A brief survey on control groups version 2

Giovanni Lupi

Politecnico di Milano
Milan, Italy
giovanniedoardo.lupi@mail.polimi.it

1 Introduction

Control groups, usually referred to as cgroups, are a Linux kernel feature which allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored. These hierarchical groups can be configured to show a specialized behavior that helps with tuning the system to make best use of its available resources. Examples of these resources are CPU time, system memory, network bandwidth, or combinations thereof.

This document presents a brief summary of the main characteristics of cgroup version 2, while also emphasizing some of the main differences with regard to cgroup version 1.

2 History

Cgroup version 1 was originally developed in 2006-2007 by engineers at Google. At the beginning of 2008, it was integrated into the Linux kernel (Linux 2.6.24). Initially, only three controllers were available, but several other ones have been progressively added through the years. However, the development of these controllers was largely uncoordinated, with the result that many inconsistencies arose between controllers and the management of the cgroup hierarchies became rather complex.

The issues with the initial implementation motivated the development of a second version of cgroup, starting from 2012. After a lengthy development period, the new version was made official in 2016 with Linux 4.5. Several of the controllers available in version 1 were still lacking in version 2, however their implementation was added in the following years. As of now, majority of the major Linux distributions moved to using version 2 by default.

3 Cgroups v2 fundamentals

The cgroup mechanism is mostly comprised of two parts: the cgroup core and the controllers. The core is primarily responsible for hierarchically organizing and grouping processes. The controllers are kernel components responsible for managing, controlling or monitoring processes in cgroups.

The cgroup hierarchy is manifest as a mounted pseudo filesystem, which effectively represents its interface. Therefore, cgroup manipulation takes form of filesystem operations, which can fundamentally be performed in four ways: via shell commands, programmatically, via a management daemon (like *systemd*),

or via container framework tools (like LXC or Docker). The filesystem is typically mounted under `/sys/fs/cgroup`.

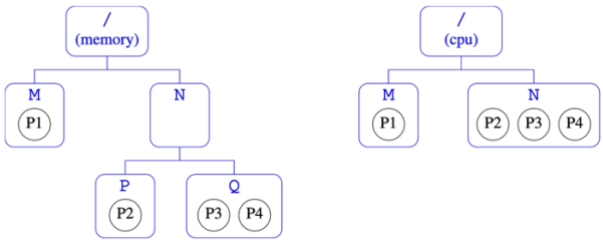
Resource controllers generally perform some resource limitation, monitoring or other type of management of groups of processes (for example the CPU controller limits the CPU consumption). Following certain structural constraints, controllers may be enabled or disabled selectively on a cgroup by defining specific attributes in the hierarchy. Controllers' behaviors are typically hierarchical, as their effect is propagated throughout the subhierarchy underneath the cgroup where the attributes are defined. In various instances, controllers are equivalently named subsystems. However, this term is arguably quite generic, so the (resource) controller denomination is more appropriate.

4 Overcoming cgroup v1 limitations

This section focuses on some of the main problems of the initial cgroup development and how the cgroup version 2 implementation is mostly capable of overcoming them.

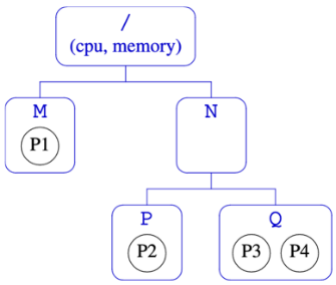
4.1 Multiple hierarchies

To provide a high degree of flexibility, cgroup v1 allowed an arbitrary number of hierarchies and each hierarchy could host any number of controllers. However, this flexibility turned out to be less useful in practice than originally planned, while simultaneously causing a series of issues. One of the most popular approaches was that of dedicating to each controller its own hierarchy.



The main advantage of this solution is that it allows each controller to manage processes at different levels of granularity. However, when moving processes across groups (for example moving P2 to group M), operations must be repeated in each hierarchy.

The opposite approach is to attach multiple controllers to the same hierarchy.



While this solution removes the need to replicate move operations on multiple hierarchies, it forces the controllers belonging to the same hierarchy to manage processes at the same level of granularity.

Another problem is that, since there can only be one instance of each controller, utility type controllers such as freezer, which can be useful in all hierarchies, could only be used in one. This issue was exacerbated by the impossibility to move controllers to another hierarchy once hierarchies were populated.

Allowing to have completely orthogonal hierarchies brings another series of issues besides the ones aforementioned. However, such a requirement ended up being not needed in practice. What's usually called for instead is the ability to have differing levels of granularity depending on the specific controller.

Because of these reasons, cgroups version 2 uses a single hierarchy for all controllers, while simultaneously providing a mechanism to allow per-controller granularity in the hierarchy.

4.2 Thread granularity

Cgroup v1 allowed thread granularity for cgroup membership, which means that threads of a single process could belong to different cgroups. However, such an organization could cause problems for some controllers. One notable example is the memory controller: since threads belonging to the same process already share an address space, it doesn't make sense to subject these threads to different memory controls.

Because of this, the ability to independently manipulate the cgroup memberships of the threads in a process was removed in the initial cgroups v2 implementation, where all the threads of a specific process were forced to be in the same cgroup. Thread level granularity was subsequently restored in a more limited form in Linux 4.14.

4.3 Groups vs tasks

Cgroup v1 allowed a cgroup to contain both tasks and child cgroups. This caused tasks belonging to a parent cgroup to compete with its children cgroups for resources. This caused several issues, as two different types of entities could compete and no standard solution was available to solve the problem. In fact, different controllers implemented different solutions. Furthermore, all these approaches were severely flawed and often displayed widely different behaviors, making cgroup as a whole highly inconsistent.

In cgroup v2, only leaf cgroups can contain processes.

4.4 General inconsistencies

Cgroup v1 grew without oversight and developed a large number of idiosyncrasies and inconsistencies between controllers. Different controllers were often designed by different groups of people, with no cohesion between the various interfaces.

Cgroup v2 establishes common conventions where appropriate and updates controllers so that they expose minimal and consistent interfaces.

4.5 Release notification

Release notification is a feature supported by cgroup v1 that provides the ability to get notified when the last process in a cgroup terminates. This mechanism was implemented in a rather expensive way. When a cgroup became empty, the kernel forked and executed a userland process that would respond to that specific event. Clearly, creating a process for each notification represents a poorly scalable implementation.

Cgroup v2 has a lightweight solution to deal with the problem.

5 Configuration

Recent *systemd* automatically mounts cgroup v2 at `/sys/fs/cgroup`. Otherwise, it is possible to manually mount the v2 hierarchy through the following command:

```
mount -t cgroup2 none /path/to/mount
```

In version 2 all controllers are automatically available under a single hierarchy, therefore it's not necessary to explicitly bind controllers to the mount point. However, a controller is available in v2 only if it isn't already bound to a v1 hierarchy. The restriction here is that a controller can't be simultaneously employed in both a cgroup v1 hierarchy and in the cgroup v2 hierarchy. This allows mixing the v2 hierarchy with the legacy v1 multiple hierarchies system in a fully backward compatible way.

Each v2 cgroup has a (read-only) *cgroup.controllers* file, which lists available controllers this cgroup can enable. It's possible that the system management software might automount the v1 cgroup filesystem,

so that all controllers are bound to the v1 hierarchy by default, causing the *cgroup.controllers* file in the v2 hierarchy to be empty. To fix this issue, it's possible to reboot the system using the kernel parameter `cgroup_no_v1=all`. This disables all v1 controllers.

6 Processes configuration

Cgroups form a tree structure and every process in the system belongs to one and only one cgroup. Initially, only the root cgroup exists to which all processes belong. A child cgroup can be created by creating a sub-directory:

```
mkdir $CGROUP_NAME
```

A given cgroup may have multiple child cgroups, which is what gives to the overall hierarchy a tree structure.

Each cgroup has a read-writable interface file *cgroup.procs*. When read, it lists the PIDs of all processes which belong to the cgroup, one-per-line. A process can be migrated into a cgroup by writing its PID to the target cgroup's *cgroup.procs* file. In general, if a process is composed of multiple threads, writing the PID of any thread migrates all threads of the process. When a process forks a child process, the new process is born into the cgroup that the forking process belongs to at the time of the operation.

After exit, a process stays associated with the cgroup that it belonged to at the time of exit until it's reaped; however, a zombie process does not appear in *cgroup.procs* and thus can't be moved to another cgroup.

A cgroup which doesn't have any children or live processes can be destroyed by removing the directory. Note that a cgroup which doesn't have any children and is associated only with zombie processes is considered empty and can be removed:

```
rmdir $CGROUP_NAME
```

7 Controlling resource controllers

A cgroup controller is usually responsible for distributing a specific type of system resource along the hierarchy, although there are utility controllers which serve purposes other than resource distribution. In cgroup v2, all mounted controllers reside in a single unified hierarchy.

Each cgroup has a *cgroup.controllers* file which lists all controllers available for the cgroup to enable. No controller is enabled by default.

Each cgroup also contains a *cgroup.subtree_control* file. This contains a list of controllers that are active (enabled) in the cgroup. By writing to this file, it is possible to enable or disable specific controllers. This can be achieved by writing to the file strings containing space-delimited controller names, each preceded by '+' (to enable a controller) or '-' (to disable a controller), as in the following example:

```
echo "+cpu +memory -io" > cgroup.subtree_control
```

When multiple operations are specified as above, either they all succeed or fail. If multiple operations on the same controller are specified, the last one is effective. The set of controllers available in this file is a subset of the set in the *cgroup.controllers* file of that cgroup. This means that only controllers which are listed in *cgroup.controllers* can effectively be enabled. An attempt to enable a controller that is not present in *cgroup.controllers* leads to an `ENOENT` error when writing to the *cgroup.subtree_control* file.

7.1 Constraint distribution

Enabling a controller in a cgroup indicates that the distribution of the target resource across its immediate children will be controlled. In other terms, a cgroup's *cgroup.subtree_control* file determines the set of controllers that are exercised in the child cgroups.

Let's consider this simple hierarchy example:

```
A(cpu,memory) - B(memory) - C()
                          \ D()
```

As A has *cpu* and *memory* enabled, A will control the distribution of CPU cycles and memory to its children, in this case, B. As B has *memory* enabled but not *cpu*, C and D will compete freely on CPU cycles but their division of memory available to B will be controlled.

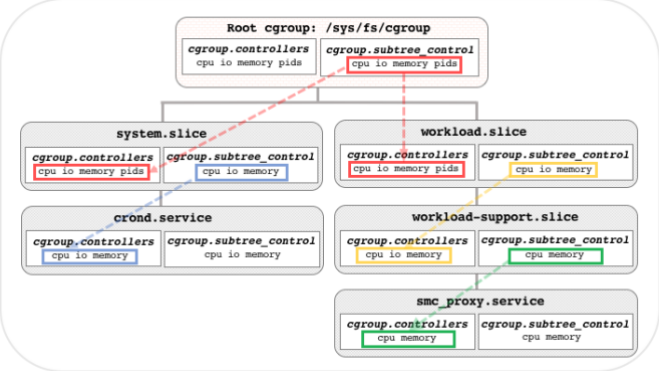
When a controller is present in the *cgroup.subtree_control* file of a parent cgroup, then the corresponding controller-interface files are automatically created in the children of that cgroup and can be used to exert resource control in the child cgroups. In the above example, enabling *cpu* on B would create the "*cpu.*" prefixed controller interface files in C and D (for example, *cpu.max* or *cpu.weight*.) Likewise, disabling *memory* from B would remove the "*memory.*" prefixed controller interface files from C and D.

This means that the controller interface files are owned by the parent rather than the cgroup itself. The parent cgroup uses these files to manage resource allocation in the child cgroups.

Since the root cgroup doesn't have any parent, it also doesn't contain any controller interface files. Therefore, control cannot be exercised on processes in the root cgroup.

7.2 Top-down constraint

Resources are distributed top-down and a cgroup can further distribute a resource only if the resource has been distributed to it from the parent. This means that all non-root *cgroup.subtree_control* files can only contain controllers which are enabled in the parent's *cgroup.subtree_control* file. In other terms, the contents of the *cgroup.controllers* file match the contents of the *cgroup.subtree_control* file in the parent cgroup. Consequentially, a controller can be enabled only if the parent has the controller enabled and a controller can't be disabled if one or more children have it enabled. The picture below gives a good visual representation of the relationship between the *cgroup.subtree_control* and *cgroup.controllers* files in a cgroup hierarchy.



It must be noted that certain so-called implicit controllers are always available, and are not listed in *cgroup.controllers* (such as *freezer* and *perf_event*).

7.3 No internal process constraint

Non-root cgroups can distribute domain resources to their children only when they don't have any processes of their own. More precisely, the rule is that a nonroot cgroup can't both (1) have member processes, and (2) distribute resources into child cgroups—that is, have a nonempty *cgroup.subtree_control* file. This rules out situations where child cgroups compete against internal processes of the parent.

The root cgroup is exempt from this restriction. Root contains processes and anonymous resource consumption which can't be associated with any other cgroups and requires special treatment from most controllers.

Note that the restriction doesn't get in the way if there is no enabled controller in the cgroup's *cgroup.subtree_control*. This is important as otherwise it wouldn't be possible to create children of a populated cgroup. Thus, if a cgroup has both member processes and child cgroups, before controllers can

be enabled for that cgroup, the member processes must be moved out of the cgroup (e.g., perhaps into the child cgroups).

8 Resource distribution models

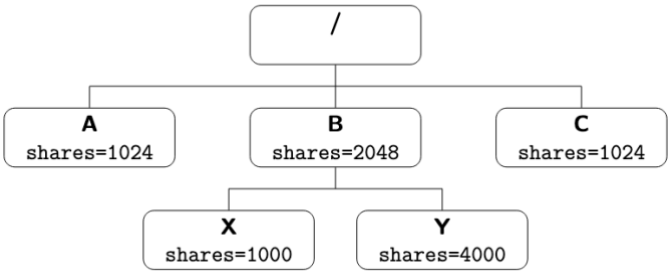
Cgroup controllers can implement different resource allocation models, depending on the specific resource type, which allow processes to exploit optimally the underlying hardware. This section describes the most relevant schemes available.

8.1 Weights

A parent's resource is distributed by adding up the weights of all active children and giving each the fraction matching the ratio of its weight against the sum. For example, given 10 cgroups, each with weight of value 100, the sum is 1000 and each cgroup receives one tenth of the resource. As only children which can make use of the resource at the moment participate in the distribution, this is work-conserving. This means that the resource that is subject to control is always used at full capabilities whenever a process is available, while the imposed limits only apply if there is competition for the resource.

All weights are in the range [1, 10000] with the default at 100.

The *cpu.weight* file, which allows to proportionally distributes CPU cycles to active children, is an example of this type.



In the above example, each cgroup holds some share (which is a term equivalent to weight) of the total cpu resources. All the shares in one level are added together to make a total and each cgroup in that level proportionally gets its share. Specifically:

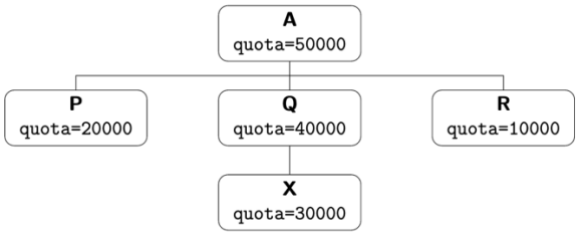
- Processes in B get $\frac{2048}{1024+2048+1024} = \frac{1}{2}$ of the CPU time
- Processes in A and C each get $\frac{1024}{1024+2048+1024} = \frac{1}{4}$ of the CPU time
- Processes in X get $\frac{2048}{1024+2048+1024} \cdot \frac{1000}{1000+4000} = \frac{1}{2} \cdot \frac{1}{5} = \frac{1}{10}$ of the CPU time
- Processes in Y get $\frac{2048}{1024+2048+1024} \cdot \frac{4000}{1000+4000} = \frac{1}{2} \cdot \frac{4}{5} = \frac{4}{10}$ of the CPU time

8.2 Limits

A strict limit indicating that each child can only consume up to the configured amount of the resource. A limited cgroup can't get more than his allocation of the resource, even if no competition is in place. However, limits can also be over-committed. Therefore, the sum of the limits of children can exceed the amount of resource available to the parent. This approach may or may not be work-conserving.

The proportion of the resource allocated to a cgroups is expressed as two numbers, a quota and a period. The quota/period ratio determines the fraction of the resource available to the cgroup. Limits are in the range [0, max] and defaults to "max", which is noop. As limits can be over-committed, all configuration combinations are valid and there is no reason to reject configuration changes or process migrations.

The *cpu.max* file, which allows to express the maximum CPU bandwidth limit for each child, is an example of this type.



In the above example, assuming that the period for each group is 100000

- Processes under A will get a maximum of 50% of the CPU
- Processes under Q will get a maximum of 40% of the CPU
- Processes under X will get a maximum of 30% of the CPU

The sibling cgroups under A are oversubscribed. In fact, P, Q and R can get an overall 70% of the CPU, while apparently the parent cgroup A con only get 50%. This however is not an issue. P, Q and R will never be able to actually obtain 70% of the CPU, as there is a higher-level limit of 50% imposed by the parent A.

8.3 Protections

A cgroup is protected up to the configured amount of the resource as long as the usages of all its ancestors are under their protected levels. It is in some sense an opposite paradigm to limits: if the resource usage is below the protection boundary, the kernel will try not to penalize this cgroup in favor of other cgroups that compete for the same resource. Protections can be hard guarantees or best effort soft boundaries. Protections can also be over-committed, in which case only up to the amount available to the parent is protected among children. This approach is work-conserving.

Protections are in the range [0, max] and defaults to 0, which is noop. As protections can be over-committed, all configuration combinations are valid and there is no reason to reject configuration changes or process migrations.

The *memory.low* file, which implements a best-effort memory protection mechanism, is an example of this type.

8.4 Allocations

A cgroup is exclusively allocated a certain amount of a finite resource. It is in some sense a stricter version of limits. Allocations can’t be over-committed. Hence, the sum of the allocations of children cannot exceed the amount of resource available to the parent

Allocations are in the range [0, max] and defaults to 0, which is no resource.

The *cpu.rt.max* file, which hard-allocates real-time slices, is an example of this type.

9 Core interface files

All cgroup core interface files are prefixed with "*cgroup.*". This section contains a list of such files.

9.1 cgroup.type

A read-write single value file which exists in all non-root cgroups.

It indicates, and can be used to change, the current type of the cgroup. The file always contains one of the following type values:

- *domain* : a normal v2 cgroup that provides process-granularity control. This is the default type.
- *threaded* : a cgroup that is a member of a threaded subtree.
- *domain threaded* : a cgroup that serves as the root of a threaded subtree.
- *domain invalid* : a cgroup inside a threaded subtree that is in an invalid state.

More information about these types and how to correctly use them can be found in section 13.

9.2 cgroup.procs

A read-write new-line separated values file which exists on all cgroups.

When read, it lists the PIDs of all processes which belong to the cgroup one-per-line. The PIDs are not ordered and the same PID may show up more than once if the process got moved to another cgroup and then back or the PID got recycled while reading.

Writing the value 0 to this file causes the writing process to be moved to the corresponding cgroup.

Within a hierarchy, a process can be a member of exactly one cgroup. Writing a process's PID to a *cgroup.procs* file automatically removes it from the cgroup of which it was previously a member.

Rules regarding writing to this file from unprivileged users are described in section 12.

Rules regarding the access to this file in threaded subtrees are discusses in section 13.

9.3 cgroup.threads

A read-write new-line separated values file which exists on all cgroups.

When read, it lists the TIDs of all threads which belong to the cgroup one-per-line. The TIDs are not ordered and the same TID may show up more than once if the thread got moved to another cgroup and then back or the TID got recycled while reading.

Rules regarding the ownership of this file during the delegation of a sub-hierarchy are discussed in section 12.

Rules regarding writing to this file within threaded subtrees are explained in section 13.

9.4 cgroup.controllers

A read-only space separated values file which exists on all cgroups.

It shows space separated list of all controllers available to the cgroup. The controllers are not ordered.

More information about this file can be found in section 7.

9.5 cgroup.subtree_control

A read-write space separated values file which exists on all cgroups. Starts out empty.

When read, it shows space separated list of the controllers which are enabled to control resource distribution from the cgroup to its children.

Space separated list of controllers prefixed with '+' or '-' can be written to enable or disable controllers. A controller name prefixed with '+' enables the controller and '-' disables. If a controller appears more than once on the list, the last one is effective. When multiple enable and disable operations are specified, either all succeed or all fail.

More information about this file can be found in section 7.

9.6 cgroup.events

A read-only flat-keyed file which exists on non-root cgroups.

Its contents are key-value pairs (delimited by newline characters, with the key and value separated by spaces) providing state information about the cgroup. Unless specified otherwise, a value change in this file generates a file modified event. The following keys appear in this file:

- *pupulated* : 1 if the cgroup or its descendants contains any live processes; otherwise, 0.
- *frozen* : 1 if the cgroup is frozen; otherwise, 0.

More information about this file can be found in section 11.

9.7 cgroup.max.descendants

A read-write single value file.

This file defines a limit on the number of live descendant cgroups that this cgroup may have. An attempt to create more descendants than allowed by the limit fails with an `EAGAIN` error.

The default value in this file is "max", which equates to no limit imposed.

9.8 cgroup.max.depth

A read-write single value file.

This file defines a limit on the depth of nesting of descendant cgroups. For example, a value of 0 in this file means that no descendant cgroups can be created. An attempt to create a descendant whose nesting level exceeds the limit fails with an `EAGAIN` error.

The default value in this file is "max", which equates to no limit imposed.

9.9 cgroup.stat

A read-only flat-keyed file which exists on all cgroups.

It contains the following key-value pairs:

- *nr_descendants* : the total number of visible (i.e., living) descendant cgroups underneath this cgroup.
- *nr_dying_descendants* : the total number of dying descendant cgroups underneath this cgroup. A cgroup enters the dying state after being deleted. It remains in that state for an undefined amount of time (which is dependent on system load) before being completely destroyed. A process can't be made a member of a dying cgroup, and a dying cgroup can't be brought back to life. A dying cgroup can consume system resources not exceeding limits, which were active at the moment of cgroup deletion.

9.10 cgroup.freeze

A read-write single value file which exists on non-root cgroups. Allowed values are “0” and “1”. The default is “0”.

Writing “1” to the file causes freezing of the cgroup and all descendant cgroups. This means that all belonging processes will be stopped and will not run until the cgroup will be explicitly unfrozen. Freezing of the cgroup may take some time; when this action is completed, the “frozen” value in the *cgroup.events* control file will be updated to “1” and the corresponding notification will be issued.

A cgroup can be frozen either by its own settings, or by settings of any ancestor cgroups. If any of the ancestor cgroups is frozen, the cgroup will remain frozen.

Processes in the frozen cgroup can be killed by a fatal signal. They also can enter and leave a frozen cgroup: either by an explicit move by a user, or if freezing of the cgroup races with *fork()*. If a process is moved to a frozen cgroup, it stops. If a process is moved out of a frozen cgroup, it becomes running.

Frozen status of a cgroup doesn’t affect any cgroup tree operations: it’s possible to delete a frozen (and empty) cgroup, as well as create new sub-cgroups.

9.11 cgroup.kill

A write-only single value file which exists in non-root cgroups. The only allowed value is “1”.

Writing “1” to the file causes the cgroup and all its descendant cgroups to be killed. This means that all processes located in the affected cgroup tree will be killed via `SIGKILL`.

Killing a cgroup tree will deal with concurrent forks appropriately and is protected against migrations.

In a threaded cgroup, writing this file fails with an `EOPNOTSUPP` error, as killing cgroups is a process directed operation, i.e., it affects the whole thread-group.

10 Controllers overview

This section contains a list of the resource controllers offered by cgroup v2.

10.1 CPU

The *cpu* controller regulates the distribution of CPU cycles and enforces CPU limits. This controller implements weight and absolute bandwidth limit models for normal scheduling policy, and absolute bandwidth allocation model only for real-time scheduling.

However, cgroup v2 doesn't yet support control of real-time processes. Hence, the *cpu* controller can only be enabled when all real-time processes are in the root cgroup. Some system management software may automatically place real-time processes into non-root cgroups during the system boot process, so these processes may need to be moved to the root cgroup before the *cpu* controller can be enabled.

In all the supported models, cycles distribution is defined only on a temporal base and it does not account for the frequency at which tasks are executed. Thus, the amount of computation completed by a task within an allocated bandwidth can be very different depending on the actual frequency the CPU is running that task.

The *cpu* interface files are:

- *cpu.stat* :
  A read-only flat-keyed file that exists whether the controller is enabled or not. It reports CPU time statistics using a set of values.
  The following statistics are always available:
    - *usage_usec* : total cpu time.
    - *user_usec* : user cpu time.
    - *system_usec* : system cpu time.

  The following statistics are instead only available when the controller is enabled:
    - *nr_periods* : number of full period intervals that have elapsed.
    - *nr_throttled* : number of times tasks in a cgroup have been throttled (that is, not allowed to run because they have exhausted all of the available time as specified by their quota).
    - *throttled_usec* : total time the tasks were not run due to being overquota (throttled).
    - *nr_bursts* : number of periods bandwidth burst occurs.
    - *burst_usec* : cumulative wall-time (in microseconds) that any cpu has used above quota in respective periods.
- *cpu.weight* :
  A single value file which exists on non-root cgroups, used to set a proportional bandwidth limit. The default is 100. The weight is in the range [1, 10000].
- *cpu.weight.nice* :
  A read-write single value file which exists on non-root cgroups. This is an alternative interface for *cpu.weight* and allows reading and setting weight using the same values used by *nice*. The default is 0. The nice value is in the range [-20, 19]. Because the range is smaller and granularity is coarser for the nice values, the read value is the closest approximation of the current weight.
- *cpu.max* :
  A read-write two value file which exists on non-root cgroups. It allows to express the maximum bandwidth limit. Its content is always in the following format: `$MAX $PERIOD`, which indicates that the group may consume up to `$MAX` in each `$PERIOD` duration. Using "max" for `$MAX` indicates no limit. If only one number is written, `$MAX` is updated. The default is "max 100000".
- *cpu.max.burst* :
  A read-write single value file which exists on non-root cgroups. The default is "0". The burst in the range [0, `$MAX`].
- *cpu.pressure* :
  A read-write nested-keyed file showing pressure stall information (PSI) for CPU. The PSI feature identifies and quantifies the disruptions caused by resource contentions and the time impact it has on complex workloads or even entire systems. Pressure information for each resource is exported

through the `some` line, which indicates the share of time in which at least some tasks are stalled on a given resource.

- *cpu.uclamp.min* :
  A read-write single value file which exists on non-root cgroups. Within the Resource Distribution Model, this file represents a protection. Hence it serves as a best-effort approach to preserve the minimum performance point (utilization) of tasks in a cgroup. The utilization is expressed as a percentage rational number, e.g., 12.34 for 12.34%. The default is "0", which corresponds to no utilization boosting. Once a value in the file is set, the tasks of the group will run at least at a minimum frequency which corresponds to the *uclamp.min* utilization. The requested minimum utilization (protection) is always capped by the current value for the maximum utilization (limit), i.e., *cpu.uclamp.max*.
- *cpu.uclamp.max* :
  A read-write single value file which exists on non-root cgroups. Within the Resource Distribution Model, this file represents a limit. Hence it serves as an approach to limit the maximum performance point (utilization) of tasks in a cgroup. The utilization is expressed as a percentage rational number, e.g., 98.76 for 98.76%. The default is "max", which corresponds to no utilization capping. Once a value in the file is set, the tasks of the group will run at least up to a maximum frequency which corresponds to the *uclamp.max* utilization.

10.2 Memory

The *memory* controller isolates the memory behavior of a group of tasks from the rest of the system and allows one to regulate its memory distribution. Memory is stateful and implements both limit and protection models. While not completely water-tight, all major memory usages by a given cgroup are tracked so that the total memory consumption can be accounted and controlled to a reasonable extent. Currently, the following types of memory usages are tracked:

- Userland memory - page cache and anonymous memory.
- Kernel data structures such as dentries and inodes.
- TCP socket buffers.

The *memory* interface files are:

- *memory.current* :
  A read-only single value file which exists on non-root cgroups. It shows the total amount of memory currently being used by the cgroup and its descendants. It includes page cache, in-kernel data structures such as inodes, and network buffers.
- *memory.min* :
  A read-write single value file which exists on non-root cgroups that specifies a minimum amount of memory the cgroup must always retain. It is a hard memory protection: if the memory usage of a cgroup is within its effective min boundary, the cgroup's memory won't be reclaimed by the system under any conditions. The default is "0". If there is no unprotected reclaimable memory available, `OOM` killer is invoked.

  Effective min boundary is limited by the *memory.min* values of all the ancestor cgroups. If there is *memory.min* overcommitment (child cgroup or cgroups are requiring more protected memory than parent will allow), then each child cgroup will get the part of parent's protection proportional to its actual memory usage below *memory.min*. Putting more memory than generally available under this protection is discouraged and may lead to constant `OOMs`.

  If a memory cgroup is not populated with processes, its *memory.min* is ignored.
- *memory.low* :
  A read-write single value file which exists on non-root cgroups and provides a best-effort memory protection mechanism. If the memory usage of a cgroup is within its effective low boundary, the

cgroup's memory won't be reclaimed unless there is no reclaimable memory available in unprotected cgroups. The default is "0". Above the effective low boundary (or effective min boundary if it is higher), pages are reclaimed proportionally to the overage, reducing reclaim pressure for smaller overages.

Effective low boundary is limited by *memory.low* values of all ancestor cgroups. If there is *memory.low* overcommitment (child cgroup or cgroups are requiring more protected memory than parent will allow), then each child cgroup will get the part of parent's protection proportional to its actual memory usage below *memory.low*. Putting more memory than generally available under this protection is discouraged.

- *memory.high* :

  A read-write single value file which exists on non-root cgroups. It specifies the memory usage throttle limit. This is the main mechanism to control memory usage of a cgroup. If a cgroup's usage goes over the high boundary, the processes of the cgroup are throttled and put under heavy reclaim pressure. Going over the high limit never invokes the OOM killer and under extreme conditions the limit may be breached. The default value is "max", meaning there is no limit.

- *memory.max* :

  A read-write single value file which exists on non-root cgroups. This is the memory usage hard limit, acting as the final protection mechanism: if a cgroup's memory usage reaches this limit and can't be reduced, the system OOM killer is invoked on the cgroup. Under certain circumstances, usage may go over the *memory.high* limit temporarily. When the high limit is used and monitored properly, *memory.max* serves mainly to provide the final safety net. The default is "max".

- *memory.oom.group* :

  A read-write single value file which exists on non-root cgroups. This file determines whether a cgroup should be treated as an indivisible workload by the OOM killer. If set, all tasks belonging to the cgroup or to its descendants (if the memory cgroup is not a leaf cgroup) are killed together or not at all. This can be used to avoid partial kills to guarantee workload integrity. If the OOM killer is invoked in a cgroup, it's not going to kill any tasks outside of this cgroup, regardless of the *memory.oom.group* values of ancestor cgroups.

- *memory.events* :

  A read-only flat-keyed file which exists on non-root cgroups, that shows the number of times certain memory events have occurred in the cgroup. Unless specified otherwise, a value change in this file generates a file modified event, which allow applications to track and monitor changes. The following entries are defined:

  - *low* : the number of times the cgroup is reclaimed due to high memory pressure even though its usage is under the low boundary. This usually indicates that the low boundary is over-committed.
  - *high* : the number of times processes of the cgroup are throttled and routed to perform direct memory reclaim because the high memory boundary was exceeded. For a cgroup whose memory usage is capped by the high limit rather than global memory pressure, this event's occurrences are expected.
  - *max* : the number of times the cgroup's memory usage was about to go over the max boundary. If direct reclaim fails to bring it down, the cgroup goes to OOM state.
  - *oom* : the number of time the cgroup's memory usage was about to reach the limit and allocation was about to fail. This event is not raised if the OOM killer is not considered as an option, e.g. for failed high-order allocations or if caller asked to not retry attempts.
  - *oom_kill* : the number of processes belonging to this cgroup killed by any kind of OOM killer.
  - *oom_group kill* : the number of times a group OOM has occurred.

- *memory.events.local* :
  Similar to *memory.events* but the fields in the file are local to the cgroup, i.e. not hierarchical. The file modified event generated on this file reflects only the local events.
- *memory.stat* :
  A read-only flat-keyed file which exists on non-root cgroups. This breaks down the cgroup's memory footprint into different types of memory, type-specific details, and other information on the state and past events of the memory management system. The list of all entries is extremely large and can be found in the kernel documentation [2].
- *memory.numa_stat* :
  A read-only nested-keyed file which exists on non-root cgroups. This breaks down the cgroup's memory footprint into different types of memory, type-specific details, and other information per node on the state of the memory management system. This is useful for providing visibility into the NUMA locality information within an memcg since the pages are allowed to be allocated from any physical node. One of the use cases is evaluating application performance by combining this information with the application's CPU allocation.
- *memory.swap.current* :
  A read-only single value file which exists on non-root cgroups. It contains the total amount of swap currently being used by the cgroup and its descendants.
- *memory.swap.high* :
  A read-write single value file which exists on non-root cgroups. The default is "max". Swap usage throttle limit. If a cgroup's swap usage exceeds this limit, all its further allocations will be throttled to allow userspace to implement custom out-of-memory procedures. This limit marks a point of no return for the cgroup. It is NOT designed to manage the amount of swapping a workload does during regular operation. Compare to *memory.swap.max*, which prohibits swapping past a set amount, but lets the cgroup continue unimpeded as long as other memory can be reclaimed. Healthy workloads are not expected to reach this limit.
- *memory.swap.max* :
  A read-write single value file which exists on non-root cgroups. The default is "max". Swap usage hard limit. If a cgroup's swap usage reaches this limit, anonymous memory of the cgroup will not be swapped out.
- *memory.swap.events* :
  A read-only flat-keyed file which exists on non-root cgroups. Unless specified otherwise, a value change in this file generates a file modified event. The following entries are defined:
    - *high* : the number of times the cgroup's swap usage was over the high threshold.
    - *max* : the number of times the cgroup's swap usage was about to go over the max boundary and swap allocation failed.
    - *fail* : the number of times swap allocation failed either because of running out of swap system-wide or max limit.

  When reduced under the current usage, the existing swap entries are reclaimed gradually and the swap usage may stay higher than the limit for an extended period of time. This reduces the impact on the workload and memory management.
- *memory.pressure* :
  A read-only nested-keyed file. A file containing memory pressure, a Pressure Stall Information (PSI) metric showing general memory health, as a measurement of the CPU time lost due to lack of memory. Provides a measurement of memory pressure that can be monitored by applications, which can use pressure thresholds to trigger various actions.

10.3 IO

The *IO* controller regulates the distribution of IO resources, allowing one to specify the number of IO operations per second (IOPS) that a cgroup can read or write. It's also capable of measuring and monitoring a cgroup's IO use and IO pressure. This controller implements both weight based and absolute bandwidth or IOPS limit distribution.

The *IO* interface files are:

- *io.stat* :
  A read-only nested-keyed file that reports IO usage statistics. The following keys are defined:
    - *rbytes* : bytes read.
    - *wbytes* : bytes written.
    - *dbytes* : number of bytes discarded.
    - *rios* : number of read IOs.
    - *wios* : number of write IOs.
    - *dios* : number of discard (or trim) IOs.
- *io.cost.qos* :
  A read-write nested-keyed file which exists only on the root cgroup. This file configures the Quality of Service of the IO cost model based controller (CONFIG_BLK_CGROUP_IOCOST) which currently implements *io.weight* proportional control.
- *io.cost.model* :
  A read-write nested-keyed file which exists only on the root cgroup. This file configures the cost model of the IO cost model based controller (CONFIG_BLK_CGROUP_IOCOST) which currently implements *io.weight* proportional control.
- *io.weight* :
  A read-write flat-keyed file which exists on non-root cgroups. The first line is the default weight applied to devices without specific override. The weights are in the range [1, 10000] and specifies the relative amount IO time the cgroup can use in relation to its siblings. The default is "default 100".
- *io.max* :
  A read-write nested-keyed file which exists on non-root cgroups. This is where IO limits can be specified. The following nested keys are defined:
    - *rbps* : max read bytes per second.
    - *wbps* : max write bytes per second
    - *riops* : max read IO operations per second.
    - *wiops* : max write IO operations per second.

  When writing, any number of nested key-value pairs can be specified in any order. "max" can be specified as the value to remove a specific limit. If the same key is specified multiple times, the outcome is undefined.
- *io.pressure* :
  A read-only nested-keyed file. It gives the percentage of wall time in which some or all tasks are waiting for a block device, or IO.

10.4 Cpuset

Cpuset provides a mechanism for assigning a set of CPUs and memory nodes to a set of tasks in a cgroup. This is especially valuable on large Non-Uniform Memory Access (NUMA) systems, where placing jobs on properly sized subsets of the systems with careful processor and memory placement to reduce cross-node memory access and contention can significantly improve overall system performance.

Cpuset constrains the CPU and memory placement of tasks to only the resources specified in the *cpuset* interface files in a task's current cgroup. The *cpuset* controller is hierarchical. That means the controller cannot use CPUs or memory nodes not allowed in its parent.

The *cpuset* interface files are:

- *cpuset.cpus* :

  A read-write multiple values file which exists on non-root cpuset-enabled cgroups. It lists the requested CPUs to be used by tasks within this cgroup. The actual list of CPUs to be granted, however, is subjected to constraints imposed by its parent and can differ from the requested CPUs. The CPU numbers are expressed through a comma-separated list of numbers or ranges, where ranges are represented through dashes. For example: 0-2,16, represents CPUs 0, 1, 2 and 16. An empty value indicates that the cgroup is using the same setting as the nearest cgroup ancestor with a non-empty *cpuset.cpus* or all the available CPUs if none is found.

- *cpuset.cpus.effective* :

  A read-only multiple values file which exists on all cpuset-enabled cgroups. It lists the onlined CPUs that are actually granted to this cgroup by its parent. These CPUs are allowed to be used by tasks within the current cgroup. If *cpuset.cpus* is empty, the *cpuset.cpus.effective* file shows all the CPUs from the parent cgroup that can be available to be used by this cgroup. Otherwise, it should be a subset of *cpuset.cpus* unless none of the CPUs listed in *cpuset.cpus* can be granted. In this case, it will be treated just like an empty *cpuset.cpus*.

- *cpuset.mems* :

  A read-write multiple values file which exists on non-root cpuset-enabled cgroups. It lists the requested memory nodes to be used by tasks within this cgroup. The actual list of memory nodes granted, however, is subjected to constraints imposed by its parent and can differ from the requested memory nodes. The memory node numbers are expressed through a comma-separated list of numbers or ranges, where ranges are represented through dashes. For example: 0-1,3, represents memory nodes 0, 1 and 3. Setting a non-empty value to *cpuset.mems* causes memory of tasks within the cgroup to be migrated to the designated nodes if they are currently using memory outside of the designated nodes. There is of course a cost for this memory migration. The migration may not be complete and some memory pages may be left behind. So it is recommended that *cpuset.mems* should be set properly before spawning new tasks into the *cpuset*. Even if there is a need to change *cpuset.mems* with active tasks, it shouldn't be done frequently.

- *cpuset.mems.effective* :

  A read-only multiple values file which exists on all cpuset-enabled cgroups. It lists the onlined memory nodes that are actually granted to this cgroup by its parent. These memory nodes are allowed to be used by tasks within the current cgroup. If *cpuset.mems* is empty, it shows all the memory nodes from the parent cgroup that will be available to be used by this cgroup. Otherwise, it should be a subset of *cpuset.mems* unless none of the memory nodes listed in *cpuset.mems* can be granted. In this case, it will be treated just like an empty *cpuset.mems*.

- *cpuset.cpu.partition* :

  A read-write single value file which exists on non-root cpuset-enabled cgroups. This flag is owned by the parent cgroup and is not delegatable. It accepts only the following input values when written to:

  - *root* : a partition root.
  - *member* : a non-root member of a partition.

  When set to be a partition root, the current cgroup is the root of a new partition or scheduling domain that comprises itself and all its descendants except those that are separate partition roots themselves and their descendants. The root cgroup is always a partition root. There are constraints

on where a partition root can be set. It can only be set in a cgroup if all the following conditions are true:

- The *cpuset.cpus* is not empty and the list of CPUs are exclusive, i.e. they are not shared by any of its siblings.
- The parent cgroup is a partition root.
- The *cpuset.cpus* is also a proper subset of the parent's *cpuset.cpus.effective*.
- There is no child cgroups with cpuset enabled. This is for eliminating corner cases that have to be handled if such a condition is allowed.

Setting it to partition root will take the CPUs away from the effective CPUs of the parent cgroup. Once it is set, this file cannot be reverted back to "member" if there are any child cgroups with cpuset enabled.

A parent partition cannot distribute all its CPUs to its child partitions. There must be at least one cpu left in the parent partition.

Once becoming a partition root, changes to *cpuset.cpus* is generally allowed as long as the first condition above is true, the change will not take away all the CPUs from the parent partition and the new *cpuset.cpus* value is a superset of its children's *cpuset.cpus* values.

10.5 PIDs

This controller permits limiting the number of processes that may be created in a cgroup (and its descendants). After a specified limit is reached, it stops any new tasks from being forked or cloned.

The number of tasks in a cgroup can be exhausted in ways which other controllers cannot prevent, thus warranting its own controller. For example, a fork bomb is likely to exhaust the number of tasks before hitting memory restrictions.

It's relevant to highlight that the PIDs used in this controller actually refer to TIDs, which are the identifiers for all the schedulable objects in the kernel.

The *PIDs* interface files are:

- *pids.max* : a read-write single value file which exists on non-root cgroups. Hard limit of number of processes. The default is "max", which equates to no limit.
- *pids.current* : a read-only single value file which exists on all cgroups. The number of processes currently in the cgroup and its descendants.

Organizational operations are not blocked by cgroup policies, so it is possible to have *pids.current* > *pids.max*. This can be done by either setting the limit to be smaller than *pids.current*, or attaching enough processes to the cgroup such that *pids.current* is larger than *pids.max*. However, it is not possible to violate a cgroup PID policy through *fork()* or *clone()*. These will return `-EAGAIN` if the creation of a new process would cause a cgroup policy to be violated.

10.6 RDMA

The *RDMA* controller regulates the distribution and accounting of RDMA resources per cgroup. It's used to allow a cgroup hierarchy to stop processes from consuming additional RDMA resources after a certain limit is reached.

Remote Direct Memory Access (RDMA) is an extension of the Direct Memory Access (DMA) technology. It enables two networked computers to exchange data in main memory without relying on the processor, cache or operating system of either device. This permits high-throughput, low-latency and overall high-performance networking.

The *RDMA* interface files are:

- *rdma.max* :

  A readwrite nested-keyed file that exists for all the cgroups except root that describes current configured resource limit for a RDMA/IB device. Lines are keyed by device name and are not

ordered. Each line contains space separated resource name and its configured limit that can be distributed. The following nested keys are defined.:

- *hca_handle* : maximum number of HCA Handles.
- *hca_object* : maximum number of HCA Objects.

- *rdma.current* :
  A read-only file that describes current resource usage. It exists for all the cgroup except root.

10.7 HugeTLB

The HugeTLB controller allows to limit the HugeTLB usage per control group and enforces the controller limit during page fault.

The HugeTLB feature of the Linux kernel enables one to use virtual memory pages of large sizes. A proper use of this feature can significantly reduce the amount of TLB misses and improve the overall memory performance.

The *HugeTLB* interface files are:

- *hugetlb.<hugepagesize>.current* : show current usage for "hugepagesize" hugetlb. It exists for all the cgroup except root.
- *hugetlb.<hugepagesize>.max* : set/show the hard limit of "hugepagesize" hugetlb usage. The default value is "max". It exists for all the cgroup except root.
- *hugetlb.<hugepagesize>.events* : a read-only flat-keyed file which exists on non-root cgroups.
- *hugetlb.<hugepagesize>.events.local* : similar to *hugetlb.<hugepagesize>.events* but the fields in the file are local to the cgroup i.e. not hierarchical. The file modified event generated on this file reflects only the local events.
- *hugetlb.<hugepagesize>.numa_stat* : similar to *memory.numa_stat*, it shows the numa information of the hugetlb pages of <hugepagesize> in this cgroup. Only active in use hugetlb pages are included. The per-node values are in bytes.

10.8 perf_event

Enabling this controller allows one to monitor through the Linux *perf* subsystem the set of processes and threads grouped in a cgroup, as opposed to monitoring each process or thread separately or per-CPU.

The Linux *perf* tool is a performance analyzing tool that features advanced performance and troubleshooting functions. It is a key feature for basic profiling of applications' behavior.

It is in some sense a special type of controller. In fact, it never appears in the *cgroup.controllers* file of any cgroup. However, it is implicitly always available in all the cgroups throughout the hierarchy.

11 Release notifications

Cgroup v2 provides a mechanism for obtaining a notification whenever a cgroup becomes empty. A cgroup is considered to be empty when it contains no live member processes in itself or its child cgroups. Such a feature was already available in cgroup v1, however the design has been significantly improved in the newer version.

This type of information could be useful for several purposes. For example, a manager process might be interested in knowing when all workers have terminated. Alternatively, it could be used to start a clean-up operation after all processes of a given sub-hierarchy have exited.

Each non-root cgroup has a *cgroup.events* file which contains key-value pairs with state information about the cgroup. Currently, only two keys with boolean values exist:

- *populated* : the key value is 0 if there is no live process in the cgroup and its descendants; 1 otherwise.
- *frozen* : the key value is 1 if this cgroup is currently frozen; 0 otherwise.

It is possible to monitor the *cgroup.events* file in order to receive a notification when the value of one of its keys changes. The monitoring is generally performed in two possible ways:

- inotify : this is a subsystem of the Linux kernel which monitors changes to the filesystem, and reports those changes to user-space processes. More information can be found in [9].
- poll : a feature that allows to wait for some event on a file descriptor. More information can be found in [10].

In any case, after a notification occurs, the *cgroup.events* file gets parsed to find out what changes occurred in the *populated* key.

The populated state updates and notifications are recursive. Let's consider the following sub-hierarchy where the numbers in the parentheses represent the numbers of processes in each cgroup

```
A(4) - B(0) - C(1)
            \ D(0)
```

A, B and C's *populated* fields would be 1, while D's 0. After the one process in C exits, B and C's *populated* fields would flip to 0 and file modified events will be generated on the *cgroup.events* files of both cgroups.

One of the noticeable advantages of this mechanism, compared to the one available in cgroup v1, is that a single process can monitor multiple *cgroup.events* files. An interesting consequence is that different processes can be used to monitor *cgroup.events* files in different subhierarchies.

12 Delegation

In the context of cgroups, delegation means passing management of some subtree of the cgroup hierarchy to a nonprivileged user. Therefore, by using the delegation mechanism, some parts of the cgroup tree may be managed by different managers than others. In this paradigm, a delegater is a privileged user (i.e., root) who owns a parent cgroup. A delegatee is instead a nonprivileged user who will be granted the permissions needed to manage some subhierarchy under that parent cgroup, known as the delegated subtree.

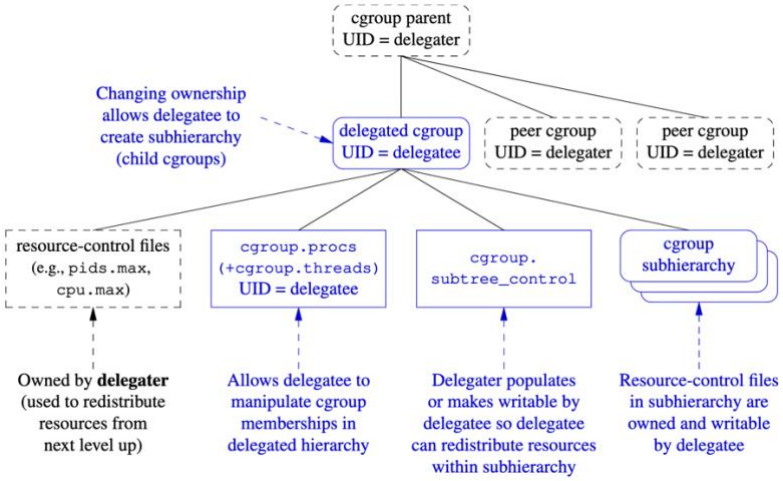
To set up delegation, the delegater makes the target cgroup writable by the delegatee. To do that, the ownership of the directory that will be the root of the delegated subtree is changed to the user ID of the delegatee (UID). In addition, the ownership of some of the files contained in that folder must also be changed. These generally include:

- *cgroup.procs* : this allows the delegatee to move processes into the root of the delegated subtree.
- *cgroup.subtree_control* : the delegatee can enable controllers in order to further redistribute resources at lower levels in the subtree.
- *cgroup.threads* : it's necessary to change the ownership of this file only if a threaded subtree is being delegated. This permits the delegatee to write thread IDs to the file.

Sometimes, it might be necessary to change the ownership of some other files, depending on the kernel version employed. The set of files whose ownership must be changed during the delegation step is listed in a pseudo-file of the kernel: `sys/kernel/cgroup/delegate`.

What the delegater shouldn't change instead is the ownership of any of the controller interface files in the root of the delegated subtree. In fact, the controller interface files are owned by the parent, which is the immediate level above the delegated subtree. The parent cgroup uses these files to manage resource allocation in the delegated subtree, while the delegatee should not have the rights to modify restrictions imposed by the upper levels.

The picture below provides an example of the delegation step.



The parent cgroup is owned by the delegater (e.g., root). The ownership of one of its children is changed to the UID of the delegatee, along with the *cgroup.procs* and *cgroup.subtree_control* files contained in that folder. The other files in the folder instead, which are the resource-control files, remain owned by the delegater. After the first delegation step, the delegatee is able to create child cgroups, which will inherit their ownership from the father, and hence will be owned by the delegatee as well. All the files in those child cgroups, including the resource-control files, will be owned by the delegatee, that is free to distribute resouces in the subhierarchy at its preference.

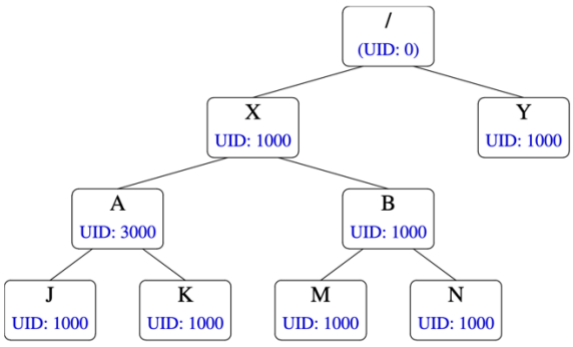
12.1 Delegation containment

A delegated sub-hierarchy is contained in the sense that processes can't be moved into or out of the sub-hierarchy by the delegatee. Hence, some delegation containment rules ensure that the delegatee can freely move processes between cgroups within the delegated subtree, but can't move processes from outside the delegated subtree into the subtree or vice versa.

A nonprivileged process (i.e., the delegatee) can write the PID of a target process into a *cgroup.procs* file only if the following conditions are satisfied:

- The writer must have write permission on the *cgroup.procs* file in the destination cgroup.
- The writer must have write permission on the *cgroup.procs* file in the nearest common ancestor of the source and destination cgroups. It might be possible in some cases for the nearest common ancestor to be the source or destination cgroup itself.

One noticeable consequence of these delegation containment rules is that the unprivileged delegatee can't place the first process into the delegated subtree. In fact, the processes initially belongs to a source cgroup outside of the delegated subtree, while the destination cgroup containing the *cgroup.procs* file is located at the root of the delegated subtree. Hence, their nearest common ancestor is for sure going to be located outside of the delegated subtree, so that, according to the second containment rule, the delegatee won't have permission to write on its *cgroup.procs* file. It is the delegater then that must place the first process into the delegated subtree.



In the example above, the root cgroup is owned by UID 0, while the other cgroups have been delegated and are owned by different users. According to the delegation containment rules, a process with UID 1000 can move a target process from cgroup M to N. In fact, UID 1000 has write permission on both the *cgroup.procs* file of the destination cgroup N and of the common ancestor B. Likewise, a process with UID 1000 can move a PID from M to X. In this case, X is both the destination cgroup and the nearest common ancestor. Conversely, a process with UID 1000 can't move a PID from X to Y, as it lacks write permission on the *cgroup.procs* file of the nearest common ancestor, which is the root cgroup.

12.2 nsdelegate

Starting with Linux 4.13, there is a second way to perform delegation in the cgroup v2 hierarchy. This is done by mounting or remounting the cgroup v2 filesystem with the `nsdelegate` mount option. For example, if the cgroup v2 filesystem has already been mounted, it's possible to remount it with the nsdelegate option as follows:

```
mount -t cgroup2 -o remount,nsdelegate \
                    none /sys/fs/cgroup/unified
```

Another option, useful on modern systems where *systemd* automatically mounts the cgroup v2 filesystem, is to boot the kernel with the following command-line options:

```
cgroup_no_v1=all systemd.legacy_systemd_cgroup_controller
```

These options cause the kernel to boot with the cgroup v1 controllers disabled and tells *systemd* not to mount and use the cgroup v2 hierarchy, so that the v2 hierarchy can be manually mounted with the desired options after boot-up.

The effect of the mount option aforementioned is to cause cgroup namespaces to automatically become delegation boundaries. The following restrictions apply for processes inside a cgroup namespace:

- Writes to controller interface files in the root directory of the namespace will fail. Processes inside the cgroup namespace can still write to delegatable files in the root directory of the cgroup namespace such as *cgroup.procs* and *cgroup.subtree_control*, and can create a subhierarchy underneath the root directory.
- Attempts to migrate processes across the namespace boundary are denied. Processes inside the cgroup namespace can still (subject to the containment rules) move processes between cgroups within the subhierarchy under the namespace root.

The `nsdelegate` mount option only has an effect when performed in the initial mount namespace, and its end results are equivalent to the other previously described delegation type.

13 Threads

Since Linux 4.14, cgroup v2 supports cgroup membership thread level granularity for a subset of controllers. This was achieved by introducing the so-called thread mode, that allows the creation of threaded subtrees in which the threads of a process may be spread across cgroups inside the tree. To distinguish the

current operation mode of a group, each nonroot cgroup contains a file named *cgroup.type*, that exposes and can be used to change the type of that cgroup. The file can contain one of the following types:

- *domain* : a normal v2 cgroup that provides process-level granularity control. If a process is a member of this cgroup, then all threads of the process are in the same cgroup. This is the default cgroup type.
- *threaded* : a cgroup that is a member of a threaded subtree. Threads can be added to this cgroup, and controllers can be enabled for the cgroup.
- *domain threaded* : a cgroup that serves as the root of a threaded subtree.
- *domain invalid* : a cgroup inside a threaded subtree that is in an invalid state. Processes can't be added to the cgroup and controllers can't be enabled. The only thing that can be done with this cgroup (other than deleting it) is to convert it to a threaded cgroup by writing the string *threaded* to its *cgroup.type* file.

Selected subtrees in the overall hierarchy can be switched to threaded mode. All members of a threaded subtree must be threaded cgroups and all the threads belonging to a specific process must necessarily reside in the same subtree.

13.1 Threaded and domain controllers

Cgroup v2 distinguishes two types of resource controllers, depending on whether or not they’re capable of supporting thread mode:

- Threaded controllers : these controllers support thread-level granularity for resource management and can be enabled inside threaded subtrees. As a result, the corresponding controller-interface files appear inside the cgroups in a threaded subtree. Currently, only the following controllers are threaded: *cpu*, *perf_event*, and *pids*.
- Domain controllers : these controllers support only process granularity for resource control. From the perspective of a domain controller, all threads of a process are always in the same cgroup, which is the root of the threaded subtree. This type of controllers can't be enabled inside a threaded subtree, as their control-interface files don't appear inside the cgroups within it. Therefore, the lower level at which a domain controller can do resource management is the root of the threaded subtree.

13.2 Creation of a threaded subtree

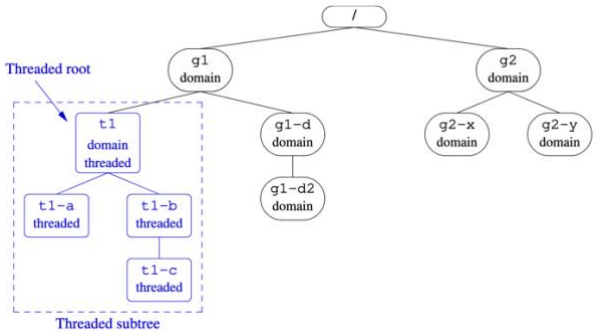
There are fundamentally two ways to create a threaded subtree.

The first one is to write the string *threaded* to the *cgroup.type* file of a cgroup that currently has the type domain. This causes the type of the parent cgroup to be automatically converted to *domain threaded*. Hence, the parent cgroup becomes the root of a threaded subtree, known as threaded root. Moreover, all the child cgroups of the newly created threaded root that were not already of type *threaded* are converted to *domain invalid*. Any subsequently created cgroups under the threaded root will also have the type *domain invalid* by default. After that, the type of each *domain invalid* cgroup under the threaded root must be changed to *threaded* by modifying the appropriate *cgroup.type* files. This step allows to get rid of all invalid cgroups and to make the threaded subtree fully usable.

The second possible way is to enable one or more threaded controllers and add a member process to a *domain* type cgroup. These two actions cause the type of that cgroup to be automatically converted into *domain threaded*, while all of its descendant cgroups that were not already of type *threaded* are automatically converted to *domain invalid*. As with the previous method, the next step is to convert each *domain invalid* cgroup into *threaded* to make the threaded subtree actually usable.

A noticeable consequence of these methods is that the threaded root can be a parent only to *threaded* and *domain invalid* cgroups. Hence, the threaded root can't be a parent of a *domain* cgroup, and a *threaded* cgroup can't have a sibling that is a *domain* cgroup.

The picture below provides an example of a cgroup hierarchy that makes use of a threaded subtree.



13.3 Using threaded subtrees

A process can be moved into a threaded subtree by writing its PID to the *cgroup.procs* file in any of the cgroups inside the tree. This has the effect of making all the threads in the process members of the corresponding cgroup, while the process itself becomes a member of the threaded root. Hence, the PID of the newly moved process will be automatically written inside the *cgroup.procs* file of the threaded root, while the TIDs of all its threads will be written in the *cgroup.threads* file of the destination cgroup. The threads can then be spread further across the subtree by writing their TIDs to the *cgroup.threads* files in the appropriate cgroups, with the restriction that all threads belonging to the same process must necessarily reside in the same threaded subtree.

Some containment rules apply when writing to *cgroup.threads* files:

- The writer must have write permission on the *cgroup.threads* file in the destination cgroup.
- The writer must have write permission on the *cgroup.procs* file in the common ancestor of the source and destination cgroups. In some cases, the common ancestor may be the source or destination cgroup itself.
- The source and destination cgroups must be in the same threaded subtree.

The *cgroup.threads* file is present in each cgroup (including domain cgroups) and can be read in order to discover the set of threads belonging to the cgroup.

The *cgroup.procs* file in the threaded root shows the PIDs of all processes that are members of the threaded subtree. The *cgroup.procs* files in the other cgroups in the subtree are instead not readable: any attempt to read them results in an error.

The PIDs of all processes moved inside a threaded subtree are automatically written to the *cgroup.procs* file of the threaded root, so all the processes of the subtree effectively belong to that cgroup. As a result, *domain* controllers cannot be enabled inside a threaded subtree and no controller-interface file appears inside the cgroups underneath the threaded root. From the point of view of a *domain* controller, threaded subtrees are invisible: a multithreaded process inside a threaded subtree appears to a *domain* controller as a process that resides in the threaded root cgroup.

Within a threaded subtree, the no internal processes rule does not apply: a non-leaf cgroup can both contain member processes (or threads) and exercise control on its child cgroups. Because of this, *threaded* controllers must be able to handle competition between the threads in a non-leaf cgroup and its child cgroups. Each *threaded* controller defines how such competitions are handled.

13.4 Writing to cgroup.type files

A number of rules apply when writing to *cgroup.type* files.

First, the string *threaded* is the only one that can be written. In other words, the only possible explicit transition is to convert a *domain* type cgroup to *threaded*.

The effect of writing *threaded* depends on the current value in *cgroup.type*, as follows:

- *domain* or *domain threaded* : starts the creation of a threaded subtree, whose root is the parent of this cgroup.

- *domain invalid* : converts this cgroup (which is inside a threaded subtree) to a usable state.
- *threaded* : no effect.

It's not possible to write to a *cgroup.type* file of a cgroup whose parent type is *domain invalid*. In other words, the cgroups of a threaded subtree must be converted to the *threaded* state in a top-down manner.

Lastly, there are also some constraints that must be satisfied in order to create a threaded subtree rooted at a certain target cgroup:

- There can be no member processes in the descendants of the target cgroup. However, the target cgroup itself is allowed to have member processes.
- No domain controller can be enabled in target cgroup *cgroup.subtree_control* file.

Any violation of these constraints results in a `ENOTSUP` error.

13.5 The domain threaded cgroup type

To sum up, the type of a cgroup can change to *domain threaded* in either of the following cases:

- The string *threaded* is written to one of its child cgroups.
- A threaded controller is enabled inside the cgroup and a process is made a member of the cgroup.

A *domain threaded* cgroup reverts to the type *domain* if and only if the above conditions no longer hold true, hence if all its threaded child cgroups are removed and either the cgroup no longer has threaded controllers enabled or no longer has member processes.

When a *domain threaded* cgroup reverts to the type *domain*:

- All its *domain invalid* descendants that are not in lower-level threaded subtrees revert to the type domain.
- The root cgroups in any lower-level threaded subtrees revert to the type *domain threaded*.

13.6 Exceptions for the root cgroup

The root cgroup of the v2 hierarchy is treated exceptionally: it can be the parent of cgroups of type both *domain* and *threaded*. Therefore, the root of the hierarchy is the only cgroup that can simultaneously be a threaded root and have children of type *domain*. When the string *threaded* is written to the *cgroup.type* file of one of the children of the root cgroup, then:

- The type of that cgroup becomes *threaded*.
- The type of any descendants of that cgroup that are not part of lower-level threaded subtrees changes to *domain invalid*.

So, in this case there is no cgroup whose type becomes *domain threaded*, although the root of the hierarchy can still be conceptually considered as the threaded root for the cgroup whose type was changed to *threaded*.

13.7 A note on real-time threads

As of now, the cgroups v2 *cpu* controller does not support control of real-time threads. Therefore, the *cpu* controller can be enabled in the root cgroup only if all real-time threads are in the root cgroup. On some systems, *systemd* automatically places certain real-time threads in nonroot cgroups in the v2 hierarchy. On such systems, these threads must first be moved to the root cgroup before the *cpu* controller can be enabled.

References

[1] cgroups(7) - Linux manual page (https://man7.org/linux/man-pages/man7/cgroups.7.html)

[2] Linux kernel documentation (https://www.kernel.org/doc/Documentation/cgroup-v2.txt)

[3] Michael Kerrisk. What's new in control groups (cgroups) v2 - linux.conf.au 2019; 23 January 2019, Christchurch, New Zealand (https://man7.org/conf/lca2019/cgroups_v2-LCA2019-Kerrisk.pdf)

[4] Michael Kerrisk. An introduction to control groups (cgroups) version 2 - NDC TechTown 2021; 20 October 2021, Kongsberg, Norway (https://www.man7.org/conf/ndctechtown2021/cgroups-v2-part-1-intro-NDC-TechTown-2021-Kerrisk.pdf)

[5] Michael Kerrisk. Diving deeper into control groups (cgroups) version 2 - NDC TechTown 2021; 20 October 2021, Kongsberg, Norway (https://www.man7.org/conf/ndctechtown2021/cgroups-v2-part-2-diving-deeper-NDC-TechTown-2021-Kerrisk.pdf)

[6] Product Documentation for Red Hat Enterprise Linux (https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/managing_monitoring_and_updating_the_kernel/assembly_configuring-resource-management-using-systemd_managing-monitoring-and-updating-the-kernel)

[7] Tejun Heo. Control of major resources in cgroup v2 (https://www.socallinuxexpo.org/sites/default/files/presentations/SCALE%2015x%20-%20cgroup.pdf)

[8] Maximizing Resource Utilization with cgroup2 (https://facebookmicrosites.github.io/cgroup2/docs/overview.html)

[9] inotify(7) - Linux manual page (https://man7.org/linux/man-pages/man7/inotify.7.html)

[10] poll(2) - Linux manual page (https://man7.org/linux/man-pages/man2/poll.2.html)