Open Source Summit Europe 2018

What's new in control groups (cgroups) v2

Michael Kerrisk, man7.org © 2018
mtk@man7.org

Open Source Summit Europe
21 October 2018, Edinburgh, Scotland

Outline

1 Introduction 3
2 Problems with cgroups v1; rationale for v2 4
3 Cgroups v2 controllers 16
4 Enabling and disabling controllers 20
5 Organizing cgroups and processes 33
6 Release notification (cgroup.events file) 38
7 Optional topic: delegation 41
8 Optional topic: overview of thread mode 47

Outline

1 Introduction 3
2 Problems with cgroups v1; rationale for v2 4
3 Cgroups v2 controllers 16
4 Enabling and disabling controllers 20
5 Organizing cgroups and processes 33
6 Release notification (cgroup.events file) 38
7 Optional topic: delegation 41
8 Optional topic: overview of thread mode 47

Outline

1 Introduction 3
2 Problems with cgroups v1; rationale for v2 4
3 Cgroups v2 controllers 16
4 Enabling and disabling controllers 20
5 Organizing cgroups and processes 33
6 Release notification (cgroup.events file) 38
7 Optional topic: delegation 41
8 Optional topic: overview of thread mode 47

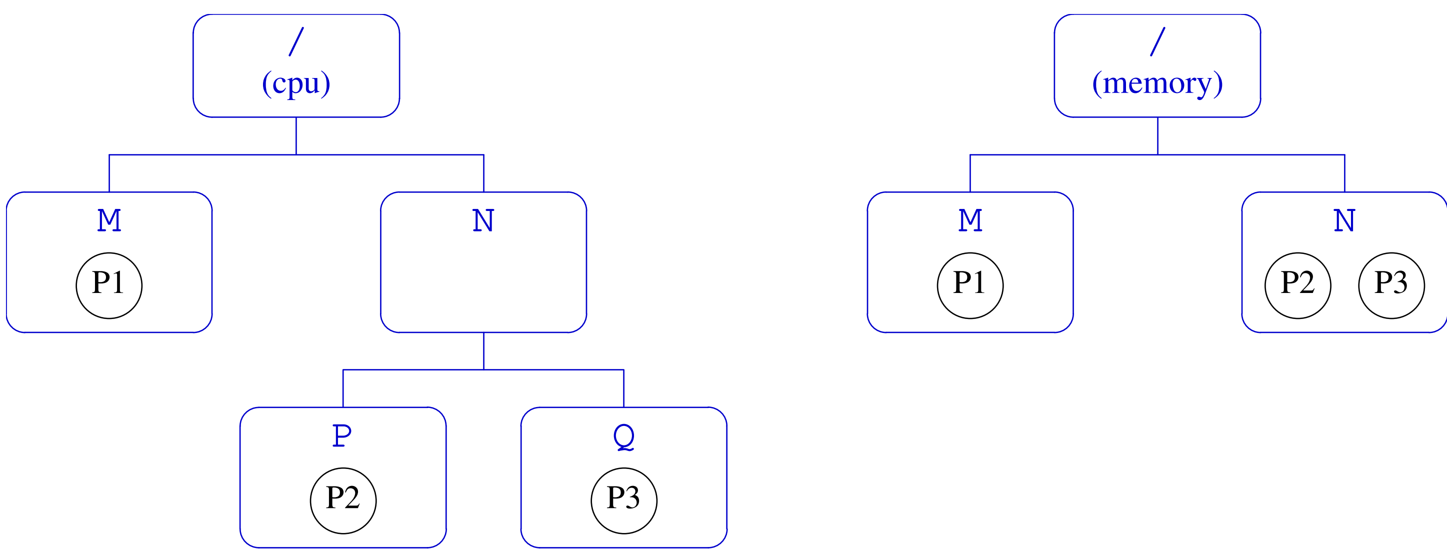
Cgroups version 2

- Designed to address perceived problems with cgroups v1
  - `Documentation/cgroup-v2.txt` details the problems
- Cgroups v2 officially released in Linux 4.5 (March 2016)
  - After lengthy development phase...
    - Because kernel code touched by cgroups was wide ranging, development was (unusually) done in mainline kernel
- Both cgroups v1 and cgroups v2 can be used on same system
  - **But** can't mount same controller in both filesystems

Problems with cgroups v1: multiple hierarchies

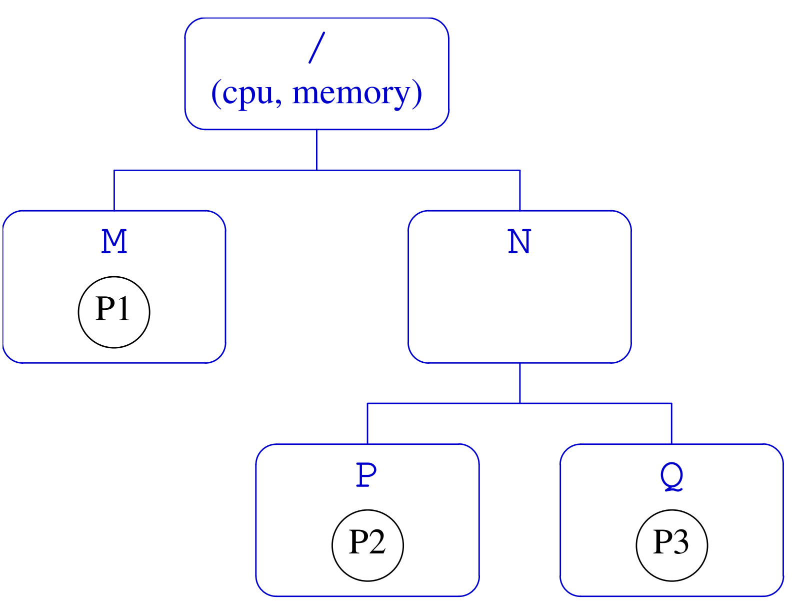
- **V1 hierarchy scheme** was supposed to allow great flexibility
  - V1: arbitrary number of hierarchies; one or more controllers can be mounted against each hierarchy
- But, that flexibility was **less useful than originally envisaged**
- What are pros and cons of separate hierarchy per controller versus attaching multiple controllers to same hierarchy?

Attaching v1 controllers to separate hierarchies



- Attaching cgroups v1 controllers to separate hierarchies means they can **manage processes at different granularities**
  - `cpu` can finely control CPU allocation for P2 and P3
  - `memory` allows P2 and P3 to share a memory allocation
- ☹ But when **moving process across cgroups** (e.g., moving P2 to cgroup M), operation **must be repeated in each hierarchy**
  - Cumbersome, slow, and nonatomic

Attaching multiple v1 controllers to the same hierarchy



- Placing multiple cgroups v1 controllers on same hierarchy removes need to replicate move operations in multiple hierarchies
- ☹ But, **controllers must manage processes to same level of granularity**
  - E.g., P2 and P3 can no longer share a memory allocation
    - Must make specific allocation decisions for P2 and P3

Problems with cgroups v1: multiple hierarchies

Other problems with the v1 hierarchy scheme:

- ☹ Utility controllers (e.g., `freezer`) that might be useful in all hierarchies could be used in only one
  - E.g., to freeze all processes in a `cpu` cgroup, there must be a `freezer` cgroup with same membership
    - And same is true if we want to freeze a `memory` cgroup, etc.
    - Argues in favor of attaching all controllers to same hierarchy or maintaining parallel hierarchies that are highly similar
  - Note: it's not possible to do a remount operation that moves a controller from one hierarchy to another

Problems with cgroups v1: multiple hierarchies

- In most use cases, **completely orthogonal (i.e., nonparallel) hierarchies were not needed**
- More common requirement: have **different levels of granularity per controller**
  - E.g., control memory only to a certain level in tree, but provide finer-grained control of CPU at deeper levels
- ⇒ Applications commonly put **most controllers on separate, but highly similar, hierarchies**
  - Topology of trees differed in cases where different granularity of control was needed
  - Only very closely related controllers (e.g., `cpu` and `cpuacct`) were put on same hierarchy
    - I.e., controllers that did not need to differ in granularity of control

Problems with cgroups v1: multiple hierarchies

- ⇒ **v2 uses single hierarchy for all controllers**
  - Establish common resource domain across different resource types, so controllers (e.g., `memory` and `io`) can cooperate
  - And there is a mechanism to allow per-controller granularity in the hierarchy

Problems with cgroups v1: thread granularity

Allowing **thread granularity** for cgroup membership proved problematic

- Didn't make sense for some controllers
  - E.g., `memory` controller (threads share memory...)
- Writing TIDs to `tasks` file is a **system-level activity**
  - At system level it is hard to know which thread does what
  - Probably, only the **application** itself really understands its thread topology
- ⇒ **v2 allows only process-granularity** membership
  - But starting with Linux 4.14, there is a limited form of thread granularity for some controllers...

Problems with cgroups v1: cgroups vs tasks

- Allowing a cgroup to contain both tasks and child cgroups was problematic in some cases
  - Two different types of entities–*tasks* and *groups* of tasks–compete for distribution of same resources
    - Different controllers dealt with this in differing ways...
    - which could cause difficulties if trying to generically combine multiple controllers on same hierarchy / parallel hierarchies
  - ⇒ **In v2, only leaf cgroups can contain processes**
    - (The story is a little more subtle; we'll revisit)

Problems with cgroups v1: inconsistency

- **Inconsistencies** between controllers ("design followed implementation")
  - With some controllers, new cgroups inherit parent's attributes; in others, they get defaults
  - Some controllers have controller-specific interfaces in root cgroup; others don't
  - Inconsistent use of values in cgroup files (e.g., "maximum" represented as "`-1`" vs "`max`")
  - v2: **consistent names and values** for interface files, **consistent inheritance rules** for all controllers
    - With some clearly documented guidelines!

Problems with cgroups v1: cgroup release

- V1 cgroup release mechanism (firing up a process) has problems:
  - Firing up a process is expensive
  - Can't delegate release handling to process inside a container
  - ⇒ **v2 has a lightweight solution**

Outline

1 Introduction 3
2 Problems with cgroups v1; rationale for v2 4
3 Cgroups v2 controllers 16
4 Enabling and disabling controllers 20
5 Organizing cgroups and processes 33
6 Release notification (cgroup.events file) 38
7 Optional topic: delegation 41
8 Optional topic: overview of thread mode 47

Cgroups v2 controllers

- V2 implements only a subset of equivalents of v1 controllers
  - Work in progress...
- `Documentation/cgroup-v2.txt` documents v2 controllers

Controllers available in cgroups v2

- `memory`: control distribution of memory
  - Successor to v1 `memory` controller
- `io`: regulate distribution of I/O resources
  - Successor to v1 `blkio` controller
- `pids`: control number of processes
  - Exactly the same as v1 controller
- `perf_event`: per-cgroup *perf* monitoring (since Linux 4.11)
  - Same as v1 controller (added in same kernel version)
- `rdma`: distribution and accounting of RDMA resources (since Linux 4.11)
  - Same as v1 controller (added in same kernel version)

Controllers available in cgroups v2

- `devices`: control access to devices (since Linux 4.15)
  - Successor to v1 `devices` controller
  - No interfaces files; instead control is done by attaching eBPF (`BPF_CGROUP_DEVICE`) program to cgroup
    - Each attempt to access device is gated by decision that eBPF program returns to kernel
- `cpu`: successor to v1 `cpu` and `cpuacct` controllers (since Linux 4.15)
- As at Linux 4.19, v2 currently lacks equivalents of:
  - `cpuset`
  - `freezer`
  - `hugetlb`
  - `net_cls` + `net_prio` (presumably will be combined?)

Outline

1 Introduction 3
2 Problems with cgroups v1; rationale for v2 4
3 Cgroups v2 controllers 16
4 Enabling and disabling controllers 20
5 Organizing cgroups and processes 33
6 Release notification (cgroup.events file) 38
7 Optional topic: delegation 41
8 Optional topic: overview of thread mode 47

Mounting the cgroups v2 filesystem

- To use cgroups v2, we mount new filesystem type:

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

  - Recent *systemd* automatically creates such a mount point, at `/sys/fs/cgroup/unified`
- All **v2 controllers are automatically available** under single hierarchy
  - No need to explicitly bind controllers to mount point
    - I.e., we don't specify `-o` *`controller`* mount option

The `cgroup.controllers` file

- Each v2 cgroup has a (read-only) `cgroup.controllers` file, which lists **available controllers** this cgroup can enable
- But, if we look in cgroups v2 root directory, we might find `cgroup.controllers` is empty:

```
# cd /sys/fs/cgroup/unified
# cat cgroup.controllers
# wc -l cgroup.controllers
0 cgroup.controllers
```

- V2 controller is available only if not bound in v1 hierarchy

```
# cat /proc/mounts | grep pids
cgroup /sys/fs/cgroup/pids cgroup rw,...,pids 0 0
```

  - That's why we didn't see `pids` in v2 `cgroup.controllers`

Ensuring that a controller is available in cgroups v2

- May need to unmount controller in v1 hierarchy to have it available in v2 hierarchy:

```
# umount /sys/fs/cgroup/pids
# cat /sys/fs/cgroup/unified/cgroup.controllers
pids
```

- But FS is really unmounted only if:
  - All processes are in root cgroup
  - There are no child cgroups
  - No process has open FDs or CWD on filesystem
  - *systemd* may have created child cgroups with processes...

Ensuring that a controller is available in cgroups v2

- One solution: move all process to root cgroup, and remove all child cgroups

```
cd /sys/fs/cgroup/pids/
pids=$(cat $(find */ -name cgroup.procs))
for p in $pids; do
    echo $p > cgroup.procs
done
rmdir $(find */ -type d | sort -r)
cd ..; umount /sys/fs/cgroup/pids
```

  - Get list of all PIDs in child cgroups
  - Write those PIDs to `cgroup.proc` in root cgroup
  - Remove all child cgroups (from bottom up)
  - Move out of root directory of filesystem before unmounting
  - See `cgroups/remove_cgroup_hier.sh`

Ensuring that a controller is available in cgroups v2

- Alternatively, (since Linux 4.6) use kernel boot parameter, `cgroup_no_v1`:
  - `cgroup_no_v1=all` ⇒ disable all v1 controllers
  - `cgroup_no_v1=controller,...` ⇒ disable selected controllers

(*systemd* falls back ok if no v1 controllers are available)

Enabling and disabling controllers

- Controllers are enabled/disabled by writing some subset of available controllers to `cgroup.subtree_control`

```
# echo "+pids -memory" > cgroup.subtree_control
```

  - `+` ⇒ enable controller, `-` ⇒ disable controller
- Enabling a controller in `cgroup.subtree_control`:
  - Allows resource to be **controlled in child cgroups**
  - **Creates controller-specific attribute files in each child directory**
- ⚠ ⚠ Attribute files in child cgroups are **used by process managing parent cgroup** to manage resource allocation across child cgroups
  - Different from v1...

Example: enabling a controller

- In the cgroup root directory, list available controllers:

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

- Create a child cgroup; see what files are in subdirectory:

```
# mkdir grp1
# ls grp1
cgroup.controllers  cgroup.events  cgroup.procs
cgroup.subtree_control
```

- Enable `pids` controller for child cgroups; new control files have been created in child cgroup:

```
# echo '+pids' > cgroup.subtree_control
# ls grp1
cgroup.controllers  cgroup.subtree_control  pids.max
cgroup.events       pids.current
cgroup.procs        pids.events
```

Example: enabling a controller

- In `grp1` cgroup, only available controller is `pids`:

```
# cat grp1/cgroup.controllers
pids
```

- In child of `grp1`, we can enable `pids` controller:

```
# mkdir grp1/sub
# echo '+pids' > grp1/cgroup.subtree_control
# cat grp1/cgroup.subtree_control
pids
```

- But `io` controller is not available:

```
# echo '+io' > grp1/cgroup.subtree_control
sh: echo: write error: No such file or directory
```

  - `ENOENT` error because "entry we are trying to add to `subtree_control` does not exist in `controllers`"

Top-down constraints

- Child cgroups are always subject to any resource constraints established by controllers in ancestor cgroups
  - $\Rightarrow$ Descendant cgroups can't relax constraints imposed by ancestor cgroups
- If a controller is disabled in a cgroup (i.e., not written to `cgroup.subtree_control` in parent cgroup), it cannot be enabled in any descendants of the cgroup

Exercises

1. This exercise demonstrates that resource constraints apply in a top-down fashion, using the cgroups v2 `pids` controller.
   - **If** it is not already mounted, mount the `cgroup2` filesystem

     ```
     # mount -t cgroup2 none /sys/fs/cgroup/unified
     ```

     - If you can't do this because the `/sys/fs/cgroup` FS is mounted read-only, then remount it as read-write:

       ```
       # mount -o remount,rw /sys/fs/cgroup
       ```

   - Check that the `pids` controller is visible in the cgroup root `cgroup.controllers` file. If it is not, unmount the cgroup v1 `pids` filesystem. (See the steps at the start of this section.)
     - In some cases, unmounting the cgroup v1 `pids` FS may not be enough, since the controller is in use (e.g., by *systemd*). So, it may be necessary to reboot with the boot parameter `cgroup_no_v1=pids`. [Exercise continues to next page...]

Exercises

- To simplify the following steps, change your current directory to the cgroup root directory (i.e., the location where the `cgroup2` filesystem is mounted; on recent *systemd*-based systems, this would be `/sys/fs/cgroup/unified`).
- Create a child and grandchild directory in the cgroup filesystem and enable the PIDs controller in the root directory and the first subdirectory:

```
# mkdir xxx
# mkdir xxx/yyy
# echo '+pids' > cgroup.subtree_control
# echo '+pids' > xxx/cgroup.subtree_control
```

- Set an upper limit of 10 tasks in the child cgroup, and an upper limit of 20 tasks in the grandchild cgroup:

```
# echo '10' > xxx/pids.max
# echo '20' > xxx/yyy/pids.max
```

Exercises

- In another terminal, use the supplied `cgroups/fork_bomb.c` program with the following command line, which causes the program to first sleep 60 seconds and then create 30 children:

```
$ ./fork_bomb 30 60
```

- The parent process in the `fork_bomb` program prints its PID before sleeping. While it is sleeping, return to the first terminal and place the parent process in the grandchild `pids` cgroup:

```
# echo parent-PID  > xxx/yyy/cgroup.procs
```

- When the parent finishes sleeping, how many children does it successfully create?

Outline

1 Introduction 3
2 Problems with cgroups v1; rationale for v2 4
3 Cgroups v2 controllers 16
4 Enabling and disabling controllers 20
5 Organizing cgroups and processes 33
6 Release notification (cgroup.events file) 38
7 Optional topic: delegation 41
8 Optional topic: overview of thread mode 47

Organizing cgroups and processes

Broadly similar to cgroups v1:

- Hierarchy organized as set of subdirectories
- All processes initially in root cgroup
- Move process into group by writing PID into `cgroup.procs`
- Read `cgroup.procs` to discover process membership
  - ⚠ Returned list is not sorted
  - ⚠ List may contain duplicate PIDs
    - E.g., if PID moved out and then back into cgroup, or PID recycled, while reading
- Child of *fork()* inherits parent's cgroup membership
- Cgroup directory with no (non-zombie) process members or child cgroups can be removed

Organizing cgroups and processes

Differences between v1 and v2:

- Root cgroup does not contain controller interface files
  - Control is not exercised on processes in root cgroup
- Cgroup can't both control cgroup children and have member processes
  - $\Rightarrow$ Place member processes in leaf nodes
- In initial implementation, cgroups v2 supported only process-level granularity
  - Writing TID of any thread to `cgroup.procs` moves all of process's threads to cgroup
  - No `tasks` file
  - From Linux 4.14, a limited form of thread-granularity cgroup membership is restored for certain controllers
    - So-called "thread mode"

“Only leaf nodes can have member processes”

- Earlier statement: “a cgroup can’t have both child cgroups and member processes”
- Let’s refine that...
- A cgroup can’t both:
  - distribute a resource to child cgroups, **and**
  - have child processes

“Only leaf nodes can have member processes”

- Revised statement: “A cgroup can’t both distribute resources and have member processes”
- Conversely (1):
  - A cgroup **can** have member processes and child cgroups...
  - **iff** it does not enable controllers for child cgroups
- Conversely (2):
  - If cgroup has child cgroups and processes, the processes must be moved elsewhere before enabling controllers
    - E.g., processes could be moved to child cgroups
- ⚠ This rule changes for certain controllers in Linux 4.14

Outline

1 Introduction 3
2 Problems with cgroups v1; rationale for v2 4
3 Cgroups v2 controllers 16
4 Enabling and disabling controllers 20
5 Organizing cgroups and processes 33
6 Release notification (cgroup.events file) 38
7 Optional topic: delegation 41
8 Optional topic: overview of thread mode 47

Cgroup (un)populated notification

- Cgroups v1: firing up a process is an expensive way of get notification of an empty cgroup!
- Cgroups v2: dispenses with `release_agent` and `notify_on_release` files
- Instead, each (non-root) cgroup has a file, `cgroup.events`, with a `populated` field:

```
# cat grp1/cgroup.events
populated 1
```

  - 1 == subhierarchy contains live processes
    - I.e., live process in any descendant cgroup
  - 0 == no live processes in subhierarchy

Cgroup (un)populated notification

- Can monitor `cgroup.events` file, to get notification of transition between populated and unpopulated states
  - *inotify*: transitions generate `IN_MODIFY` events
  - *poll()*/*epoll*: transitions generate `POLLPRI`/`EPOLLPRI` events
- One process can monitor multiple `cgroup.events` files
  - Much cheaper notification!
  - **Notification can be delegated** per container
    - I.e., one process can monitor all `cgroup.events` files in a subhierarchy

Outline

1 Introduction 3
2 Problems with cgroups v1; rationale for v2 4
3 Cgroups v2 controllers 16
4 Enabling and disabling controllers 20
5 Organizing cgroups and processes 33
6 Release notification (cgroup.events file) 38
7 Optional topic: delegation 41
8 Optional topic: overview of thread mode 47

Delegation

- Delegation == passing management of some subtree of hierarchy to another (less privileged) user
  - I.e., some other user who will manage resource control in the subhierarchy of processes
- Terminology:
  - **Delegater**: privileged user who owns a parent cgroup
  - **Delegatee**: less privileged user who is assigned management of a subhierarchy under parent cgroup

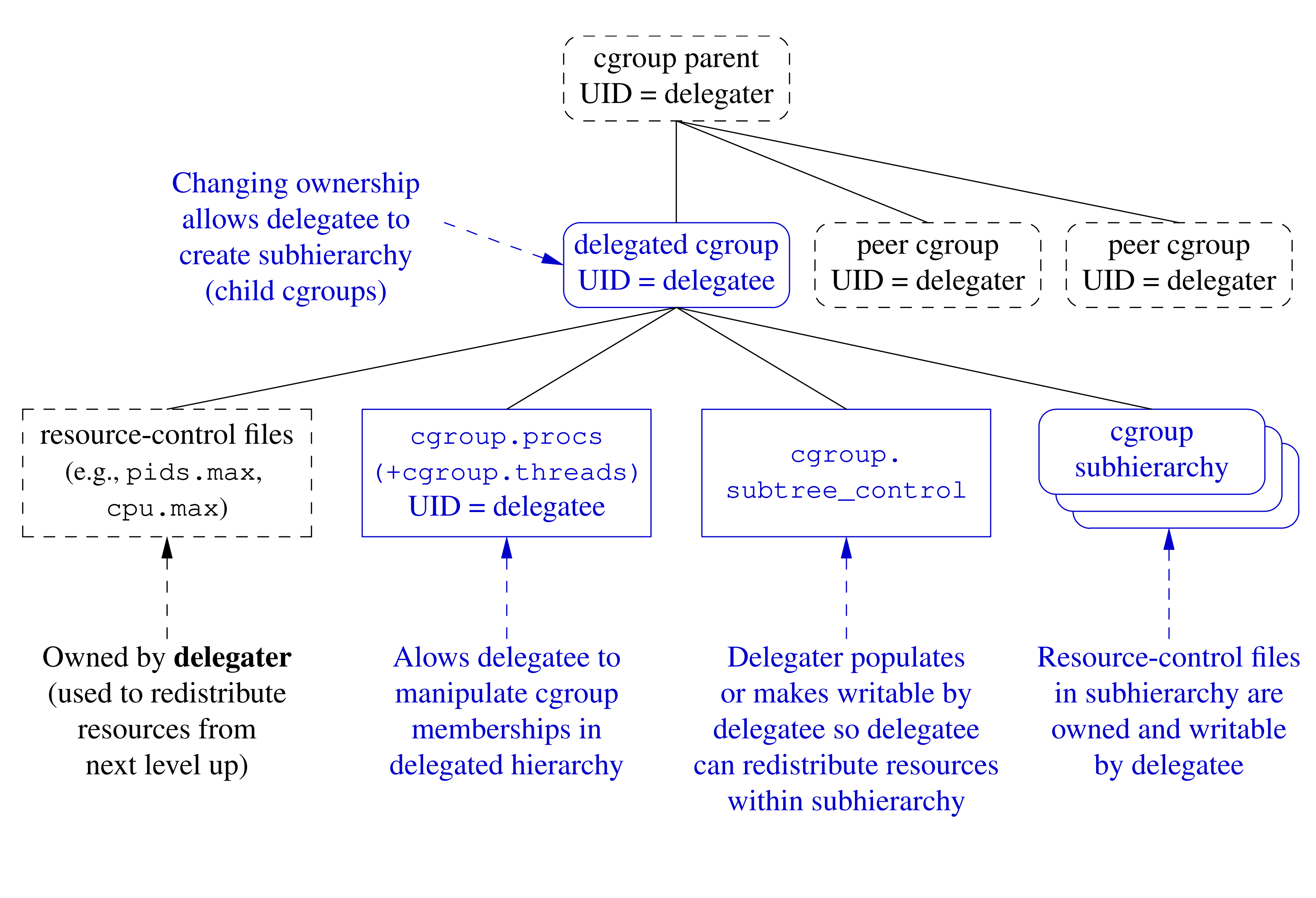
Delegation set-up

- To set up delegation, delegater grants delegatee write access to certain files
  - Normally done by changing ownership to UID of delegatee
- In addition to directory at root of delegated subtree, ownership of following files inside that directory is changed:
  - `cgroups.procs`
  - `cgroup.subtree_control`
    - So that delegatee can control resources in child cgroups it creates
  - `cgroup.threads`, if delegating a threaded subtree

Delegation set-up

- ⚠ Delegater **should not** make resource-control interface files writable by delegatee
  - Those files are used by **parent** (delegater) to control resource allocation in the child (delegatee)
  - ⇒ Delegatee should not have permission to change them

Delegation set-up



Post-delegation operation

- After delegation, delegatee can:
  - Create subhierarchy under delegated cgroup
  - Move process between cgroups inside subhierarchy
    - But, "delegation containment" rules mean delegatee can't move process into/out of subhierarchy (see *cgroups(7)*)
  - Control distribution of resources in subhierarchy
    - If controller is present in `cgroup.subtree_control`

Outline

1 Introduction 3
2 Problems with cgroups v1; rationale for v2 4
3 Cgroups v2 controllers 16
4 Enabling and disabling controllers 20
5 Organizing cgroups and processes 33
6 Release notification (cgroup.events file) 38
7 Optional topic: delegation 41
8 Optional topic: overview of thread mode 47

Background

- Original design goal in v2: all threads in multithreaded (MT) process are always in same cgroup
- V1 permitted threads to be split across cgroups, but:
  - This made no sense for some controllers (e.g., `memory`)
  - Writing TIDs to `tasks` file is a **system-level activity**, but only **applications** really understand their thread topology

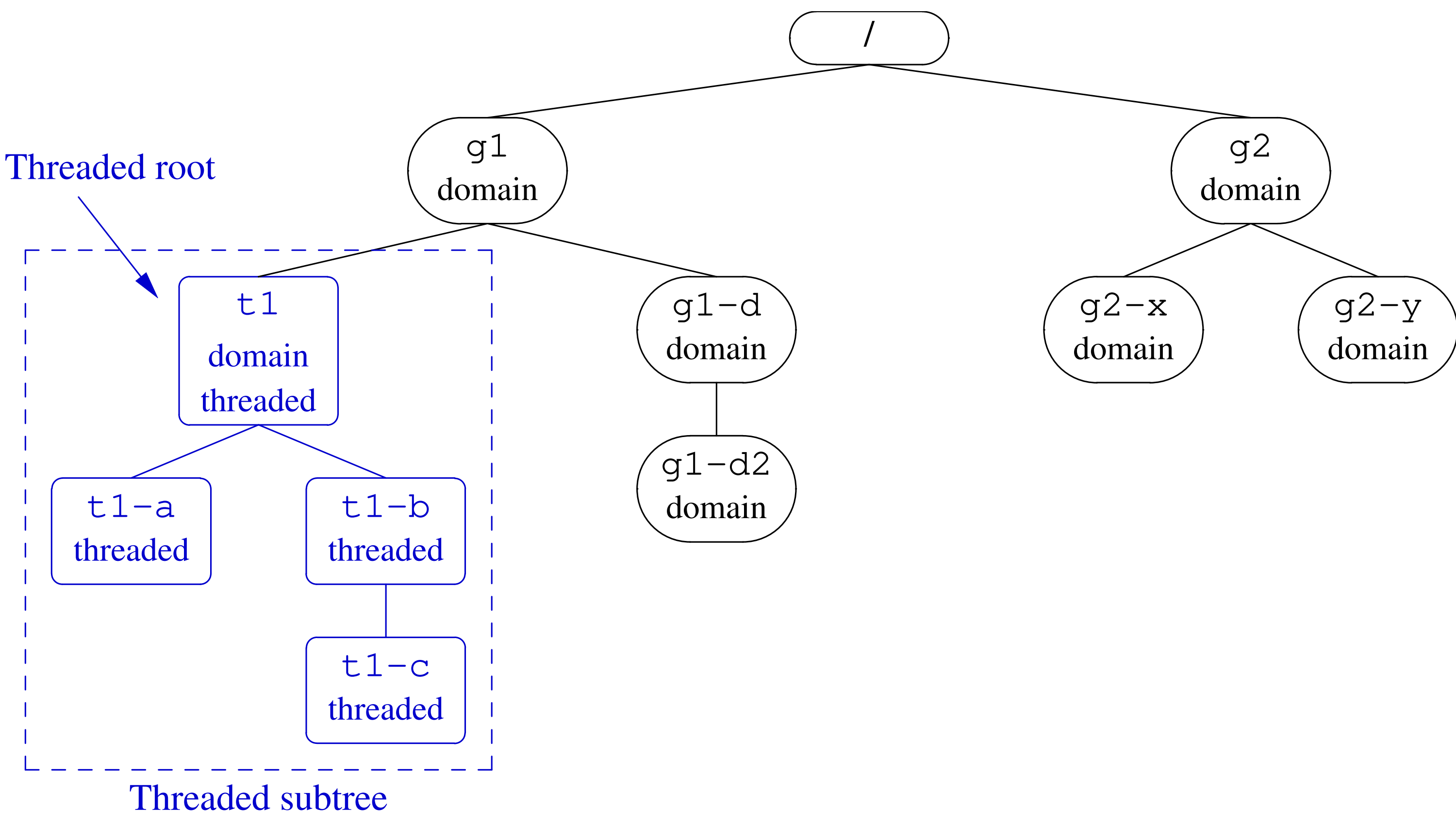
Background

- Nevertheless, there were use cases for thread-level control with `cpu` controller
- Result was a stand-off for a long period:
  - Cgroups v2 developers: "control is only at process level"
  - Kernel scheduler maintainers: "we won't merge a v2 `cpu` controller that doesn't allow thread-granularity control"
- Solution: **thread mode**, added in Linux 4.14
  - Allows thread-level granularity for certain controllers

“`domain`” versus “`threaded`” cgroups

- Cgroups in v2 hierarchy are initially all in “`domain`” mode:
  - All threads in MT process must be in same cgroup
  - This is the original cgroup v2 default
- Selected **subtrees** of hierarchy can be switched to “`threaded`” mode
  - All members of subtree must be “`threaded`” cgroups
  - Threads of MT processes can be in different cgroups under a “`threaded`” subtree
    - Restriction: all threads of a MT process must be inside **same** “`threaded`” subtree
- There can be multiple “`threaded`” subtrees, each containing multiple processes
- Thus, v2 now has thread granularity, but in more restricted manner than v1

Cgroup v2 thread mode



A threaded subtree within the cgroup v2 hierarchy

- Threads of MT process can be split across cgroups in threaded subtree

Threaded and domain controllers

Starting with Linux 4.14, there are two kinds of controllers...

- **Threaded** controllers: support thread-granularity control
  - `cpu`, `perf_event`, `pids`
- **Domain** (**nonthreaded**) controllers: support only process-granularity control

Threaded and domain controllers

- **Threaded** controllers understand threaded subtrees
  - IOW: controller-interface files for threaded controllers do appear in threaded subtrees
- To **domain** controllers, threaded subtrees are "invisible"
  - IOW: controller-interface files for domain controllers **do not** appear in threaded subtrees
    - I.e., domain controllers don't distribute resources in threaded subtree
  - From perspective of domain controllers, all threads in MT process appear to be in one cgroup–the "domain threaded" root cgroup
    - Because all threads must be in same threaded subtree

New interface files for thread mode

- `cgroup.threads`: define/view thread membership of cgroup
- `cgroup.type`: defines type of cgroup, and contains one of:
  - `domain`: normal group providing process-granularity control
  - `threaded`: a group that is a member of a threaded subtree
  - `domain threaded`: a domain group that serves as root of a threaded cgroup subtree
  - `domain invalid`: group in an "invalid" state
    - Can't be populated with processes and can't have controllers enabled
    - Can be converted to "`threaded`" group

Creating a threaded subtree

- There are two different ways of creating a threaded subtree
  - Full details are in the *cgroups(7)* manual page
- But many details and rules about how this must be done...
  - More complex than we have time to cover
  - Possible demo...
    - And use `cgroups/view_v2_cgroups.go` to inspect cgroups

Thanks!

Michael Kerrisk mtk@man7.org @mkerrisk

Slides at http://man7.org/conf/
Source code at http://man7.org/tlpi/code/

Training: Linux system programming, security and isolation APIs, and more; http://man7.org/training/

The Linux Programming Interface, http://man7.org/tlpi/