace, which gives the ability to remember the cgroup of the process at the point of creation of the cgroup namespace. **Supports both cgroup v1 and cgroup v2.**
- Implementation: a new CLONE flag was added, CLONE_NEWCGROUP, and a new object was added to ***nsproxy***, representing cgroup namespace, ***cgroup_ns.***

The ***/proc/<pid>/cgroup*** file may leak potential system level information to the isolated processes. For example, without cgroup namespaces:

$ cat /proc/self/cgroup

*0:cpuset,cpu,cpuacct,memory,devices,freezer,hugetlb**:/batchjobs/container_id1***

But with cgroup namespaces, from within new cgroupns,

cat /proc/self/cgroup

0:cpuset,cpu,cpuacct,memory,devices,freezer,hugetlb:/

Legal Disclaimer

Thank You

Backup

Implicit controllers (cgroup v2)

- **The ability to make one controller dependent on another is one of the new features of cgroup v2.**
  - **controller dependency is not possible in cgroup v1.**
- **This can be defined in code by setting the *depends_on* member of the *cgroup_subsys.***
- **For example, the *io* controller depends on the *memory* controller.**

```
struct cgroup_subsys io_cgrp_subsys = {
...
    .depends_on = 1 << memory_cgrp_id,
...
};
```

- **This means that enabling *'io*' enables *'memory*' *implicitly*, but it is not visible (no interface files)**

Example 1 (cgroup v1 propagation)

The *net_cls* controller – when creating new cgroup, the *net_cls.classid* value is propagated to the existing subgroups:

rr:/sys/fs/cgroup/net_cls$ *mkdir group1*

rr:/sys/fs/cgroup/net_cls/group1$ *echo 0x2 > net_cls.classid*

r:/sys/fs/cgroup/net_cls/group1$ *mkdir nested1*

rr:/sys/fs/cgroup/net_cls/group1$ *cat nested1/net_cls.classid*

2

Note: after the child groups are created, changes in the parent are not propagated to the existing child groups:

rr:/sys/fs/cgroup/net_cls/group1$ *echo 0x1 > net_cls.classid*

rr:/sys/fs/cgroup/net_cls/group1$ *cat nested1/net_cls.classid*

2

example 2 (cgroup v1 clone_children)

The Following sequence shows propagation from parent **when creating a new group:**

rr:/sys/fs/cgroup/cpuset$ *echo 1 > cgroup.clone_children*

rr:/sys/fs/cgroup/cpuset$ *mkdir group1*

rr:/sys/fs/cgroup/cpuset$ *echo 1-2 > group1/cpuset.cpus*

rr:/sys/fs/cgroup/cpuset$ *cat group1/cpuset.cpus*

1-2

rr:/sys/fs/cgroup/cpuset$ *cat group1/nested1/cpuset.cpus*

1-2

Notes:

Without echo 1 > cgroup.clone_children this propagation ***won't*** *work.*

The clone_children is effective ***only*** *with the cpuset controller.*

Example 3 – cgroup v1 net_prio

s:/$mkdir /sys/fs/cgroup/net_prio/group1

s:/$echo "eth0 4" > /sys/fs/cgroup/net_prio/group1/net_prio.ifpriomap

This sets the *netprio_map* object of eth0 net_device.

This will set the priority of outgoing (egress) traffic of packets (skbs) of processes attached to group1 to be 4.

This is implemented by the *skb_update_prio()* method

http://lxr.free-electrons.com/source/net/core/dev.c#L2926

This is done prior to queuing the packet with the qdisc (Queuing Discipline).

Setting the priority of a socket can be done by setting the SO_PRIORITY socket option, but this option is not always available.

Example 4 : Delegation Containment

cgroup v1:

Run as root:

mkdir -p /sys/fs/cgroup/devices/group1/nested1

su user1

echo $$ > /sys/fs/cgroup/devices/group1/nested1/cgroup.procs

You will get –EPERM

But after you will set access permission to nested1 by running as root:

chown -R user1:user1 /sys/fs/cgroup/devices/group1/nested1/

Running it will succeed.

Example 4 : delegation containment – v2

In order to support delegation, three conditions should be met: *the writer's euid must match either uid or suid of the target process.The writer must have write access to the "cgroup.procs" file. The writer must have write access to the "cgroup.procs" file of the common ancestor of the source and destination cgroups.*

Run as root:

```
echo $$

4767

mkdir -p /cgroup2/group1/nested1

chown –R user1:user1 /cgroup2/group1

echo $$  > /cgroup2/group1/cgroup.procs

su user1

echo 4767  > /cgroup2/group1/nested1/cgroup.procs
```

Getting info about cgroups

How do I know to which cgroup does a process belong to?

cat "/proc/$PID/cgroup" shows this info.

- The entry for cgroup v2 is always in the format **"0::$PATH".**
- So for example, if we created a cgroup named group1 and attached a task with PID 1000 to it, then running:

cat "/proc/1000/cgroup

0::/group1

And for a nested group:

cat /proc/869/cgroup

0::/group1/nested

/proc/cgroups shows info on both cgroup v1/cgroup v2. The hierarchy_id for cgroupv2 controllers is 0.

#subsys_name	hierarchy	num_cgroups	enabled
cpuset	2	1	1
cpu	3	1	1
cpuacct	3	1	1
blkio	0	1	1
memory	0	1	1
devices	6	61	1
freezer	7	1	1
net_cls	8	1	1
perf_event	9	1	1
net_prio	8	1	1
hugetlb	10	1	1
pids	0	1	1

Cgroup v2 Control files (Interface files)

For the memory controller:

- *memory.current*
- *memory.events*
- *memory.high*
- *memory.low*
- *memory.max*

- For the IO controller:
  - *io.max*
  - *io.weight*

Note: Work is currently being done by Vladimir Davydov, one of the Memory Resource Controller ***(memcg)*** maintainers, for adding cgroup v2 kmem accounting, so entries like memory.swap.current, memory.swap.max are coming soon.

Minor advantage: There is a single Linux kernel infrastructure for containers (namespaces and cgroups) while for Xen an KVM we have two different implementations without any common code.

Now run the following iptables rule (from anywhere in the system):

iptables -A OUTPUT -m cgroup --path test -j LOG

And then ping anywhere from the shell that is now a process in “test”.

The socket created has a pointer to “test” v2 cgroup, and the iptables match rule is for “test” (--path test). So the packets will be dumped to syslog.

- Running this test from any subgroup of test will have the same result.

- The sum of the allocations of **immediate subgroups** can not exceed the amount of resources available to the parent.

So, for example:

If in /cgroup2/group1

pids.max = 4

Then if in

/cgroup2/group1/nested1

/cgroup2/group1/nested2

There are together 4 processes, we cannot fork a fifth in either of them.

Script for connecting two namespaces

The script in the following slide shows how to connect two namespaces by veth (Virtual Ethernet drivers) so that you will be able to ping and send traffic between them:

Script for connecting two namespaces

```
ip netns add netA
ip netns add netB
ip link add name vm1-eth0 type veth peer name vm1-eth0.1
ip link add name vm2-eth0 type veth peer name vm2-eth0.1
ip link set vm1-eth0.1 netns netA
ip link set vm2-eth0.1 netns netB
ip netns exec netA ip l set lo up
ip netns exec netA ip l set vm1-eth0.1 up
ip netns exec netB ip l set lo up
ip netns exec netB ip l set vm2-eth0.1 up
ip netns exec netA ip a add 192.168.0.10 dev vm1-eth0.1
ip netns exec netB ip a add 192.168.0.20 dev vm2-eth0.1
ip netns exec netA ip r add 192.168.0.0/24 dev vm1-eth0.1
ip netns exec netB ip r add 192.168.0.0/24 dev vm2-eth0.1
brctl addbr mybr
ip l set mybr up
ip l set vm1-eth0 up
brctl addif mybr vm1-eth0
ip l set vm2-eth0 up
brctl addif mybr vm2-eth0
```

Upcoming…

RDMA cgroups controller – patches were already sent by Parav Pandit:

https://lwn.net/Articles/674161/

The RDMA cgroup will support both V1 and V2.