Linux Control Group v2 – An Introduction

Waiman Long
Principal Software Engineer, Red Hat Inc
Devconf.cz 2019
What is cgroup?

- In the Linux kernel, “cgroup” stands for “control group” which is a collection of processes that are bound by the same criteria and associated with a set of parameters or limits.
- It is a kernel mechanism for organizing process hierarchically and distributing system resources along the hierarchy in a controlled and configurable manner.
- Together with namespace, they are the two key pieces of kernel technology that enable container support.
- Many kernel resources can be controlled by cgroup, such as
  - CPU
  - Memory
  - Block I/O
  - Devices
  - Network
  - Process IDs
How to use cgroup?

- From the kernel's perspective, the process hierarchy is presented as a directory tree in a cgroup virtual filesystem.
- Each directory (cgroup) in the tree has a set of cgroup control files that can be read and/or written into.
- Every process running in a system is associated with one of the directories in the hierarchy. Default is root, the top level directory in a hierarchy.
- The cgroup core manages control files with the “cgroup.” prefix as well as some other files that don’t have a specific prefix, while each of the controllers manage control files that start with their respective controller name prefix.
- Below are the files available under the /sys/fs/cgroup/cpuset directory (cgroup v1)

<table>
<thead>
<tr>
<th>cgroup.clone_children</th>
<th>cpuset.effective_mems</th>
<th>cpuset.memory_spread_slab</th>
</tr>
</thead>
<tbody>
<tr>
<td>cgroup.event_control</td>
<td>cpuset.mem_exclusive</td>
<td>cpuset.mems</td>
</tr>
<tr>
<td>cgroup.procs</td>
<td>cpuset.mem_hardwall</td>
<td>cpuset.sched_load_balance</td>
</tr>
<tr>
<td>cgroup.sane_behavior</td>
<td>cpuset.memory_migrate</td>
<td>cpuset.sched_relax_domain_level</td>
</tr>
<tr>
<td>cpuset.cpu_exclusive</td>
<td>cpuset.memory_pressure</td>
<td>notify_on_release</td>
</tr>
<tr>
<td>cpuset.cpus</td>
<td>cpuset.memory_pressure_enabled</td>
<td>release_agent</td>
</tr>
<tr>
<td>cpuset.effective_cpus</td>
<td>cpuset.memory_spread_page</td>
<td>tasks</td>
</tr>
</tbody>
</table>
Using cgroup

- Users can manually access the control files in the cgroup virtual file system. They can create sub-directories to form new cgroups and move processes around.

- More commonly, tools like cgcreate, cgexec and cgclassify (from libcgroup) are used to manage cgroups.

- In most cases, high level software like Docker, LXC, libvirt, systemd, and so on manages cgroups according to the policy choice selected by the users and hide the low level details from the users.
Cgroup Internals

- Cgroup is largely composed of 2 parts – the core and controllers.
- The cgroup core is responsible for hierarchically organizing processes.
- A cgroup controller is primarily responsible for distributing a specific type of system resource along the hierarchy. There are also some utility controllers which serve purpose other than resource distribution such as accounting and control.
- The currently available controllers include:
  - blkio
  - cpu,cpuacct
  - cpuset
  - devices
  - freezer
  - hugetlb
  - memory
  - net_cls,net_prio
  - perf_event
  - pids
  - rdma
Cgroup Controllers & Hierarchy

• For the original cgroup controllers (v1), each of which can have its own process hierarchy. Or they can team up with other related controllers into the same hierarchy. Different distros may make different choices of how those controllers are being combined, if at all.

• The flexibility of combining different v1 controllers is probably the biggest single problem it has due to the tyranny of choice.

• There are cases where one controller may want to coordinate with another one in managing their respective resources. With the flexibility in hierarchy, there is no guarantee that the two controllers may meaningfully coordinate with each other as they may be in completely different process hierarchy.

• This is where cgroup v2 comes in. Cgroup v2 provides a single unified process hierarchy for all the controllers that it owns. This allows coordination to happen among different controllers. The new writeback controller is one such example as it requires coordination between the memory and the block I/O controllers. The writeback controller is v2 only.

• The process hierarchy managed by v1 controllers are internally called legacy hierarchy, while the hierarchy used by v2 controllers are called default or unified hierarchy.
Using Unified Hierarchy

- Not all the cgroup v1 controllers are currently supported in the unified hierarchy. Some of them are still being worked on while others may never be supported in v2.

- A controller can be used either in a v1 legacy hierarchy or a v2 unified hierarchy, but not both. So one have to choose which hierarchy should each of the controllers go to.

- Any v2-usable controllers that are not bound to a legacy hierarchy will be bound to unified hierarchy.

- The “cgroup_no_v1=” boot command line option can also be used to force no v1 binding for the specified controllers and force them to be bound to v2.

- Cgroup v2 also supports the concept of delegation where a less privileged user is allowed to manage cgroups in a limited way. Cgroup v1 has no such support.
Using Controllers in Unified Hierarchy

- Unlike the legacy hierarchy, a controller in the unified hierarchy is not enabled by default except at the top level cgroup (root). A new “cgroup.subtree_control” file controls which controllers should be enabled in the child cgroups.

- To enable the “cpu” controller, for example, in the child cgroups, use
  # echo “+cpu” > cgroup.subtree_control
  To disable that controller, use
  # echo “-cpu” > cgroup_subtree_control

- If a controller is not enabled at a cgroup, the controller setting at the nearest ancestor cgroup where that controller is enabled will be applied to the processes in that ancestor cgroup as well as all the processes in its descendant cgroups.

  Root(cpu) -- A(cpu) -- B -- C => Root|D(cpu) -- A|B|C(cpu)
  +- D

- From a controller’s perspective, all the cgroups that don’t have that controller enabled are essentially merged to their nearest ancestors with the enabled controller.
Cgroup v2 Threaded Mode

- In the legacy hierarchy, process management is done at the thread level. In the unified hierarchy, however, process management was done at the process level. So all the threads in a process should be in the same v2 cgroup. The cgroup v2 control file “cgroup.procs” shows all the processes that belong to this particular cgroup.

- That is fine for most of the controllers except the cpu controller which requires thread level management.

- There were differences in opinion on how to enable thread support for v2 cpu controller leading to the stalling of development for nearly one and a half year.

- Finally, a consensus emerged on adding a new threaded mode support in cgroup v2. Only selected controllers are designated to be allowed in cgroups that have threaded mode turned on. They are called threaded controllers. The other controllers that are not threaded are called domain controllers.
Using cgroup v2 Threaded Mode

- The cgroups with threaded mode on are called threaded cgroups, while the other cgroups are call domain cgroups.

- Threaded mode is enabled by issuing the following action:
  
  ```bash
  # echo threaded > cgroup.type
  ```

- Once threaded mode is on, thread level management can be done by moving threads by echoing their thread IDs to the “cgroup.threads” control file. With threaded mode off, moving a thread to “cgroup.threads” will move all the threads in the associated process over as well.

- The domain cgroup that is parent to child threaded cgroups is called threaded domain. Not all the child cgroups underneath a threaded domain need to be threaded.

- A threaded child cgroup and all its descendants is called a threaded subtree. All the cgroups in a threaded subtree must be threaded. Any non-threaded cgroup in that subtree will be in an invalid state and cannot not be used.
**v1 & v2 Controller Behavior Differences**

- For the same controller, the behavior and the naming of the control files provide by that controller may differ between v1 and v2.

- The v1 controllers were developed independently by different developers at different periods of time. As a result, the naming and usage conventions of one v1 controller may differ quite a lot from another v1 controller.

- Some v1 controllers have numerous features added over time whose usefulness may be dubious at best. Some of the provided features may not be the most elegant ways to achieve the desired effect.

- One of the design goals of the v2 controller interfaces is to make the control file naming and usage more consistent among the different controllers.

- Another goal is to trim out features that are not essential to the core functionality of the controller.

- As a result, a controller will expose less control files with slightly different names in the unified hierarchy than in the legacy hierarchy.
Migrating from cgroup v1 to v2

- Cgroup v1 is still being used predominantly in many Linux systems. However, most new cgroup development activities focuses on cgroup v2.

- To use those new features provided by cgroup v2, people has to migrate their systems to use cgroup v2.

- Not all the cgroup v1 controllers are currently supported in the unified hierarchy yet. Some of them are still being worked on while others may never be supported in v2.

- The new v2 controllers that are added recently or in development are:
  - cpuset (in v5.0)
  - freezer (under development, likely in 5.1)

- It has not been decided if the hugetlb controller should move to v2 as well.

- The devices, net_cls and net_prio controllers will not be supported in v2. Instead, they should be managed by using ebpf programs attached to each of the cgroups that want them.
Issues with Unified Hierarchy

• The unified hierarchy certainly solves some of the problems associated with the legacy hierarchy, but it also has its own drawbacks.

• The primary drawback is the proliferation of cgroups to accommodate all the different combinations of controller settings for each of v2 controllers that are needed by the users.

• Of all the cgroup controllers, the cpu controller has performance implication with nesting. The deeper the level of nesting, the more performance degradation will be observed. So if the cpu controller needs to be enabled in a cgroup way down in the unified hierarchy, all its ancestor cgroups must have cpu controller enabled as well causing performance problem.

• Developers are working upstream to try to reduce the performance degradation associated with the cpu controller (sched group), but how much improvement can be made is still not clear.

• For the cpu controller, there are 2 main reasons for the slowdown:
  - More management overhead on the scheduler with deeper level of nesting.
  - More cacheline contention with managing scheduler statistical counts.
Cgroup v2 Bypass Mode

- The cgroup v2 bypass mode is a proposed patchset upstream that can help to alleviate the performance impact of using the cpu controller.

- With the bypass mode patchset applied, the state of a controller becomes tri-state. A bypass mode will be added in addition to “on” and “off”.

- With bypass mode on, a controller is not enabled in that cgroup but it allows a child cgroup to have that controller turned on or in bypass mode again. In this way, one can turn on the cpu controller only when it is needed reducing the level of nesting.
The cgroup v2 Core

- `cgroup.controllers` – a read-only file listing the enabled controllers.
- `cgroup.events` – a read-only file indicating if there are live processes residing in this cgroup and all its descendants (1 or 0, non-root only).
- `cgroup.max.depth` – a read-write file controlling the maximum descent depth below the current cgroup. Default is “max” which means no limit.
- `cgroup.max.descendants` – a read-write file controlling the maximum number of descendant cgroups. Default is “max”.
- `cgroup.procs` – a read-write file listing the PIDs of all processes that belong to this cgroup.
- `cgroup.stat` – a read-only file showing the number of visible and dying descendant cgroups.
- `cgroup.subtree_control` – a read-write file listing what controllers are enabled in child cgroups.
- `cgroup.threads` – a read-write file listing the TIDs of all threads that belong to this cgroup.
- `cgroup.type` – a read-write file for thread mode management.
The v2 CPU Controller

- The cpu controller manage how much cpu time should be allocated to processes in the cgroup.

- The cgroup v1 cpuacct controller is integrated into the v2 cpu controller.

- `cpu.max` – read-write 2-value file listing the maximum bandwidth limit and the period in us. It is for CFS bandwidth control, The default is “max 100000” which means no bandwidth limit with a period of 100ms.

- `cpu.stat` – read-only file that exists in all non-root cgroups no matter if cpu controller is enabled or not. It always lists the following three stats – `usage_usec`, `user_usec` and `system_usec`. The following three stats are shown only in cgroups with cpu controller enabled – `nr_periods`, `nr_throttled` and `throttled_usec`.

- `cpu.weight` – read-write file on non-root cgroups. The default is 100 within a range of 1-10,000. The default of 100 is equivalent to 1024 in the v1 cpu.shares file.

- `cpu.weight.nice` – read-write file on non-root cgroups. The default is 0 within a range of -20 – 19. It is an alternative way of setting weight using the nice value scale.

- `cpu.pressure` – a read-only file on non-root cgroups. It shows pressure stall information for the CPU.
The v2 Cpuset Controller

- The cpuset controller manages the affinity of processes to CPUs as well as memory nodes. On NUMA machines, the number of memory nodes is usually the same as the number of processor sockets, though memory-only nodes without CPU is also possible.

- cpuset.cpus – read-write file on non-root cgroups showing the list of CPUs being requested to be used by processes within this cgroup.

- cpuset.cpus.effective – read-only file on all cgroups showing the list of CPUs actually allocated to this cgroup.

- cpuset.mems – read-write on no-root cgroups showing the list of memory nodes being requested to be used by processes within this cgroup.

- cpuset.mems.effective – read-only file on non-root cgroups showing the list of memory nodes actually allocated to this cgroup.

- cpuset.cpus.partition – read-write file on non-root cgroups managing CPU partitions. A CPU partition is a separate scheduling domain with a set of exclusive CPUs not shared with other other scheduling domains. It is used quite often in managing realtime processes.
The v2 Memory Controller

- The memory controller regulates distribution of memory. Currently the following memory usages are being tracked:
  - Userland memory – page cache and anonymous memory
  - Kernel data structures such as dentries and inodes.
  - TCP socket buffers
- memory.current – read-only non-root file listing total amount of memory consumed by the cgroup and its descendants.
- memory.min – read-write non-root file showing the hard memory protection limit (default 0). Memory reclaim and OOM kill won’t happen if memory consumption is below the limit.
- memory.low – read-write non-root file showing the soft memory protection limit (default 0). Memory reclaim and OOM kill shouldn’t happen if memory consumption is below the limit.
- memory.high – read-write non-root file showing the memory usage throttle limit (default max). Memory will be reclaimed if going over the limit.
The v2 Memory Controller (Cont.)

- memory.max – read-write non-root file showing the memory usage hard limit (default max). Memory will be reclaimed if going over the limit. If reclaim fails, OOM will be invoked.
- memory.oom.group – read-write boolean non-root file (default 0) determining if the entire cgroup should be treated as an indivisible workload by the OOM killer.
- memory.events – read-only non-root file listing the number of memory events that happen. It generates a file modified event whenever its values change.
- memory.stat – read-only non-root file listing memory consumption (in bytes) in different areas by processes in the cgroup.
- memory.swap.current – read-only non-root file listing swap space consumption by the cgroup and its descendants.
- memory.swap.max – read-write non-root file listing swap usage hard limit (default max).
- memory.swap.events – read-only non-root file listing number of swap related events.
- memory.pressure – read-only non-root file showing pressure stall information.
The v2 IO Controller

- The IO controller regulates the distribution of IO resources.
- io.stat – read-only non-root file showing the IO usage information.
- io.weight – read-write non-root file showing relative amount of IO time the cgroup can use relative to its siblings. The default is 100 within the range of 1-10000.
- io.max – read-write non-root file showing BPS and IOPS based IO limit (read and write).
- io.pressure – read-only non root showing IO pressure stall information.
- There are current two sub-controllers underneath the IO controller umbrella:
  - Writeback – The IO controller, in conjunction with memory controller, implements control of page cache writeback Ios. Filesystem support is required and currently only ext2, ext4 and btrfs are supported.
  - IO Latency – A latency target is specified for throttling peers (siblings) when the limit is exceeded.
Cgroup Namespace

- The cgroup namespace provides a mechanism to virtualize the view of the "/proc/$PID/cgroup" file and cgroup mounts.

- Cgroup v2 supports the nsdelegate mount option which consider cgroup namespaces as delegation boundaries. This mount option is system-wide and should be used in the init namespace only.
Support of Cgroup v2 in RHEL

- Cgroup v2 is not going to be supported in any of the RHEL7 release streams as their inclusion will likely break kABI.

- As cpuset v2 controller is too late for RHEL8.0, cgroup v2 support will only be in technical preview mode.

- Both the cpuset and freezer v2 controllers are planned to be included in the RHEL8.1 release. We are hoping to formally support cgroup v2 starting from RHEL8.1 pending proper qualification of our middleware layer with respect to cgroup v2 support.
Looking Forward

- We are going to see cgroup v2 adoption rate increases gradually over time.
- At the same time, cgroup v1 will stay on for a long period of time as v1 and v2 are vastly different in certain aspects.
- Feature creep into cgroup v2 will happen, but it will be highly controlled and rationalized before new features can go in.
- Most distros are going to support both cgroup v1 and v2 in some way for a rather long period of time.
Thank You!