Cgroups V2 overview
Linux Plumbers Conference (LPC) 2016, Santa Fe, New Mexico
(Including a demo)

Rami Rosen
About Myself:

Tech Leader at Intel, working in various projects, primarily Kernel networking.

My website: [http://ramirose.wix.com/ramirosen](http://ramirose.wix.com/ramirosen)

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

- Cgroups V2 – Background
- Cgroups V2 – Current Status
- Cgroups V2 Controllers
- Demo

Backup slides
cgroup v2 - background

- Cgroup V2 development started about 4 years ago (September 2012) by Tejun Heo.
- Cgroup v1 is the existing cgroup kernel implementation.
- “Unified Hierarchy” development features are available over three years with special mount option (\_DEVEL\_sane\_behavior).
- This mount option is removed now.
- From kernel 4.5, cgroup V2 is an official part of the kernel with special filesystem type (cgroup2)
- **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.
  - As the maintainer admitted, “sometimes design followed implementation”.
Sometimes too much flexibility causes a hindrance.

Control groups are one of those features that kernel developers love to hate.

"Fixing control groups", https://lwn.net/Articles/484251/

**no consistency** across cgroup controllers; for example:

- When creating a **new** controller, in several controllers the attribute values are inherited from parent (like `net_prio` or `net_cls`), and in several others the attribute values are the defaults, regardless 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

  - **cgroup-v2.txt** in the kernel Documentation describes in detail cgroup v1 inconsistencies.

- In cgroup v2, there is **one hierarchy**, “the unified hierarchy”. **A process belongs to a single cgroup.**
Currently, 3 cgroup v2 controllers are in the mainline kernel:

- **Memory, IO and PIDs.**

Work is currently being done also for the following Cgroup V2 controllers:

- **RDMA** controller (by Parav from Emulex) – submitted to the cgroup V2 mailing list, under discussion.
- **Freezer** (Calvin Owens, Facebook). Hopefully it will be ready at the end of 2016.
The Cgroup v2 CPU Controller

• There was an objection from the kernel scheduler developers to the Cgroup v2 CPU Controller patches.
• The Cgroup V2 model by which all of a process's threads must be in the same cgroup does not comply, so it seems, with CPU controller use cases, according to the scheduler developers.
• Also they did not like the “no internal process” rule (see later).
• The CPU controller, together with the IO and the Memory Controllers, are the most important controllers.
  • See this lwn.net article, “The case of the stalled CPU controller”, Jonathan Corbet, 2016.
    • https://lwn.net/Articles/697366/
    • Also a lengthy thread in the cgroup mailing list, which started by a status summary update, sent by Tejun:
      • “State of CPU controller in cgroup v2”
      • More than 45 posts: https://lkml.org/lkml/2016/8/5/368
      • A “resource group” patch set was suggested by Tejun Heo several months ago, but it seems that it is not got momentum. https://lwn.net/Articles/679774/
The Cgroup v2 CPU Controller - contd

- Currently the implementation of Cgroup V2 CPU Controller exists as an out of tree repo.

- It seems that distributions like Fedora will not want to adopt an out of tree solution.
  - It seems also that Systemd upstream will not move to cgroup v2 till at least it will be supported by major application like Docker, LXC and libvirt without any issues.
  - Anyhow, it seems that Cgroup V1 and Cgroup V2 will coexist in the kernel till application will no longer support Cgroup V1. (like we had in the past with iptables and ipchains).
Single Hierarchy in Cgroup V2

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.
• In cgroup v2, 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.controllers`.

- /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 compete 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.
/group2
cgroup_root
subtree_control=pids

/group2/group1
Cgroup
subtree_control=pids

pids.max
pids.current

echo 0 > cgrop.proc
will fail

pids.max
pids.current

pids.max
pids.current

pids.max
pids.current

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

```
/cgroup2
cgroup_root
cgroup.subtree_control=pids
```

```
Trying
echo -pids >
cgroup.subtree_control
will fail
```

```
/cgroup2/group1
Cgroup
cgroup.subtree_control=pids
```

```
pids.max
pids.current
```
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_control
```
echo +pids > cgroup2/group1/cgroup.subtree_control
• What happens if there were processes in *nested1* and *nested2* before running:

```bash
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 cgrops.
- 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.
Demo

Setup

• **Ubuntu 15.04 server**
  • This is the first Ubuntu release where system was enabled, and is on by default.
  • Systemd uses cgroup v1 controllers
  • Remove systemd with:
    • `apt-get install upstart-sysv -y`
      • This triggers removing ubuntu-standard and systemd-sysv
    • `update-initramfs -u`

• **Kernel 4.6**
  • Added `cgroup_no_v1=all` to the kernel command line (in grub config file) to disable loading of v1 controllers.
    • The `cgroup_no_v1=all` kernel parameter was first added in kernel 4.6.
  • Mount group2 filesystem on `/cgroup2` (it can be anywhere on the filesystem, including `/sys/fs/cgroup`).
Build **iptables** from the git repo:

- Needed for cgroup match module, extensions/libxt_cgroup.so
  - Copied it to /lib/xtables/
  - Work was done both in the userspace module and kernel module to support cgroup V2.
Steps

Mount cgroup v2 Virtual Filesystem on /cgroup2: (1_mount.sh)

$ mount -t cgroup2 none /cgroup2

This results with creation of three subentries are created under /cgroup2:

$ ls -al /cgroup2

total 4

dr-xr-xr-x 3 root root 0 Oct 22 10:22 .
drwxr-xr-x 24 root root 4096 Oct 21 08:19 ..
-r--r--r-- 1 root root 0 Oct 21 18:30 cgroup.controllers
-rw-r--r-- 1 root root 0 Oct 21 18:30 cgroup.procs
-rw-r--r-- 1 root root 0 Oct 21 18:39 cgroup.subtree_control
Running:

`cat /cgroup2/cgroup.controllers`

Will show the three enabled cgroup v2 controllers:

`io memory pids`

########################################################################

Running:

`cat /cgroup2/cgroup.subtree_control`

Will show nothing, as `cgroup.subtree_control` is always created empty (for the root object and for every newly created subgroup).

########################################################################

Running `cat /cgroup2/cgroup.procs` will show all processes in the system.
Steps – creating a subgroup and enabling a controller

Create a subgroup called “mySubGroup_1” (2_create_mySubgroup_1.sh)

```
$ mkdir /cgroup2/mygroup
```

Now, running: `cat /cgroup2/mygroup/cgroup.controllers` will show *nothing*, as no controller was enabled on the parent.

Also running `cat /cgroup2/mygroup/cgroup.subtree_control` will return nothing, as it is created empty by default.

Enable the `pids` controller of `mySubGroup_1` by setting it in its parent:

```
$ echo +pids > /cgroup2/cgroup.subtree_control
```

And now, running:

```
cat /cgroup2/mygroup/cgroup.controllers
```

```
pids
```

This shows that the pids controller is enabled in the immediate subgroup, `mySubGroup_1`. 
Also two interface files for the pids controller (\texttt{pids.current} and \texttt{pids.max}) were created for \texttt{mygroup}:

\texttt{ls -al /cgroup2/mygroup}

\texttt{total 0}

\texttt{drwxr-xr-x 2 root root 0 Oct 22 12:01 .}
\texttt{dr-xr-xr-x 3 root root 0 Oct 22 11:55 ..}
\texttt{-r--r--r-- 1 root root 0 Oct 22 11:55 cgroup.controllers}
\texttt{-r--r--r-- 1 root root 0 Oct 22 11:55 cgroup.events}
\texttt{-rw-r--r-- 1 root root 0 Oct 22 11:55 cgroup.procs}
\texttt{-rw-r--r-- 1 root root 0 Oct 22 11:55 cgroup.subtree_control}
\texttt{-r--r--r-- 1 root root 0 Oct 22 12:00 pids.current}
\texttt{-rw-r--r-- 1 root root 0 Oct 22 12:00 pids.max}
cat /cgroup2/mygroup/cgroup.events

Will show

*populated 0*

As there is not yet any process in the newly created subgroup, *mygroup*. 
Steps – match subgroup by path

Create the following iptables “match subgroup by path” rule

- The `xt_cgroup` kernel module should be loaded for enabling this.

```
iptables -A OUTPUT -m cgroup --path mygroup -j LOG
```

Attach the current shell process into `mySubgroup1`:

```
echo 0 > /cgroup2/mygroup/cgroup.procs
```

Start sending traffic from the shell while monitoring the syslog from another terminal:

```
ping -c 5 10.0.0.1
```
Steps – match subgroup by path (contd)

tail -f /var/log/syslog from another terminal shows:

Shows:

... 

Oct 22 13:22:38 rubyhost kernel: [ 523.788360] IN= OUT=eth0 SRC=143.185.141.66 DST=10.0.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=34714 DF PROTO=ICMP TYPE=8 CODE=0 ID=1323 SEQ=1

Oct 22 13:22:39 rubyhost kernel: [ 524.788141] IN= OUT=eth0 SRC=143.185.141.66 DST=10.0.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=34884 DF PROTO=ICMP TYPE=8 CODE=0 ID=1323 SEQ=2

Oct 22 13:22:40 rubyhost kernel: [ 525.788133] IN= OUT=eth0 SRC=143.185.141.66 DST=10.0.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=34952 DF PROTO=ICMP TYPE=8 CODE=0 ID=1323 SEQ=3

Oct 22 13:22:41 rubyhost kernel: [ 526.788134] IN= OUT=eth0 SRC=143.185.141.66 DST=10.0.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=34961 DF PROTO=ICMP TYPE=8 CODE=0 ID=1323 SEQ=4

Oct 22 13:22:42 rubyhost kernel: [ 527.788133] IN= OUT=eth0 SRC=143.185.141.66 DST=10.0.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=35171 DF PROTO=ICMP TYPE=8 CODE=0 ID=1323 SEQ=5

...

INTEL CONFIDENTIAL
Steps - No Internal Process rule

Create a subgroup, netsted1 for cgroup2/mySubGroup_1:

$ mkdir /cgroup2/mygroup/nested

Enable the pids controller on the nested subgroup just created by (5_enable_pid_on_nested.sh)

$ echo +pids > /cgroup2/mygroup/cgroup.subtree_control

Now attach some process (crond for example) to nested:

$ echo `pidof cron` > /cgroup2/mygroup/nested/cgroup.procs

And then try to attach the shell process to mySubGroup_1:

echo 0 > /cgroup2/mygroup/cgroup.procs

-bash: echo: write error: Device or resource busy

Since mygroup is an internal, non leave node, this operation fails.
Steps - cleanup

Trying to directly delete /cgroups2, on which the root object is mounted, will failed:

```
$ rmdir /cgroup2/
```

```
rmmdir: failed to remove '/cgroup2/': Device or resource busy
```

The right way is, after removing all subgroups, by:

```
$ umount /cgroup2
```

- *After the umount is done, you can delete /cgroup2 with rmdir /cgroup2*

Note: if there are any subgroups under /cgroup2, *umount /cgroup2* will fail.
Links

“Control Group Status Update” – a talk by Tejun Heo, Kernel Recipes 2016 (Paris):
https://www.youtube.com/watch?v=RLqXG4ArPe4&t=1732s

“Understanding the new control groups API”, article in lwn.net by Rami Rosen, March 23, 2016:
https://lwn.net/Articles/679786/

RDMA cgroups controller – Parav Pandit:
https://lwn.net/Articles/674161/

The RDMA cgroup will support both V1 and V2.

eBPF cgroup Patches by Daniel Mack.
"Network filtering for control groups"
https://lwn.net/Articles/698073
Thank You!
Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A “Mission Critical Application” is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL’S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES AND EXPENSES AND REASONABLE ATTORNEYS’ FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm%20Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel Performance Benchmark Limitations

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.


Intel® Active Management Technology requires the platform to have an Intel® AMT-enabled chipset, network hardware and software, as well as connection with a power source and a corporate network connection. With regard to notebooks, Intel AMT may not be available or certain capabilities may be limited over a host OS-based VPN or when connecting wirelessly, on battery power, sleeping, hibernating or powered off. For more information, see http://www.intel.com/technology/amt.

Hyper-Threading Technology requires a computer system with an Intel® Pentium® 4 Processor supporting HT Technology and an HT Technology-enabled chipset, BIOS, and operating system. Performance will vary depending on your hardware and operating system configuration. Consult with your system vendor for more information.

Intel® Virtualization Technology requires a computer system with an enabled Intel® Processor, BIOS, and operating system. Performance will vary depending on your hardware and software configurations. Consult with your system vendor for more information.

Software applications may not be compatible with all operating systems. Please check with your application vendor.

* Other vendors are listed by Intel as a convenience to Intel’s general customer base, but Intel does not make any representations or warranties whatsoever regarding quality, reliability, functionality, or compatibility of these devices. This list and/or these devices may be subject to change without notice.

Copyright © 2016, Intel Corporation. All rights reserved.
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.

```c
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:

```bash
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:

```bash
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

```bash
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:

```bash
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:

```bash
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.

<table>
<thead>
<tr>
<th>#subsys_name</th>
<th>hierarchy</th>
<th>num_cgroups</th>
<th>enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpuset</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>cpu</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>cpuacct</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>blkio</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>memory</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>devices</td>
<td>6</td>
<td>61</td>
<td>1</td>
</tr>
<tr>
<td>freezer</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>net_cls</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>perf_event</td>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>net_prio</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>hugetlb</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pids</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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

```bash
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
```
PIIDs 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
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.**