```
================
Control Group v2
================

:Date: October, 2015
:Author: Tejun Heo <tj@kernel.org>

This is the authoritative documentation on the design, interface and
conventions of cgroup v2.  It describes all userland-visible aspects
of cgroup including core and specific controller behaviors.  All
future changes must be reflected in this document.  Documentation for
v1 is available under Documentation/cgroup-v1/.

.. CONTENTS

   1. Introduction
     1-1. Terminology
     1-2. What is cgroup?
   2. Basic Operations
     2-1. Mounting
     2-2. Organizing Processes and Threads
       2-2-1. Processes
       2-2-2. Threads
     2-3. [Un]populated Notification
     2-4. Controlling Controllers
       2-4-1. Enabling and Disabling
       2-4-2. Top-down Constraint
       2-4-3. No Internal Process Constraint
     2-5. Delegation
       2-5-1. Model of Delegation
       2-5-2. Delegation Containment
     2-6. Guidelines
       2-6-1. Organize Once and Control
       2-6-2. Avoid Name Collisions
   3. Resource Distribution Models
     3-1. Weights
     3-2. Limits
     3-3. Protections
     3-4. Allocations
   4. Interface Files
     4-1. Format
     4-2. Conventions
     4-3. Core Interface Files
   5. Controllers
     5-1. CPU
       5-1-1. CPU Interface Files
     5-2. Memory
       5-2-1. Memory Interface Files
       5-2-2. Usage Guidelines
       5-2-3. Memory Ownership
     5-3. IO
       5-3-1. IO Interface Files
       5-3-2. Writeback
     5-4. PID
       5-4-1. PID Interface Files
     5-5. Device
     5-6. RDMA
       5-6-1. RDMA Interface Files
```

Introduction

Terminology

"cgroup" stands for "control group" and is never capitalized. The singular form is used to designate the whole feature and also as a qualifier as in "cgroup controllers". When explicitly referring to multiple individual control groups, the plural form "cgroups" is used.

What is cgroup?

cgroup is a mechanism to organize processes hierarchically and distribute system resources along the hierarchy in a controlled and configurable manner.

cgroup is largely composed of two parts - the core and controllers. cgroup core is primarily responsible for hierarchically organizing processes. 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.

cgroups form a tree structure and every process in the system belongs to one and only one cgroup. All threads of a process belong to the same cgroup. On creation, all processes are put in the cgroup that the parent process belongs to at the time. A process can be migrated to another cgroup. Migration of a process doesn't affect already existing descendant processes.

Following certain structural constraints, controllers may be enabled or disabled selectively on a cgroup. All controller behaviors are hierarchical - if a controller is enabled on a cgroup, it affects all

processes which belong to the cgroups consisting the inclusive
sub-hierarchy of the cgroup. When a controller is enabled on a nested
cgroup, it always restricts the resource distribution further. The
restrictions set closer to the root in the hierarchy can not be
overridden from further away.

Basic Operations

Mounting

Unlike v1, cgroup v2 has only single hierarchy. The cgroup v2
hierarchy can be mounted with the following mount command::

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

cgroup2 filesystem has the magic number 0x63677270 ("cgrp"). All
controllers which support v2 and are not bound to a v1 hierarchy are
automatically bound to the v2 hierarchy and show up at the root.
Controllers which are not in active use in the v2 hierarchy can be
bound to other hierarchies. This allows mixing v2 hierarchy with the
legacy v1 multiple hierarchies in a fully backward compatible way.

A controller can be moved across hierarchies only after the controller
is no longer referenced in its current hierarchy. Because per-cgroup
controller states are destroyed asynchronously and controllers may
have lingering references, a controller may not show up immediately on
the v2 hierarchy after the final umount of the previous hierarchy.
Similarly, a controller should be fully disabled to be moved out of
the unified hierarchy and it may take some time for the disabled
controller to become available for other hierarchies; furthermore, due
to inter-controller dependencies, other controllers may need to be
disabled too.

While useful for development and manual configurations, moving
controllers dynamically between the v2 and other hierarchies is
strongly discouraged for production use. It is recommended to decide
the hierarchies and controller associations before starting using the
controllers after system boot.

During transition to v2, system management software might still
automount the v1 cgroup filesystem and so hijack all controllers
during boot, before manual intervention is possible. To make testing
and experimenting easier, the kernel parameter cgroup_no_v1= allows
disabling controllers in v1 and make them always available in v2.

cgroup v2 currently supports the following mount options.

  nsdelegate

        Consider cgroup namespaces as delegation boundaries. This
        option is system wide and can only be set on mount or modified
        through remount from the init namespace. The mount option is
        ignored on non-init namespace mounts. Please refer to the
        Delegation section for details.

Organizing Processes and Threads

Processes

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 forming 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. 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.

A process can be migrated into a cgroup by writing its PID to the
target cgroup's "cgroup.procs" file. Only one process can be migrated
on a single write(2) call. 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
```

"/proc/$PID/cgroup" lists a process's cgroup membership. If legacy
cgroup is in use in the system, this file may contain multiple lines,
one for each hierarchy. The entry for cgroup v2 is always in the
format "0::$PATH"::

```
  # cat /proc/842/cgroup
  ...
  0::/test-cgroup/test-cgroup-nested
```

If the process becomes a zombie and the cgroup it was associated with
is removed subsequently, " (deleted)" is appended to the path::

```
  # cat /proc/842/cgroup
  ...
  0::/test-cgroup/test-cgroup-nested (deleted)
```

Threads

cgroup v2 supports thread granularity for a subset of controllers to support use cases requiring hierarchical resource distribution across the threads of a group of processes. By default, all threads of a process belong to the same cgroup, which also serves as the resource domain to host resource consumptions which are not specific to a process or thread. The thread mode allows threads to be spread across a subtree while still maintaining the common resource domain for them.

Controllers which support thread mode are called threaded controllers. The ones which don't are called domain controllers.

Marking a cgroup threaded makes it join the resource domain of its parent as a threaded cgroup. The parent may be another threaded cgroup whose resource domain is further up in the hierarchy. The root of a threaded subtree, that is, the nearest ancestor which is not threaded, is called threaded domain or thread root interchangeably and serves as the resource domain for the entire subtree.

Inside a threaded subtree, threads of a process can be put in different cgroups and are not subject to the no internal process constraint - threaded controllers can be enabled on non-leaf cgroups whether they have threads in them or not.

As the threaded domain cgroup hosts all the domain resource consumptions of the subtree, it is considered to have internal resource consumptions whether there are processes in it or not and can't have populated child cgroups which aren't threaded. Because the root cgroup is not subject to no internal process constraint, it can serve both as a threaded domain and a parent to domain cgroups.

The current operation mode or type of the cgroup is shown in the "cgroup.type" file which indicates whether the cgroup is a normal domain, a domain which is serving as the domain of a threaded subtree, or a threaded cgroup.

On creation, a cgroup is always a domain cgroup and can be made threaded by writing "threaded" to the "cgroup.type" file. The operation is single direction::

```
  # echo threaded > cgroup.type
```

Once threaded, the cgroup can't be made a domain again. To enable the thread mode, the following conditions must be met.

- As the cgroup will join the parent's resource domain. The parent must either be a valid (threaded) domain or a threaded cgroup.

- When the parent is an unthreaded domain, it must not have any domain controllers enabled or populated domain children. The root is exempt from this requirement.

Topology-wise, a cgroup can be in an invalid state. Please consider the following topology::

```
  A (threaded domain) - B (threaded) - C (domain, just created)
```

C is created as a domain but isn't connected to a parent which can host child domains. C can't be used until it is turned into a threaded cgroup. "cgroup.type" file will report "domain (invalid)" in these cases. Operations which fail due to invalid topology use EOPNOTSUPP as the errno.

A domain cgroup is turned into a threaded domain when one of its child cgroup becomes threaded or threaded controllers are enabled in the "cgroup.subtree_control" file while there are processes in the cgroup. A threaded domain reverts to a normal domain when the conditions clear.

When read, "cgroup.threads" contains the list of the thread IDs of all threads in the cgroup. Except that the operations are per-thread instead of per-process, "cgroup.threads" has the same format and behaves the same way as "cgroup.procs". While "cgroup.threads" can be written to in any cgroup, as it can only move threads inside the same threaded domain, its operations are confined inside each threaded subtree.

The threaded domain cgroup serves as the resource domain for the whole subtree, and, while the threads can be scattered across the subtree, all the processes are considered to be in the threaded domain cgroup. "cgroup.procs" in a threaded domain cgroup contains the PIDs of all processes in the subtree and is not readable in the subtree proper. However, "cgroup.procs" can be written to from anywhere in the subtree to migrate all threads of the matching process to the cgroup.

Only threaded controllers can be enabled in a threaded subtree. When a threaded controller is enabled inside a threaded subtree, it only accounts for and controls resource consumptions associated with the threads in the cgroup and its descendants. All consumptions which aren't tied to a specific thread belong to the threaded domain cgroup.

Because a threaded subtree is exempt from no internal process constraint, a threaded controller must be able to handle competition between threads in a non-leaf cgroup and its child cgroups. Each threaded controller defines how such competitions are handled.

[Un]populated Notification

Each non-root cgroup has a "cgroup.events" file which contains "populated" field indicating whether the cgroup's sub-hierarchy has live processes in it. Its value is 0 if there is no live process in the cgroup and its descendants; otherwise, 1. poll and [id]notify events are triggered when the value changes. This can be used, for example, to start a clean-up operation after all processes of a given sub-hierarchy have exited. The populated state updates and notifications are recursive. 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.

Controlling Controllers

Enabling and Disabling

Each cgroup has a "cgroup.controllers" file which lists all
controllers available for the cgroup to enable::

```
  # cat cgroup.controllers
  cpu io memory
```

No controller is enabled by default. Controllers can be enabled and
disabled by writing to the "cgroup.subtree_control" file::

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

Only controllers which are listed in "cgroup.controllers" can be
enabled. 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.

Enabling a controller in a cgroup indicates that the distribution of
the target resource across its immediate children will be controlled.
Consider the following sub-hierarchy. The enabled controllers are
listed in parentheses::

```
  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.

As a controller regulates the distribution of the target resource to
the cgroup's children, enabling it creates the controller's interface
files in the child cgroups. In the above example, enabling "cpu" on B
would create the "cpu." prefixed controller interface files in C and
D. Likewise, disabling "memory" from B would remove the "memory."
prefixed controller interface files from C and D. This means that the
controller interface files - anything which doesn't start with
"cgroup." are owned by the parent rather than the cgroup itself.

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. 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.

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. In other words, only domain cgroups which don't contain any processes can have domain controllers enabled in their "cgroup.subtree_control" files.

This guarantees that, when a domain controller is looking at the part of the hierarchy which has it enabled, processes are always only on the leaves. 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. How resource consumption in the root cgroup is governed is up to each controller (for more information on this topic please refer to the Non-normative information section in the Controllers chapter).

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. To control resource distribution of a cgroup, the cgroup must create children and transfer all its processes to the children before enabling controllers in its "cgroup.subtree_control" file.

Delegation

Model of Delegation

A cgroup can be delegated in two ways. First, to a less privileged user by granting write access of the directory and its "cgroup.procs", "cgroup.threads" and "cgroup.subtree_control" files to the user. Second, if the "nsdelegate" mount option is set, automatically to a cgroup namespace on namespace creation.

Because the resource control interface files in a given directory control the distribution of the parent's resources, the delegatee shouldn't be allowed to write to them. For the first method, this is achieved by not granting access to these files. For the second, the kernel rejects writes to all files other than "cgroup.procs" and "cgroup.subtree_control" on a namespace root from inside the namespace.

The end results are equivalent for both delegation types. Once delegated, the user can build sub-hierarchy under the directory, organize processes inside it as it sees fit and further distribute the resources it received from the parent. The limits and other settings of all resource controllers are hierarchical and regardless of what

happens in the delegated sub-hierarchy, nothing can escape the
resource restrictions imposed by the parent.

Currently, cgroup doesn't impose any restrictions on the number of
cgroups in or nesting depth of a delegated sub-hierarchy; however,
this may be limited explicitly in the future.

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.

For delegations to a less privileged user, this is achieved by
requiring the following conditions for a process with a non-root euid
to migrate a target process into a cgroup by writing its PID to the
"cgroup.procs" file.

- The writer must have write access to the "cgroup.procs" file.

- The writer must have write access to the "cgroup.procs" file of the
  common ancestor of the source and destination cgroups.

The above two constraints ensure that while a delegatee may migrate
processes around freely in the delegated sub-hierarchy it can't pull
in from or push out to outside the sub-hierarchy.

For an example, let's assume cgroups C0 and C1 have been delegated to
user U0 who created C00, C01 under C0 and C10 under C1 as follows and
all processes under C0 and C1 belong to U0::

```
  ~~~~~~~~~~~~~ - C0 - C00
  ~ cgroup    ~      \ C01
  ~ hierarchy ~
  ~~~~~~~~~~~~~ - C1 - C10
```

Let's also say U0 wants to write the PID of a process which is
currently in C10 into "C00/cgroup.procs". U0 has write access to the
file; however, the common ancestor of the source cgroup C10 and the
destination cgroup C00 is above the points of delegation and U0 would
not have write access to its "cgroup.procs" files and thus the write
will be denied with -EACCES.

For delegations to namespaces, containment is achieved by requiring
that both the source and destination cgroups are reachable from the
namespace of the process which is attempting the migration. If either
is not reachable, the migration is rejected with -ENOENT.

Guidelines

Organize Once and Control

Migrating a process across cgroups is a relatively expensive operation
and stateful resources such as memory are not moved together with the

process. This is an explicit design decision as there often exist
inherent trade-offs between migration and various hot paths in terms
of synchronization cost.

As such, migrating processes across cgroups frequently as a means to
apply different resource restrictions is discouraged. A workload
should be assigned to a cgroup according to the system's logical and
resource structure once on start-up. Dynamic adjustments to resource
distribution can be made by changing controller configuration through
the interface files.

Avoid Name Collisions

Interface files for a cgroup and its children cgroups occupy the same
directory and it is possible to create children cgroups which collide
with interface files.

All cgroup core interface files are prefixed with "cgroup." and each
controller's interface files are prefixed with the controller name and
a dot. A controller's name is composed of lower case alphabets and
'_'s but never begins with an '_' so it can be used as the prefix
character for collision avoidance. Also, interface file names won't
start or end with terms which are often used in categorizing workloads
such as job, service, slice, unit or workload.

cgroup doesn't do anything to prevent name collisions and it's the
user's responsibility to avoid them.

Resource Distribution Models

cgroup controllers implement several resource distribution schemes
depending on the resource type and expected use cases. This section
describes major schemes in use along with their expected behaviors.

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. As only children which can make use of the
resource at the moment participate in the distribution, this is
work-conserving. Due to the dynamic nature, this model is usually
used for stateless resources.

All weights are in the range [1, 10000] with the default at 100. This
allows symmetric multiplicative biases in both directions at fine
enough granularity while staying in the intuitive range.

As long as the weight is in range, all configuration combinations are
valid and there is no reason to reject configuration changes or
process migrations.

"cpu.weight" proportionally distributes CPU cycles to active children and is an example of this type.

Limits

A child can only consume upto the configured amount of the resource.
Limits can be over-committed - the sum of the limits of children can
exceed the amount of resource available to the parent.

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.

"io.max" limits the maximum BPS and/or IOPS that a cgroup can consume
on an IO device and is an example of this type.

Protections

A cgroup is protected to be allocated upto the configured amount of
the resource if the usages of all its ancestors are under their
protected levels. Protections can be hard guarantees or best effort
soft boundaries. Protections can also be over-committed in which case
only upto the amount available to the parent is protected among
children.

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.

"memory.low" implements best-effort memory protection and is an
example of this type.

Allocations

A cgroup is exclusively allocated a certain amount of a finite
resource. Allocations can't be over-committed - the sum of the
allocations of children can not exceed the amount of resource
available to the parent.

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

As allocations can't be over-committed, some configuration
combinations are invalid and should be rejected. Also, if the
resource is mandatory for execution of processes, process migrations
may be rejected.

"cpu.rt.max" hard-allocates realtime slices and is an example of this type.

Interface Files

Format

All interface files should be in one of the following formats whenever possible::

```
  New-line separated values
  (when only one value can be written at once)

	VAL0\n
	VAL1\n
	...

  Space separated values
  (when read-only or multiple values can be written at once)

	VAL0 VAL1 ...\n

  Flat keyed

	KEY0 VAL0\n
	KEY1 VAL1\n
	...

  Nested keyed

	KEY0 SUB_KEY0=VAL00 SUB_KEY1=VAL01...
	KEY1 SUB_KEY0=VAL10 SUB_KEY1=VAL11...
	...
```

For a writable file, the format for writing should generally match reading; however, controllers may allow omitting later fields or implement restricted shortcuts for most common use cases.

For both flat and nested keyed files, only the values for a single key can be written at a time. For nested keyed files, the sub key pairs may be specified in any order and not all pairs have to be specified.

Conventions

- Settings for a single feature should be contained in a single file.

- The root cgroup should be exempt from resource control and thus shouldn't have resource control interface files. Also, informational files on the root cgroup which end up showing global information available elsewhere shouldn't exist.

- If a controller implements weight based resource distribution, its interface file should be named "weight" and have the range [1, 10000] with 100 as the default. The values are chosen to allow

```
  enough and symmetric bias in both directions while keeping it
  intuitive (the default is 100%).

- If a controller implements an absolute resource guarantee and/or
  limit, the interface files should be named "min" and "max"
  respectively.  If a controller implements best effort resource
  guarantee and/or limit, the interface files should be named "low"
  and "high" respectively.

  In the above four control files, the special token "max" should be
  used to represent upward infinity for both reading and writing.

- If a setting has a configurable default value and keyed specific
  overrides, the default entry should be keyed with "default" and
  appear as the first entry in the file.

  The default value can be updated by writing either "default $VAL" or
  "$VAL".

  When writing to update a specific override, "default" can be used as
  the value to indicate removal of the override.  Override entries
  with "default" as the value must not appear when read.

  For example, a setting which is keyed by major:minor device numbers
  with integer values may look like the following::

    # cat cgroup-example-interface-file
    default 150
    8:0 300

  The default value can be updated by::

    # echo 125 > cgroup-example-interface-file

  or::

    # echo "default 125" > cgroup-example-interface-file

  An override can be set by::

    # echo "8:16 170" > cgroup-example-interface-file

  and cleared by::

    # echo "8:0 default" > cgroup-example-interface-file
    # cat cgroup-example-interface-file
    default 125
    8:16 170

- For events which are not very high frequency, an interface file
  "events" should be created which lists event key value pairs.
  Whenever a notifiable event happens, file modified event should be
  generated on the file.


Core Interface Files
--------------------
```

```
All cgroup core files are prefixed with "cgroup."

  cgroup.type

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

	When read, it indicates the current type of the cgroup, which
	can be one of the following values.

	- "domain" : A normal valid domain cgroup.

	- "domain threaded" : A threaded domain cgroup which is
	  serving as the root of a threaded subtree.

	- "domain invalid" : A cgroup which is in an invalid state.
	  It can't be populated or have controllers enabled.  It may
	  be allowed to become a threaded cgroup.

	- "threaded" : A threaded cgroup which is a member of a
	  threaded subtree.

	A cgroup can be turned into a threaded cgroup by writing
	"threaded" to this file.

  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.

	A PID can be written to migrate the process associated with
	the PID to the cgroup.  The writer should match all of the
	following conditions.

	- It must have write access to the "cgroup.procs" file.

	- It must have write access to the "cgroup.procs" file of the
	  common ancestor of the source and destination cgroups.

	When delegating a sub-hierarchy, write access to this file
	should be granted along with the containing directory.

	In a threaded cgroup, reading this file fails with EOPNOTSUPP
	as all the processes belong to the thread root.  Writing is
	supported and moves every thread of the process to the cgroup.

  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.

	A TID can be written to migrate the thread associated with the
	TID to the cgroup.  The writer should match all of the
	following conditions.

	- It must have write access to the "cgroup.threads" file.

	- The cgroup that the thread is currently in must be in the
	  same resource domain as the destination cgroup.

	- It must have write access to the "cgroup.procs" file of the
	  common ancestor of the source and destination cgroups.

	When delegating a sub-hierarchy, write access to this file
	should be granted along with the containing directory.

  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.

  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.

  cgroup.events
	A read-only flat-keyed file which exists on non-root cgroups.
	The following entries are defined.  Unless specified
	otherwise, a value change in this file generates a file
	modified event.

	  populated
		1 if the cgroup or its descendants contains any live
		processes; otherwise, 0.

  cgroup.max.descendants
	A read-write single value files.  The default is "max".

	Maximum allowed number of descent cgroups.
	If the actual number of descendants is equal or larger,
	an attempt to create a new cgroup in the hierarchy will fail.

  cgroup.max.depth
```

```
        A read-write single value files.  The default is "max".

        Maximum allowed descent depth below the current cgroup.
        If the actual descent depth is equal or larger,
        an attempt to create a new child cgroup will fail.

  cgroup.stat
        A read-only flat-keyed file with the following entries:

          nr_descendants
                Total number of visible descendant cgroups.

          nr_dying_descendants
                Total number of dying descendant cgroups. A cgroup becomes
                dying after being deleted by a user. The cgroup will remain
                in dying state for some time undefined time (which can depend
                on system load) before being completely destroyed.

                A process can't enter a dying cgroup under any circumstances,
                a dying cgroup can't revive.

                A dying cgroup can consume system resources not exceeding
                limits, which were active at the moment of cgroup deletion.


Controllers
===========

CPU
---

The "cpu" controllers regulates distribution of CPU cycles.  This
controller implements weight and absolute bandwidth limit models for
normal scheduling policy and absolute bandwidth allocation model for
realtime scheduling policy.

WARNING: cgroup2 doesn't yet support control of realtime processes and
the cpu controller can only be enabled when all RT processes are in
the root cgroup.  Be aware that system management software may already
have placed RT processes into nonroot cgroups during the system boot
process, and these processes may need to be moved to the root cgroup
before the cpu controller can be enabled.


CPU Interface Files
~~~~~~~~~~~~~~~~~~~

All time durations are in microseconds.

  cpu.stat
        A read-only flat-keyed file which exists on non-root cgroups.
        This file exists whether the controller is enabled or not.

        It always reports the following three stats:

        - usage_usec
        - user_usec
        - system_usec
```

```
        and the following three when the controller is enabled:

        - nr_periods
        - nr_throttled
        - throttled_usec

  cpu.weight
        A read-write single value file which exists on non-root
        cgroups.  The default is "100".

        The weight in the range [1, 10000].

  cpu.weight.nice
        A read-write single value file which exists on non-root
        cgroups.  The default is "0".

        The nice value is in the range [-20, 19].

        This interface file is an alternative interface for
        "cpu.weight" and allows reading and setting weight using the
        same values used by nice(2).  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.
        The default is "max 100000".

        The maximum bandwidth limit.  It's in the following format::

          $MAX $PERIOD

        which indicates that the group may consume upto $MAX in each
        $PERIOD duration.  "max" for $MAX indicates no limit.  If only
        one number is written, $MAX is updated.
```

Memory

The "memory" controller regulates distribution of memory. Memory is stateful and implements both limit and protection models. Due to the intertwining between memory usage and reclaim pressure and the stateful nature of memory, the distribution model is relatively complex.

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 above list may expand in the future for better coverage.


Memory Interface Files
~~~~~~~~~~~~~~~~~~~~~~

All memory amounts are in bytes.  If a value which is not aligned to
PAGE_SIZE is written, the value may be rounded up to the closest
PAGE_SIZE multiple when read back.

  memory.current
	A read-only single value file which exists on non-root
	cgroups.

	The total amount of memory currently being used by the cgroup
	and its descendants.

  memory.low
	A read-write single value file which exists on non-root
	cgroups.  The default is "0".

	Best-effort memory protection.  If the memory usages of a
	cgroup and all its ancestors are below their low boundaries,
	the cgroup's memory won't be reclaimed unless memory can be
	reclaimed from unprotected cgroups.

	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.  The default is "max".

	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.

  memory.max
	A read-write single value file which exists on non-root
	cgroups.  The default is "max".

	Memory usage hard limit.  This is the final protection
	mechanism.  If a cgroup's memory usage reaches this limit and
	can't be reduced, the OOM killer is invoked in the cgroup.
	Under certain circumstances, the usage may go over the limit
	temporarily.

	This is the ultimate protection mechanism.  As long as the
	high limit is used and monitored properly, this limit's
	utility is limited to providing the final safety net.

  memory.events
	A read-only flat-keyed file which exists on non-root cgroups.
	The following entries are defined.  Unless specified
```

```
	otherwise, a value change in this file generates a file
	modified event.

	  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
		reached the limit and allocation was about to fail.

		Depending on context result could be invocation of OOM
		killer and retrying allocation or failing allocation.

		Failed allocation in its turn could be returned into
		userspace as -ENOMEM or silently ignored in cases like
		disk readahead.  For now OOM in memory cgroup kills
		tasks iff shortage has happened inside page fault.

	  oom_kill
		The number of processes belonging to this cgroup
		killed by any kind of OOM killer.

  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.

	All memory amounts are in bytes.

	The entries are ordered to be human readable, and new entries
	can show up in the middle. Don't rely on items remaining in a
	fixed position; use the keys to look up specific values!

	  anon
		Amount of memory used in anonymous mappings such as
		brk(), sbrk(), and mmap(MAP_ANONYMOUS)

	  file
		Amount of memory used to cache filesystem data,
		including tmpfs and shared memory.
```

```
	  kernel_stack
		Amount of memory allocated to kernel stacks.

	  slab
		Amount of memory used for storing in-kernel data
		structures.

	  sock
		Amount of memory used in network transmission buffers

	  shmem
		Amount of cached filesystem data that is swap-backed,
		such as tmpfs, shm segments, shared anonymous mmap()s

	  file_mapped
		Amount of cached filesystem data mapped with mmap()

	  file_dirty
		Amount of cached filesystem data that was modified but
		not yet written back to disk

	  file_writeback
		Amount of cached filesystem data that was modified and
		is currently being written back to disk

	  inactive_anon, active_anon, inactive_file, active_file, unevictable
		Amount of memory, swap-backed and filesystem-backed,
		on the internal memory management lists used by the
		page reclaim algorithm

	  slab_reclaimable
		Part of "slab" that might be reclaimed, such as
		dentries and inodes.

	  slab_unreclaimable
		Part of "slab" that cannot be reclaimed on memory
		pressure.

	  pgfault
		Total number of page faults incurred

	  pgmajfault
		Number of major page faults incurred

	  workingset_refault

		Number of refaults of previously evicted pages

	  workingset_activate

		Number of refaulted pages that were immediately activated

	  workingset_nodereclaim

		Number of times a shadow node has been reclaimed

	  pgrefill
```

Amount of scanned pages (in an active LRU list)

pgscan

Amount of scanned pages (in an inactive LRU list)

pgsteal

Amount of reclaimed pages

pgactivate

Amount of pages moved to the active LRU list

pgdeactivate

Amount of pages moved to the inactive LRU lis

pglazyfree

Amount of pages postponed to be freed under memory pressure

pglazyfreed

Amount of reclaimed lazyfree pages

memory.swap.current

A read-only single value file which exists on non-root cgroups.

The total amount of swap currently being used by the cgroup and its descendants.

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.

Usage Guidelines

"memory.high" is the main mechanism to control memory usage. Over-committing on high limit (sum of high limits > available memory) and letting global memory pressure to distribute memory according to usage is a viable strategy.

Because breach of the high limit doesn't trigger the OOM killer but throttles the offending cgroup, a management agent has ample opportunities to monitor and take appropriate actions such as granting more memory or terminating the workload.

Determining whether a cgroup has enough memory is not trivial as memory usage doesn't indicate whether the workload can benefit from more memory. For example, a workload which writes data received from network to a file can use all available memory but can also operate as performant with a small amount of memory. A measure of memory pressure - how much the workload is being impacted due to lack of memory - is necessary to determine whether a workload needs more memory; unfortunately, memory pressure monitoring mechanism isn't implemented yet.

Memory Ownership

A memory area is charged to the cgroup which instantiated it and stays charged to the cgroup until the area is released. Migrating a process to a different cgroup doesn't move the memory usages that it instantiated while in the previous cgroup to the new cgroup.

A memory area may be used by processes belonging to different cgroups. To which cgroup the area will be charged is in-deterministic; however, over time, the memory area is likely to end up in a cgroup which has enough memory allowance to avoid high reclaim pressure.

If a cgroup sweeps a considerable amount of memory which is expected to be accessed repeatedly by other cgroups, it may make sense to use POSIX_FADV_DONTNEED to relinquish the ownership of memory areas belonging to the affected files to ensure correct memory ownership.

IO

The "io" controller regulates the distribution of IO resources. This controller implements both weight based and absolute bandwidth or IOPS limit distribution; however, weight based distribution is available only if cfq-iosched is in use and neither scheme is available for blk-mq devices.

IO Interface Files

io.stat

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

Lines are keyed by $MAJ:$MIN device numbers and not ordered. The following nested keys are defined.

rbytes	Bytes read
wbytes	Bytes written
rios	Number of read IOs
wios	Number of write IOs

An example read output follows:

```
8:16 rbytes=1459200 wbytes=314773504 rios=192 wios=353
8:0 rbytes=90430464 wbytes=299008000 rios=8950 wios=1252
```

```
  io.weight
	A read-write flat-keyed file which exists on non-root cgroups.
	The default is "default 100".

	The first line is the default weight applied to devices
	without specific override.  The rest are overrides keyed by
	$MAJ:$MIN device numbers and not ordered.  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 weight can be updated by writing either "default
	$WEIGHT" or simply "$WEIGHT".  Overrides can be set by writing
	"$MAJ:$MIN $WEIGHT" and unset by writing "$MAJ:$MIN default".

	An example read output follows::

	  default 100
	  8:16 200
	  8:0 50

  io.max
	A read-write nested-keyed file which exists on non-root
	cgroups.

	BPS and IOPS based IO limit.  Lines are keyed by $MAJ:$MIN
	device numbers and not ordered.  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.

	BPS and IOPS are measured in each IO direction and IOs are
	delayed if limit is reached.  Temporary bursts are allowed.

	Setting read limit at 2M BPS and write at 120 IOPS for 8:16::

	  echo "8:16 rbps=2097152 wiops=120" > io.max

	Reading returns the following::

	  8:16 rbps=2097152 wbps=max riops=max wiops=120

	Write IOPS limit can be removed by writing the following::

	  echo "8:16 wiops=max" > io.max

	Reading now returns the following::
```

```
          8:16 rbps=2097152 wbps=max riops=max wiops=max
```

Writeback
~~~~~~~~~

Page cache is dirtied through buffered writes and shared mmaps and written asynchronously to the backing filesystem by the writeback mechanism. Writeback sits between the memory and IO domains and regulates the proportion of dirty memory by balancing dirtying and write IOs.

The io controller, in conjunction with the memory controller, implements control of page cache writeback IOs. The memory controller defines the memory domain that dirty memory ratio is calculated and maintained for and the io controller defines the io domain which writes out dirty pages for the memory domain. Both system-wide and per-cgroup dirty memory states are examined and the more restrictive of the two is enforced.

cgroup writeback requires explicit support from the underlying filesystem. Currently, cgroup writeback is implemented on ext2, ext4 and btrfs. On other filesystems, all writeback IOs are attributed to the root cgroup.

There are inherent differences in memory and writeback management which affects how cgroup ownership is tracked. Memory is tracked per page while writeback per inode. For the purpose of writeback, an inode is assigned to a cgroup and all IO requests to write dirty pages from the inode are attributed to that cgroup.

As cgroup ownership for memory is tracked per page, there can be pages which are associated with different cgroups than the one the inode is associated with. These are called foreign pages. The writeback constantly keeps track of foreign pages and, if a particular foreign cgroup becomes the majority over a certain period of time, switches the ownership of the inode to that cgroup.

While this model is enough for most use cases where a given inode is mostly dirtied by a single cgroup even when the main writing cgroup changes over time, use cases where multiple cgroups write to a single inode simultaneously are not supported well. In such circumstances, a significant portion of IOs are likely to be attributed incorrectly. As memory controller assigns page ownership on the first use and doesn't update it until the page is released, even if writeback strictly follows page ownership, multiple cgroups dirtying overlapping areas wouldn't work as expected. It's recommended to avoid such usage patterns.

The sysctl knobs which affect writeback behavior are applied to cgroup writeback as follows.

  vm.dirty_background_ratio, vm.dirty_ratio
	These ratios apply the same to cgroup writeback with the amount of available memory capped by limits imposed by the memory controller and system-wide clean memory.

  vm.dirty_background_bytes, vm.dirty_bytes
~~~~~~~~~

```
        For cgroup writeback, this is calculated into ratio against
        total available memory and applied the same way as
        vm.dirty[_background]_ratio.


PID
---

The process number controller is used to allow a cgroup to stop any
new tasks from being fork()'d or clone()'d after a specified limit is
reached.

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.

Note that PIDs used in this controller refer to TIDs, process IDs as
used by the kernel.


PID Interface Files
~~~~~~~~~~~~~~~~~~~

  pids.max
        A read-write single value file which exists on non-root
        cgroups.  The default is "max".

        Hard limit of number of processes.

  pids.current
        A read-only single value file which exists on all cgroups.

        The number of processes currently in the cgroup and its
        descendants.

Organisational 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.


Device controller
-----------------

Device controller manages access to device files. It includes both
creation of new device files (using mknod), and access to the
existing device files.

Cgroup v2 device controller has no interface files and is implemented
on top of cgroup BPF. To control access to device files, a user may
create bpf programs of the BPF_CGROUP_DEVICE type and attach them
to cgroups. On an attempt to access a device file, corresponding
BPF programs will be executed, and depending on the return value
the attempt will succeed or fail with -EPERM.
```

```
A BPF_CGROUP_DEVICE program takes a pointer to the bpf_cgroup_dev_ctx
structure, which describes the device access attempt: access type
(mknod/read/write) and device (type, major and minor numbers).
If the program returns 0, the attempt fails with -EPERM, otherwise
it succeeds.

An example of BPF_CGROUP_DEVICE program may be found in the kernel
source tree in the tools/testing/selftests/bpf/dev_cgroup.c file.


RDMA
----

The "rdma" controller regulates the distribution and accounting of
of RDMA resources.

RDMA Interface Files
~~~~~~~~~~~~~~~~~~~~

  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
          ==========    =============================

        An example for mlx4 and ocrdma device follows::

          mlx4_0 hca_handle=2 hca_object=2000
          ocrdma1 hca_handle=3 hca_object=max

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

        An example for mlx4 and ocrdma device follows::

          mlx4_0 hca_handle=1 hca_object=20
          ocrdma1 hca_handle=1 hca_object=23


Misc
----

perf_event
~~~~~~~~~~

perf_event controller, if not mounted on a legacy hierarchy, is
```

automatically enabled on the v2 hierarchy so that perf events can always be filtered by cgroup v2 path. The controller can still be moved to a legacy hierarchy after v2 hierarchy is populated.

Non-normative information

This section contains information that isn't considered to be a part of the stable kernel API and so is subject to change.

CPU controller root cgroup process behaviour

When distributing CPU cycles in the root cgroup each thread in this cgroup is treated as if it was hosted in a separate child cgroup of the root cgroup. This child cgroup weight is dependent on its thread nice level.

For details of this mapping see sched_prio_to_weight array in kernel/sched/core.c file (values from this array should be scaled appropriately so the neutral - nice 0 - value is 100 instead of 1024).

IO controller root cgroup process behaviour

Root cgroup processes are hosted in an implicit leaf child node. When distributing IO resources this implicit child node is taken into account as if it was a normal child cgroup of the root cgroup with a weight value of 200.

Namespace

Basics

cgroup namespace provides a mechanism to virtualize the view of the "/proc/$PID/cgroup" file and cgroup mounts. The CLONE_NEWCGROUP clone flag can be used with clone(2) and unshare(2) to create a new cgroup namespace. The process running inside the cgroup namespace will have its "/proc/$PID/cgroup" output restricted to cgroupns root. The cgroupns root is the cgroup of the process at the time of creation of the cgroup namespace.

Without cgroup namespace, the "/proc/$PID/cgroup" file shows the complete path of the cgroup of a process. In a container setup where a set of cgroups and namespaces are intended to isolate processes the "/proc/$PID/cgroup" file may leak potential system level information to the isolated processes. For Example::

```
  # cat /proc/self/cgroup
  0::/batchjobs/container_id1
```

The path '/batchjobs/container_id1' can be considered as system-data

and undesirable to expose to the isolated processes. cgroup namespace
can be used to restrict visibility of this path. For example, before
creating a cgroup namespace, one would see::

```
  # ls -l /proc/self/ns/cgroup
  lrwxrwxrwx 1 root root 0 2014-07-15 10:37 /proc/self/ns/cgroup -> cgroup:
[4026531835]
  # cat /proc/self/cgroup
  0::/batchjobs/container_id1
```

After unsharing a new namespace, the view changes::

```
  # ls -l /proc/self/ns/cgroup
  lrwxrwxrwx 1 root root 0 2014-07-15 10:35 /proc/self/ns/cgroup -> cgroup:
[4026532183]
  # cat /proc/self/cgroup
  0::/
```

When some thread from a multi-threaded process unshares its cgroup
namespace, the new cgroupns gets applied to the entire process (all
the threads). This is natural for the v2 hierarchy; however, for the
legacy hierarchies, this may be unexpected.

A cgroup namespace is alive as long as there are processes inside or
mounts pinning it. When the last usage goes away, the cgroup
namespace is destroyed. The cgroupns root and the actual cgroups
remain.

The Root and Views

The 'cgroupns root' for a cgroup namespace is the cgroup in which the
process calling unshare(2) is running. For example, if a process in
/batchjobs/container_id1 cgroup calls unshare, cgroup
/batchjobs/container_id1 becomes the cgroupns root. For the
init_cgroup_ns, this is the real root ('/') cgroup.

The cgroupns root cgroup does not change even if the namespace creator
process later moves to a different cgroup::

```
  # ~/unshare -c # unshare cgroupns in some cgroup
  # cat /proc/self/cgroup
  0::/
  # mkdir sub_cgrp_1
  # echo 0 > sub_cgrp_1/cgroup.procs
  # cat /proc/self/cgroup
  0::/sub_cgrp_1
```

Each process gets its namespace-specific view of "/proc/$PID/cgroup"

Processes running inside the cgroup namespace will be able to see
cgroup paths (in /proc/self/cgroup) only inside their root cgroup.
From within an unshared cgroupns::

```
  # sleep 100000 &
  [1] 7353
  # echo 7353 > sub_cgrp_1/cgroup.procs
  # cat /proc/7353/cgroup
  0::/sub_cgrp_1
```

From the initial cgroup namespace, the real cgroup path will be visible::

```
  $ cat /proc/7353/cgroup
  0::/batchjobs/container_id1/sub_cgrp_1
```

From a sibling cgroup namespace (that is, a namespace rooted at a different cgroup), the cgroup path relative to its own cgroup namespace root will be shown. For instance, if PID 7353's cgroup namespace root is at '/batchjobs/container_id2', then it will see::

```
  # cat /proc/7353/cgroup
  0::/../container_id2/sub_cgrp_1
```

Note that the relative path always starts with '/' to indicate that its relative to the cgroup namespace root of the caller.

Migration and setns(2)

Processes inside a cgroup namespace can move into and out of the namespace root if they have proper access to external cgroups. For example, from inside a namespace with cgroupns root at /batchjobs/container_id1, and assuming that the global hierarchy is still accessible inside cgroupns::

```
  # cat /proc/7353/cgroup
  0::/sub_cgrp_1
  # echo 7353 > batchjobs/container_id2/cgroup.procs
  # cat /proc/7353/cgroup
  0::/../container_id2
```

Note that this kind of setup is not encouraged. A task inside cgroup namespace should only be exposed to its own cgroupns hierarchy.

setns(2) to another cgroup namespace is allowed when:

(a) the process has CAP_SYS_ADMIN against its current user namespace
(b) the process has CAP_SYS_ADMIN against the target cgroup
    namespace's userns

No implicit cgroup changes happen with attaching to another cgroup namespace. It is expected that the someone moves the attaching process under the target cgroup namespace root.

Interaction with Other Namespaces

Namespace specific cgroup hierarchy can be mounted by a process running inside a non-init cgroup namespace::

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

This will mount the unified cgroup hierarchy with cgroupns root as the filesystem root. The process needs CAP_SYS_ADMIN against its user and mount namespaces.

The virtualization of /proc/self/cgroup file combined with restricting the view of cgroup hierarchy by namespace-private cgroupfs mount provides a properly isolated cgroup view inside the container.

Information on Kernel Programming

This section contains kernel programming information in the areas where interacting with cgroup is necessary. cgroup core and controllers are not covered.

Filesystem Support for Writeback

A filesystem can support cgroup writeback by updating address_space_operations->writepage[s]() to annotate bio's using the following two functions.

```
  wbc_init_bio(@wbc, @bio)
        Should be called for each bio carrying writeback data and
        associates the bio with the inode's owner cgroup.  Can be
        called anytime between bio allocation and submission.

  wbc_account_io(@wbc, @page, @bytes)
        Should be called for each data segment being written out.
        While this function doesn't care exactly when it's called
        during the writeback session, it's the easiest and most
        natural to call it as data segments are added to a bio.
```

With writeback bio's annotated, cgroup support can be enabled per super_block by setting SB_I_CGROUPWB in ->s_iflags. This allows for selective disabling of cgroup writeback support which is helpful when certain filesystem features, e.g. journaled data mode, are incompatible.

wbc_init_bio() binds the specified bio to its cgroup. Depending on the configuration, the bio may be executed at a lower priority and if the writeback session is holding shared resources, e.g. a journal entry, may lead to priority inversion. There is no one easy solution for the problem. Filesystems can try to work around specific problem cases by skipping wbc_init_bio() or using bio_associate_blkcg() directly.

Deprecated v1 Core Features

- Multiple hierarchies including named ones are not supported.

- All v1 mount options are not supported.

- The "tasks" file is removed and "cgroup.procs" is not sorted.

- "cgroup.clone_children" is removed.

- /proc/cgroups is meaningless for v2. Use "cgroup.controllers" file
  at the root instead.

Issues with v1 and Rationales for v2

Multiple Hierarchies

cgroup v1 allowed an arbitrary number of hierarchies and each hierarchy could host any number of controllers. While this seemed to provide a high level of flexibility, it wasn't useful in practice.

For example, as there is only one instance of each controller, utility type controllers such as freezer which can be useful in all hierarchies could only be used in one. The issue is exacerbated by the fact that controllers couldn't be moved to another hierarchy once hierarchies were populated. Another issue was that all controllers bound to a hierarchy were forced to have exactly the same view of the hierarchy. It wasn't possible to vary the granularity depending on the specific controller.

In practice, these issues heavily limited which controllers could be put on the same hierarchy and most configurations resorted to putting each controller on its own hierarchy. Only closely related ones, such as the cpu and cpuacct controllers, made sense to be put on the same hierarchy. This often meant that userland ended up managing multiple similar hierarchies repeating the same steps on each hierarchy whenever a hierarchy management operation was necessary.

Furthermore, support for multiple hierarchies came at a steep cost. It greatly complicated cgroup core implementation but more importantly the support for multiple hierarchies restricted how cgroup could be used in general and what controllers was able to do.

There was no limit on how many hierarchies there might be, which meant that a thread's cgroup membership couldn't be described in finite length. The key might contain any number of entries and was unlimited in length, which made it highly awkward to manipulate and led to addition of controllers which existed only to identify membership, which in turn exacerbated the original problem of proliferating number of hierarchies.

Also, as a controller couldn't have any expectation regarding the topologies of hierarchies other controllers might be on, each controller had to assume that all other controllers were attached to completely orthogonal hierarchies. This made it impossible, or at least very cumbersome, for controllers to cooperate with each other.

In most use cases, putting controllers on hierarchies which are completely orthogonal to each other isn't necessary. What usually is called for is the ability to have differing levels of granularity depending on the specific controller. In other words, hierarchy may be collapsed from leaf towards root when viewed from specific controllers. For example, a given configuration might not care about how memory is distributed beyond a certain level while still wanting to control how CPU cycles are distributed.

Thread Granularity

cgroup v1 allowed threads of a process to belong to different cgroups. This didn't make sense for some controllers and those controllers ended up implementing different ways to ignore such situations but much more importantly it blurred the line between API exposed to individual applications and system management interface.

Generally, in-process knowledge is available only to the process itself; thus, unlike service-level organization of processes, categorizing threads of a process requires active participation from the application which owns the target process.

cgroup v1 had an ambiguously defined delegation model which got abused in combination with thread granularity. cgroups were delegated to individual applications so that they can create and manage their own sub-hierarchies and control resource distributions along them. This effectively raised cgroup to the status of a syscall-like API exposed to lay programs.

First of all, cgroup has a fundamentally inadequate interface to be exposed this way. For a process to access its own knobs, it has to extract the path on the target hierarchy from /proc/self/cgroup, construct the path by appending the name of the knob to the path, open and then read and/or write to it. This is not only extremely clunky and unusual but also inherently racy. There is no conventional way to define transaction across the required steps and nothing can guarantee that the process would actually be operating on its own sub-hierarchy.

cgroup controllers implemented a number of knobs which would never be accepted as public APIs because they were just adding control knobs to system-management pseudo filesystem. cgroup ended up with interface knobs which were not properly abstracted or refined and directly revealed kernel internal details. These knobs got exposed to individual applications through the ill-defined delegation mechanism effectively abusing cgroup as a shortcut to implementing public APIs without going through the required scrutiny.

This was painful for both userland and kernel. Userland ended up with misbehaving and poorly abstracted interfaces and kernel exposing and locked into constructs inadvertently.

Competition Between Inner Nodes and Threads

cgroup v1 allowed threads to be in any cgroups which created an interesting problem where threads belonging to a parent cgroup and its children cgroups competed for resources. This was nasty as two different types of entities competed and there was no obvious way to settle it. Different controllers did different things.

The cpu controller considered threads and cgroups as equivalents and mapped nice levels to cgroup weights. This worked for some cases but fell flat when children wanted to be allocated specific ratios of CPU cycles and the number of internal threads fluctuated - the ratios constantly changed as the number of competing entities fluctuated. There also were other issues. The mapping from nice level to weight wasn't obvious or universal, and there were various other knobs which simply weren't available for threads.

The io controller implicitly created a hidden leaf node for each cgroup to host the threads. The hidden leaf had its own copies of all the knobs with ``leaf_`` prefixed. While this allowed equivalent control over internal threads, it was with serious drawbacks. It always added an extra layer of nesting which wouldn't be necessary otherwise, made the interface messy and significantly complicated the implementation.

The memory controller didn't have a way to control what happened between internal tasks and child cgroups and the behavior was not clearly defined. There were attempts to add ad-hoc behaviors and knobs to tailor the behavior to specific workloads which would have led to problems extremely difficult to resolve in the long term.

Multiple controllers struggled with internal tasks and came up with different ways to deal with it; unfortunately, all the approaches were severely flawed and, furthermore, the widely different behaviors made cgroup as a whole highly inconsistent.

This clearly is a problem which needs to be addressed from cgroup core in a uniform way.

Other Interface Issues

cgroup v1 grew without oversight and developed a large number of idiosyncrasies and inconsistencies. One issue on the cgroup core side was how an empty cgroup was notified - a userland helper binary was forked and executed for each event. The event delivery wasn't recursive or delegatable. The limitations of the mechanism also led to in-kernel event delivery filtering mechanism further complicating the interface.

Controller interfaces were problematic too. An extreme example is controllers completely ignoring hierarchical organization and treating all cgroups as if they were all located directly under the root cgroup. Some controllers exposed a large amount of inconsistent implementation details to userland.

There also was no consistency across controllers. When a new cgroup was created, some controllers defaulted to not imposing extra restrictions while others disallowed any resource usage until explicitly configured. Configuration knobs for the same type of control used widely differing naming schemes and formats. Statistics and information knobs were named arbitrarily and used different formats and units even in the same controller.

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

Controller Issues and Remedies

Memory

The original lower boundary, the soft limit, is defined as a limit that is per default unset. As a result, the set of cgroups that global reclaim prefers is opt-in, rather than opt-out. The costs for optimizing these mostly negative lookups are so high that the implementation, despite its enormous size, does not even provide the basic desirable behavior. First off, the soft limit has no hierarchical meaning. All configured groups are organized in a global rbtree and treated like equal peers, regardless where they are located in the hierarchy. This makes subtree delegation impossible. Second, the soft limit reclaim pass is so aggressive that it not just introduces high allocation latencies into the system, but also impacts system performance due to overreclaim, to the point where the feature becomes self-defeating.

The memory.low boundary on the other hand is a top-down allocated reserve. A cgroup enjoys reclaim protection when it and all its ancestors are below their low boundaries, which makes delegation of subtrees possible. Secondly, new cgroups have no reserve per default and in the common case most cgroups are eligible for the preferred reclaim pass. This allows the new low boundary to be efficiently implemented with just a minor addition to the generic reclaim code, without the need for out-of-band data structures and reclaim passes. Because the generic reclaim code considers all cgroups except for the ones running low in the preferred first reclaim pass, overreclaim of individual groups is eliminated as well, resulting in much better overall workload performance.

The original high boundary, the hard limit, is defined as a strict limit that can not budge, even if the OOM killer has to be called. But this generally goes against the goal of making the most out of the available memory. The memory consumption of workloads varies during runtime, and that requires users to overcommit. But doing that with a strict upper limit requires either a fairly accurate prediction of the working set size or adding slack to the limit. Since working set size estimation is hard and error prone, and getting it wrong results in OOM kills, most users tend to err on the side of a looser limit and end up wasting precious resources.

The memory.high boundary on the other hand can be set much more conservatively. When hit, it throttles allocations by forcing them into direct reclaim to work off the excess, but it never invokes the OOM killer. As a result, a high boundary that is chosen too aggressively will not terminate the processes, but instead it will lead to gradual performance degradation. The user can monitor this and make corrections until the minimal memory footprint that still gives acceptable performance is found.

In extreme cases, with many concurrent allocations and a complete breakdown of reclaim progress within the group, the high boundary can

be exceeded. But even then it's mostly better to satisfy the
allocation from the slack available in other groups or the rest of the
system than killing the group. Otherwise, memory.max is there to
limit this type of spillover and ultimately contain buggy or even
malicious applications.

Setting the original memory.limit_in_bytes below the current usage was
subject to a race condition, where concurrent charges could cause the
limit setting to fail. memory.max on the other hand will first set the
limit to prevent new charges, and then reclaim and OOM kill until the
new limit is met - or the task writing to memory.max is killed.

The combined memory+swap accounting and limiting is replaced by real
control over swap space.

The main argument for a combined memory+swap facility in the original
cgroup design was that global or parental pressure would always be
able to swap all anonymous memory of a child group, regardless of the
child's own (possibly untrusted) configuration. However, untrusted
groups can sabotage swapping by other means - such as referencing its
anonymous memory in a tight loop - and an admin can not assume full
swappability when overcommitting untrusted jobs.

For trusted jobs, on the other hand, a combined counter is not an
intuitive userspace interface, and it flies in the face of the idea
that cgroup controllers should account and limit specific physical
resources. Swap space is a resource like all others in the system,
and that's why unified hierarchy allows distributing it separately.