Understanding the new control groups API

March 23, 2016

This article was contributed by Rami Rosen

After many years, the Linux kernel's control group (cgroup) infrastructure is undergoing a rewrite that makes changes to the API in a number of places. Understanding the changes is important to developers, particularly those working with containerization projects. This article will look at the new features of cgroups v2, which were recently declared production-ready in kernel 4.5. It is based on a talk I gave at the recent Netdev 1.1 conference in Seville, Spain. The video [YouTube] for that talk is now available online.

Background

The cgroup subsystem and associated controllers handle management and accounting of various system resources like CPU, memory, I/O, and more. Together with the Linux namespace subsystem, which is a bit older (having started around 2002) and is considered a bit more mature (apart, perhaps, from user namespaces, which still raise discussions), these subsystems form the basis of Linux containers. Currently, most projects involving Linux containers, like Docker, LXC, OpenVZ, Kubernetes, and others, are based on both of them.

The development of the Linux cgroup subsystem started in 2006 at Google, led primarily by Rohit Seth and Paul Menage. Initially the project was called "Process Containers", but later on the name was changed to "Control Groups", to avoid confusion with Linux containers, and nowadays everybody calls them "cgroups" for short.

There are currently 12 cgroup controllers in cgroups v1; all—except one—have existed for several years. The new addition is the PIDs controller, developed by Aditya Kali and merged in kernel 4.3. It allows restricting the number of processes created inside a control group, and it can be used as an anti-fork-bomb solution. The PID space in Linux consists of, at a maximum, about four million PIDs (`PID_MAX_LIMIT`). Given today's RAM capacities, this limit could easily and quite quickly be exhausted by a fork bomb from within a single container. The PIDs controller is supported by both cgroups v1 and cgroups v2.

Over the years, there was a lot of criticism about the implementation of cgroups, which seems to present a number of inconsistencies and a lot of chaos. For example, when creating subgroups (cgroups within cgroups), several cgroup controllers propagate parameters to their immediate subgroups, while other controllers do not. Or, for a different example, some controllers use interface files (such as the cpuset controller's `clone_children`) that appear in all controllers even though they only affect one.

As maintainer Tejun Heo himself has admitted [YouTube], "design followed implementation", "different decisions were taken for different controllers", and "sometimes too much flexibility causes a hindrance". In an LWN article from 2012, it was said that "control groups are one of those features that kernel developers love to hate."

Migration

The cgroups v2 interface was declared non-experimental in kernel 4.5. However, the cgroups v1 subsystem was not removed from the kernel, so, after the system boots, both cgroups v1 and cgroups v2 are enabled by default. You can use a mixture of both of them, although you cannot use the same type of controller in both cgroups v1 and in cgroups v2 at the same time.

It is worth mentioning that there is a patch that adds a kernel command-line option for disabling cgroups v1 controllers (`cgroup_no_v1`), which was merged for kernel 4.6.

Kernel support for cgroups v1 will probably still exist for at least several more years, as long as there are user-space applications that use it—quite like what we had in the past with iptables and ipchains, and what we observe now with iptables and nftables. Some user-space applications have already started migration to cgroups v2—for example, systemd and CGManager.

Both versions of cgroups are controlled by way of a synthetic filesystem that gets mounted by the user. During the last three years or so, a special mount option was available in cgroups v1 (`__DEVEL__sane_behavior`). This mount option enabled using certain experimental features, some of

(__DEVEL__sane_behavior). This mount option enabled using certain experimental features, some of which formed the basis of cgroups v2 (the option was removed in kernel 4.5, however). For example, using this mount option forces the use the unified hierarchy mode, in which controller management is handled similarly to how it is done in cgroups v2. The `__DEVEL__sane_behavior` mount option is mutually exclusive with the mount options that were removed in cgroups v2, like `noprefix`, `clone_children`, `release_agent`, and more.

Systemd started to use cgroups for service management rather than for resource management many years ago. Each systemd service is mapped to a separate control group. However, the migration of systemd to cgroups v2 is still partial, as it uses the `__DEVEL__sane_behavior` mount option. Also, in CGManager, current support for cgroups v2 is partial: it is available only when using Upstart, and not when using systemd.

Currently, three cgroup controllers are available in cgroups v2: I/O, memory, and PIDs. There are already patches and discussions in the cgroups mailing list about adding the CPU controller as well.

There are also interesting patches adding support for resource groups, posted just last week by Heo. In cgroups v1, you could assign threads of the same process to different cgroups, but this is not possible in cgroups v2. As a result, in-process resource-management abilities, like the ability to control CPU cycle distribution hierarchically between the threads of a process, is missing, as all of the threads belong to a single cgroup. With the suggested resource groups (rgroups) infrastructure, this ability can be implemented as a natural extension of the `setpriority()` system call.

Details of the cgroups v2 interface

Mounting cgroups v2 is done as follows:

```
mount -t cgroup2 none $MOUNT_POINT
```

Note that the type argument (following `-t`) specified has changed; cgroups v1 used `-t cgroup`. As in cgroups v1, the mount point can be anywhere in the filesystem. But, in contrast, there are no mount options at all in cgroups v2. One could use mount options to enable controllers in cgroups v1, but in cgroups v2 this is done differently, as we will see below. Creation of new subgroups in cgroups v2 is done with `mkdir groupName`, and removal is done with `rmdir groupName`.

After mounting cgroups v2, a cgroup root object is created, with three cgroup core interface files beneath it. For example, if cgroups v2 is mounted on `/sys/fs/cgroup2`, the following files are created under that

directory:

- `cgroup.controllers` – This shows the supported cgroup controllers. All v2 controllers not bound to a v1 hierarchy are automatically bound to the v2 hierarchy, and show up in `cgroup.controllers` of the cgroup root object.
- `cgroup.procs` – When the the cgroup filesystem is first mounted, `cgroup.procs` in the root cgroup contains the list of PIDs of all processes in the system, excluding zombie processes. For each newly created subgroup, the `cgroup.procs` is empty, as no process is attached to the newly created group. Attaching a process to a subgroup is done by writing its PID into the subgroup's `cgroup.procs`.
- `cgroup.subtree_control` – This holds the controllers that are enabled for the immediate subgroups. This entry is empty just after mount, as no controllers are enabled by default. Enabling and disabling controllers in the immediate subgroups of a parent is done only by writing into its `cgroup.subtree_control` file. So, for example, enabling the memory controller is done by:

  ```
  echo "+memory" > /sys/fs/cgroup2/cgroup.subtree_control
  ```

  and disabling it is done by:

  ```
  echo "-memory" > /sys/fs/cgroup2/cgroup.subtree_control
  ```

  You can enable/disable more than one controller in the same command line.

These three cgroup core interface files are also created for each newly created subgroup. Apart from these three files, a cgroup core interface file called `cgroup.events` is created. This interface file is unique to non-root subgroups.

The `cgroup.events` file reflects the number of processes attached to the subgroup, and consists of one item, "`populated: value`". The value is `0` when there are no processes attached to that subgroup or its descendants, and `1` when there are one or more processes attached to that subgroup or its descendants.

As mentioned, subgroup creation is similar to how it is done in cgroups v1. But in cgroups v2, you can only create subgroups in a single hierarchy, under the cgroups v2 mount point. When a new subgroup is created, the value of the "populated" entry in `cgroup.events` is `0`, as you would expect, as there is no process yet attached to this newly created subgroup.

You can monitor events in this subgroup by calling `poll()`, `inotify()`, or `dnotify()` from user space. Thus, you can be notified notified when those files change, which can be used to determine when the last

process attached to a subgroup terminates or when the first process is attached to that subgroup. This mechanism is much more efficient in terms of performance than the parallel mechanism in cgroups v1, the release agent.

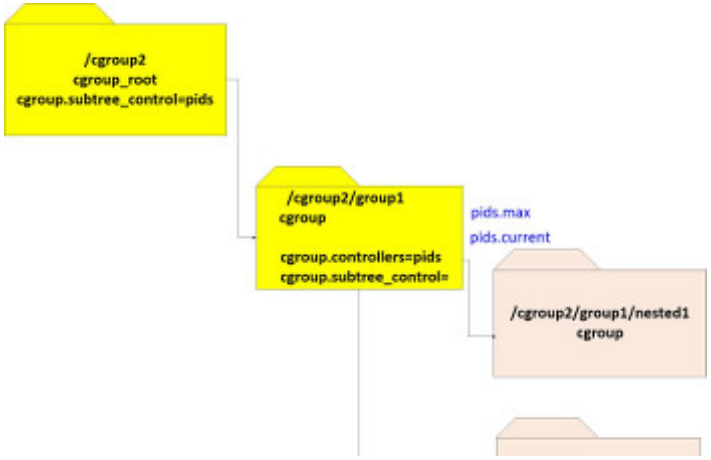
It is worth mentioning that this notification mechanism can also be used by controller-specific interface files. For example, the cgroups v2 memory controller has an interface file called `memory.events`, which enables monitoring memory events like out-of-memory (OOM) in a similar way.

When a new subgroup is created, controller-specific files are created for each enabled controller in this subgroup. For example, when the PIDs controller is enabled, two interface files are created: `pids.max` and `pids.current`, for setting a limit on the number of processes forked in that subgroup, and for accounting of the number of processes in that subgroup.

Let's take a look at two diagrams illustrating what we just described. The following sequence mounts cgroups v2 on `/cgroup2` and creates a subgroup called "group1", creates two subgroups of group1 ("nested1" and "nested2"), then enables the PIDs controller in group1:

```
mount -t cgroup2 nodev /cgroup2
mkdir /cgroup2/group1
mkdir /cgroup2/group1/nested1
mkdir /cgroup2/group1/nested2
echo +pids > /cgroup2/cgroup.subtree_control
```

The following diagram illustrates the status after running this sequence. We can see that the two PIDs controller interface files, `pids.max` and `pids.current`, were created for group1.

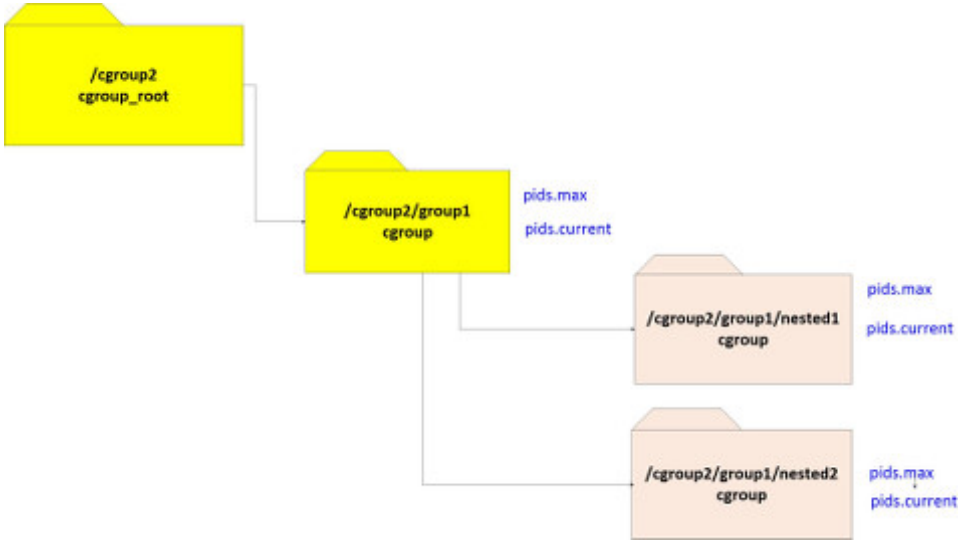


Now, if we run:

```
echo +pids > /cgroup2/group1/cgroup.subtree_control
```

this will enable the PIDs controller in group1's immediate subgroups, nested1 and nested2. By writing `+pids` into the `subtree_control` of the root cgroup, we only enable the PIDs controller in the root's *direct* child subgroups and no other descendants. As a result, the PIDs-controller–specific files (`pids.max` and `pids.current`) are created for both these newly-created subgroups.

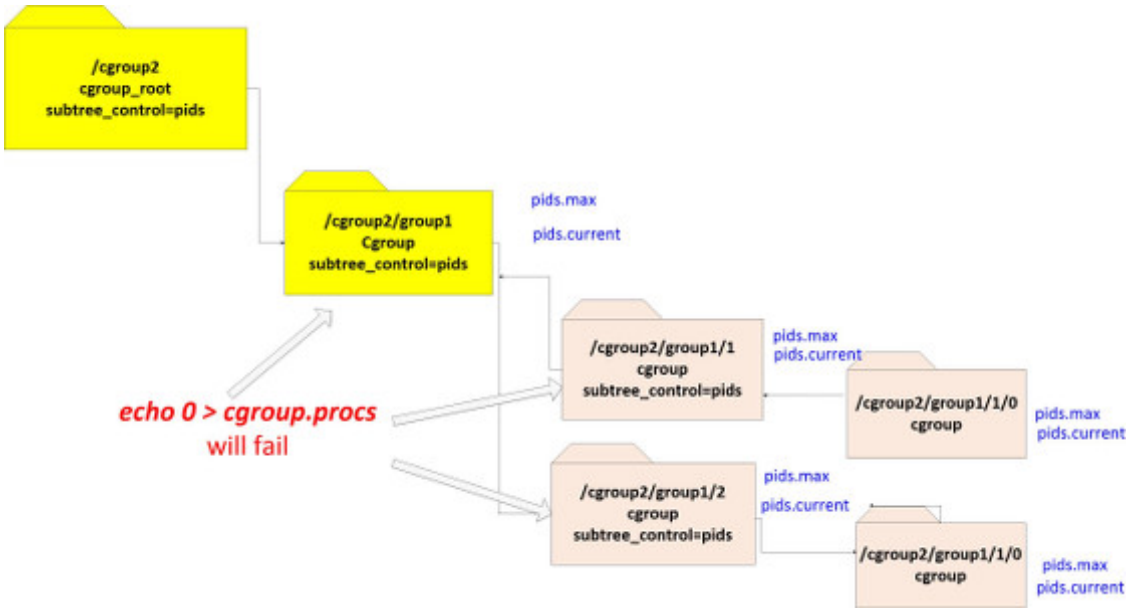
The subsequent diagram shows the status after enabling the PIDs controller on group1.



The no-internal-process rule

Unlike in cgroups v1, in cgroups v2 you can attach processes *only* to leaves. This means that you cannot attach a process to an internal subgroup if it has any controller enabled. The reason behind this rule is that processes in a given subgroup competing for resources with threads attached to its parent group create significant implementation difficulties.

The following diagram illustrates this.



(Note: when you write `0` into `cgroup.procs`, this will write the PID of the process performing the writing into the file.)

The documentation discusses the no-internal-process rule in more detail.

In cgroups v1, a process can belong to many subgroups, if those subgroups are in different hierarchies with different controllers attached. But, because belonging to more than one subgroup made it difficult to disambiguate subgroup membership, in cgroups v2, a process can belong only to a single subgroup.

We will look at an example when this restriction is important. In cgroups v1, there are two network controllers: `net_prio` (written by Neil Horman) and `net_cls` (by Thomas Graf). These controllers were not extended to support cgroups v2. Instead, the `xt_cgroup` netfilter matching module was extended to support matching by a cgroup path. For example, the following iptables rule matches traffic that was generated by a socket created in a process attached to *mygroup* (or its descendants):

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

Such a match is not possible in cgroups v1, because sometimes a process can belong to more than a single

subgroup. In cgroups v2, this problem does not exist, because of the single-subgroup rule.

Summary

Work is ongoing; in addition to the resource-group patches mentioned earlier, there are patches for a new RDMA cgroup controller that are currently in the pipeline. This patch set allows resource accounting and limit enforcement on a per-cgroup, per-RDMA-device basis. These patches are in the post-RFC phase, and are in the ninth iteration as of this writing; it seems likely that they are to be merged soon.

As we have seen, the new interface of cgroups v2, which was recently declared stable in the kernel, has several advantages over cgroups v1, such as its notification-to-user-space mechanism. Although the cgroups v2 implementation is still in its initial stages, it seems to be much better organized and more consistent than cgroups v1.

(Log in to post comments)

Understanding the new control groups API

Posted Mar 24, 2016 20:41 UTC (Thu) by **flussence** (subscriber, #85566) [Link]

That iptables example looks handy - sounds like I'll be able to firewall specific apps/daemons by cgrouping them appropriately. I know it was already possible to achieve the same end with net namespaces and forwarding, but that method feels disproportionately complicated.

Reply to this comment

no more cgroups without controllers, it seems

Posted Mar 31, 2016 9:41 UTC (Thu) by **wingo** (guest, #26929) [Link]

AFAIU with cgroups v2 it is no longer possible to create cgroups without associated controllers, just for organizational purposes. Is that correct? So a program can't create a cgroup tree just for its own uses; Linux seems to be doubling down on centralized administration of the cgroup tree. I guess that's fine but it's a notable difference relative to cgroups v1.

Reply to this comment

LWN .net

no more cgroups without controllers, it seems

Posted Mar 31, 2016 19:34 UTC (Thu) by **RamiRosen** (guest, #37330) [Link]

As a matter of fact, it is possible to create a subgroup in cgroup V2 without any associated controller. Moreover, by default, in cgroup V2, every newly created subgroup is created empty, with no controllers associated to it, except the root cgroup, to which all controllers are associated, as mentioned in the article. And nothing enforces you to attach controllers later to that newly created cgroup V2 subgroup. In cgroup V1, you could create a *root* cgroup without associating any controller to it, and this is indeed as opposed to cgroup v2, and probably this what the reader meant, and he is right about this and about its implications.
- Rami Rosen

Reply to this comment

Understanding the new control groups API

Posted Apr 20, 2016 7:28 UTC (Wed) by **bgoglin** (subscriber, #7800) [Link]

I booted a 4.6-rc2 kernel with cgroup_no_v1=all but I only see "io" and "memory" in cgroup.controllers. Are all controllers ported to the v2 API already? (I want to look at the cpuset controller)

Reply to this comment

Understanding the new control groups API

Posted Apr 21, 2016 6:58 UTC (Thu) by **RamiRosen** (guest, #37330) [Link]

No, not all cgroup V1 controllers were ported to cgroup V2.

The article says:
"Currently, three cgroup controllers are available in cgroups v2: I/O, memory, and PIDs. There are already patches and discussions in the cgroups mailing list about adding the CPU controller as well".

The reason that you do not see the PIDS cgroup V2 controller is probably because a cgroup V1 controller is mounted by systemd (or some other app), and you cannot mount the same type of controller in both cgroup V1 and cgroup V2. Another reason can be that

in both cgroup V1 and cgroup V2. Another reason can be that
your kernel is built without support for PIDS controller (CONFIG_CGROUP_PIDS is not set).

Some of the cgroup V1 controllers, like the two network controllers, will probably not be ported to cgroup V2. Work was done in supporting iptable rules, which can filter according to a cgroup associated with a socket, as was shown in the article.

Rami Rosen

Reply to this comment

Understanding the new control groups API

Posted Oct 15, 2016 20:14 UTC (Sat) by **gdamjan** (subscriber, #33634) [Link]

is there a tc match for the unified cgroup2 yet?

Reply to this comment

Understanding the new control groups API

Posted Nov 22, 2018 21:29 UTC (Thu) by **arekm** (subscriber, #4846) [Link]

PID controller is weird. Example:

```
# pwd
/sys/fs/cgroup/somegroup
# grep "" pids.*
pids.current:110
pids.events:max 0
pids.max:60
```

You can have much more processes in cgroup than pids.max says. And that's done with migrating processes to somegroup cgroup. With migration limits don't apply.

Migrations are the only way to assign cgroup to, for example, httpd subprocesses.

I think that should be considered a bug.

Reply to this comment