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

Cgroups 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 cgroups, 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`.

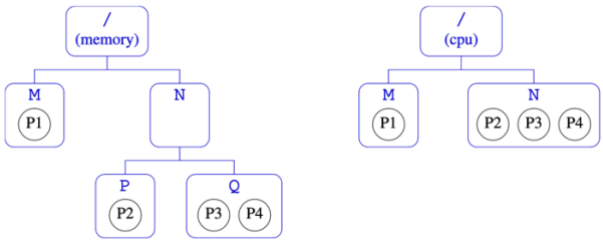
Resource controllers generally perform some resource limitation, monitoring or other type of management of groups of processes (for example the CPU controller limits the CPU consumption). Following certain structural constraints, controllers may be enabled or disabled selectively on a cgroup by defining 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.

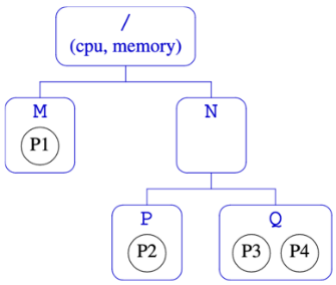
4.1 Multiple hierarchies

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 level of granularities. 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 cgroups v1 hierarchy and in the cgroups v2 hierarchy. This allows mixing v2 hierarchy with the legacy v1 multiple hierarchies 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 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 cgroups 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.

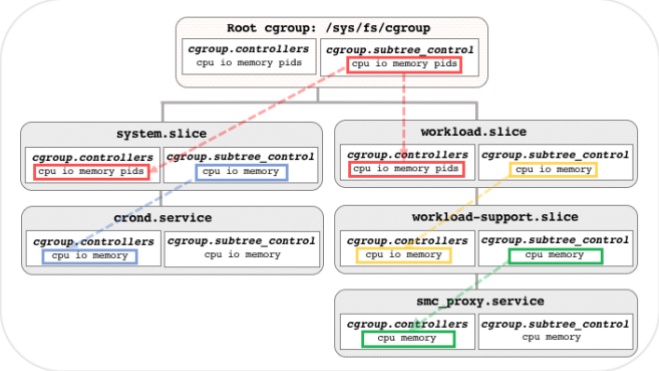
When a controller is present in the *cgroup.subtree_control* file of a parent cgroup, then the corresponding controller-interface files are automatically created in the children of that cgroup and can be used to exert resource control in the child cgroups.. In the above example, enabling *cpu* on B would 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

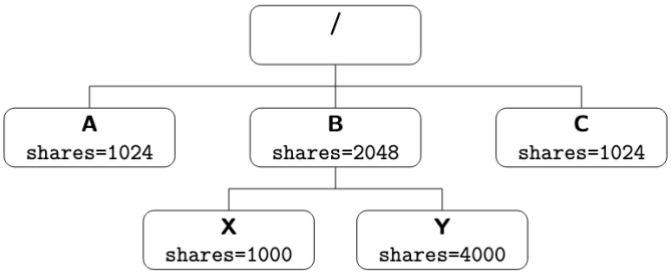
Cgroup controllers can implement different resource allocation models, depending on the specific resource type, which allow processes to exploit optimally the underlying hardware. This section describes the most relevant schemes available.

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:

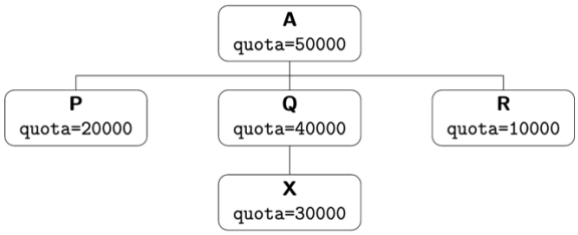
- Processes in B get $\frac{2048}{1024+2048+1024} = \frac{1}{2}$ of the CPU time
- Processes in A and C each get $\frac{1024}{1024+2048+1024} = \frac{1}{4}$ of the CPU time
- Processes in X get $\frac{2048}{1024+2048+1024} \cdot \frac{1000}{1000+4000} = \frac{1}{2} \cdot \frac{1}{5} = \frac{1}{10}$ of 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 is a list of such files. Further details about each file can be found in [2].

- *cgroup.type* : indicates the current type of the cgroup
- *cgroup.procs* : lists the PIDs of all processes which belong to the cgroup
- *cgroup.threads* : lists the TIDs of all threads which belong to the cgroup
- *cgroup.controllers* : lists all controllers available to the cgroup
- *cgroup.subtree_control* : lists the controllers which are enabled to control resource distribution from the cgroup to its children
- *cgroup.events* : a value change in this file generates a file modified event
- *cgroup.max.descendants* : maximum allowed number of descent cgroups

- *cgroup.max.depth* : maximum allowed descent depth below the current cgroup
- *cgroup.stat*

10 Controllers overview

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

10.1 CPU

Interface files:

- *cpu.stat* : reports statistics
- *cpu.weight*
- *cpu.weight.nice*
- *cpu.max* : maximum bandwidth limit

10.2 Memory

Interface files:

- *memory.current* : reports statistics
- *memory.low*
- *memory.high*
- *memory.max*
- *memory.events*
- *memory.stat*
- *memory.swap.current*
- *memory.swap.max*

10.3 IO

Interface files:

- *io.stat* : reports statistic
- *io.weight*
- *io.ma*

10.4 PIDs

Interface files:

- *pids.max* : reports statistics
- *pids.current*

10.5 RDMA

Interface files:

- *rdma.max* : reports statistic
- *rdma.current*

10.6 perf_event

Text

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.

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)