Diving deeper into control groups (cgroups) v2

Michael Kerrisk, man7.org © 2021

mtk@man7.org

20 October 2021, Kongsberg, Norway
## Outline

<table>
<thead>
<tr>
<th></th>
<th>Introduction</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Release notification</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Delegation</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>Thread mode</td>
<td>29</td>
</tr>
</tbody>
</table>
Who am I?

- Maintainer of Linux *man-pages* project since 2004
  - \( \approx 1060 \) pages, mainly for system calls & C library functions
    - (I wrote a lot of those pages...)
  - (Comaintainer since 2020)
- Author of a book on the Linux programming interface
  - http://man7.org/tlpi/
- **Trainer**/writer/engineer
  - http://man7.org/training/
- Email: mtk@man7.org
- Twitter: @mkerrisk
Outline

Topics:
- Release notification
- Delegation
- Thread mode

Questions: at the end
All of the following features were present in cgroups v1...

But better designed in cgroups v2
1 Introduction 3
2 Release notification 7
3 Delegation 14
4 Thread mode 29
Consider the following scenario:

- We create a cgroup subdirectory
- Some processes are moved into cgroup
- Eventually, all of those processes leave the cgroup
  - (Terminate or are moved to different cgroup)

We can get a notification when last process leaves cgroup

Example use cases:

- Manager process might want to know when all workers have terminated
- **systemd**: respawn a daemon that prematurely terminated
Cgroup (un)populated notification

- Each non-root cgroup has a file, \texttt{cgroup.events}, containing key-value pairs with state info about cgroup:

```bash
# cat grp1/cgroup.events
populated 1
frozen 0
```

- The \textit{Boolean} \texttt{populated} field tells us whether a cgroup has member processes
  - 1 \(\Rightarrow\) subhierarchy contains live processes
    - I.e., live process in cgroup, or in any descendant cgroup
  - 0 \(\Rightarrow\) no live processes in subhierarchy
Cgroup (un)populated notification

- Can monitor `cgroup.events` file, to get notification of changes to keys
  - `inotify`: changes generate `IN_MODIFY` events
  - `poll()/epoll/select()`: changes generate `POLLPRI` / `EPOLLPRI` / exceptional events
- After notification, parse `cgroup.events` to find populated key
- One process can monitor multiple `cgroup.events` files
  - Notification can be delegated per container
    - I.e., different processes can monitor `cgroup.events` files in different subhierarchies
    - Was not possible in cgroups v1...
Release notification example

- Create a cgroup that we will populate with processes:

```bash
sh1# cd /sys/fs/cgroup
sh1# mkdir mygrp
```

- In a second shell, monitor `cgroup.events` file using `inotify`

```bash
sh2$ cd /sys/fs/cgroup
sh2$ while inotifywait -q -e modify mygrp/cgroup.events; do
grep populated mygrp/cgroup.events | sed 's/~/ /'
done
```

- On each notification, loop displays value of `populated` key
Release notification example

- In first shell, place a `sleep` process in `mygrp`:

```
sh1# sleep 1000 &
[1] 8197
sh1# echo 8197 > mygrp/cgroup.procs
```

- In second shell we see:

```
mygrp/cgroup.events MODIFY
populated 1
```

- If we place a second `sleep` process in cgroup, `populated` key does not change:

```
sh1# sleep 2000 &
[2] 8650
sh1# echo 8650 > mygrp/cgroup.procs
sh1# grep populated mygrp/cgroup.events
populated 1
```

- And no `inotify` notification occurs in second shell
Release notification example

- If we kill first *sleep* process, *populated* key doesn’t change:

  ```
  sh1# kill %1
  [1]- Terminated sleep 1000
  sh1# grep populated mygrp/cgroup.events
  populated 1
  ```

- And no *inotify* notification occurs in second shell

- Then we kill the second *sleep* process:

  ```
  # kill %2
  ```

- In second terminal, we get an *inotify* notification and see that *populated* key has changed:

  ```
  mygrp/cgroup.events MODIFY
  populated 0
  ```
Outline

1. Introduction 3
2. Release notification 7
3. Delegation 14
4. Thread mode 29
Delegation

- So far, we always did cgroup operations as superuser....
- But for, say, running an unprivileged container, we would like to manage cgroups as an unprivileged user
- **Delegation** == passing management of some subtree of hierarchy to another (less privileged) user
- 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
  - ⇒ Change ownership to UID of delegatee
- We change ownership of directory that will be root of delegated subtree, and certain files inside that directory:
  - cgroups.procs
  - cgroup.subtree_control
- And (if they are present) any other filenames listed in /sys/kernel/cgroup/delegate

```
$ cat /sys/kernel/cgroup/delegate
cgroup.procs
cgroup.threads
cgroup.subtree_control
memory.oom.group
```

- (Future-proofing for new delegatable files added in future kernel versions)
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

Changing ownership allows delegatee to create subhierarchy (child cgroups)

- **cgroup parent**
  - UID = delegater

- **delegated cgroup**
  - UID = delegatee

- **peer cgroup**
  - UID = delegater

- **peer cgroup**
  - UID = delegater

**resource-control files** (e.g., pids.max, cpu.max)

- Owned by **delegater** (used to redistribute resources from next level up)

**cgroup.procs** (+cgroup.threads)

- UID = delegatee

**cgroup.subtree_control**

- Delegater populates or makes writable by delegatee so delegatee can redistribute resources within subhierarchy

**cgroup subhierarchy**

- Resource-control files in subhierarchy are owned and writable by delegatee

- Allows delegatee to manipulate cgroup memberships in delegated hierarchy

- Owned by **delegater** (used to redistribute resources from next level up)

**man7.org**

©2021, Michael Kerrisk @mkerrisk

Diving deeper into control groups (cgroups) v2 18 / 37
Post-delegation operation

- After delegation, delegatee can:
  - Create subhierarchy under delegated cgroup
  - Organize processes in that subhierarchy
  - Control distribution of resources in subhierarchy
    - If controller is present in \texttt{cgroup.subtree_control}
Delegation containment

- Process with non-\textit{root} UID can migrate “target” PID to \texttt{cgroup.procs} file only if following are true:
  - Writer has write access to \texttt{cgroup.procs} in destination cgroup
  - Writer has write access to \texttt{cgroup.procs} in nearest common ancestor of source and destination cgroups
- $\Rightarrow$ A delegated hierarchy is “contained”
  - Delegatee can move processes between cgroups inside subhierarchy
  - Delegatee can’t move processes into/out of subhierarchy
Delegation containment

Boxes with UIDs are cgroups delegated to specified UID

According to delegation containment rules, UID 1000 could move a process from M to N, or M to X, or J to B

But not, for example, from X to Y
Delegation example

- Privileged user enables `pids` controller for child subgroups
  
  ```
  # cd /sys/fs/cgroup
  # echo '+pids' > cgroup.subtree_control
  ```

- Create child group
  
  ```
  # mkdir dlgt_grp
  ```

- Limit number of processes in the new cgroup:
  
  ```
  # echo 20 > dlgt_grp/pids.max
  ```

  (Just to illustrate the exercise of control from the upper level, by delegater)
Delegation example

- Pass ownership of new directory and its `cgroup.procs` and `cgroup.subtree_control` files to unprivileged user (`mtk`):

  ```
  # chown mtk:mtk dlgt_grp dlgt_grp/cgroup.procs dlgt_grp/cgroup.subtree_control
  ```

- Verify set-up

  ```
  # ls -ld dlgt_grp dlgt_grp/cgroup.procs dlgt_grp/cgroup.subtree_control
  drwxr-xr-x. 2 mtk mtk [...] dlgt_grp
  -rw-r--r--. 1 mtk mtk [...] dlgt_grp/cgroup.procs
  -rw-r--r--. 1 mtk mtk [...] dlgt_grp/cgroup.subtree_control
  ```
Delegation example

- Unprivileged user enables `pids` controller in delegated cgroup and creates some child cgroups under delegated cgroup:

```bash
$ whoami
mtk
$ cd /sys/fs/cgroup/dlgt_grp/
$ echo '+pids' > cgroup.subtree_control
$ mkdir grp0 grp1 grp2
```

- We can see that `pids` controller is enabled in new cgroups:

```bash
$ ls grp1/pids.*
grp1/pids.current  grp1/pids.events  grp1/pids.max
```
Delegation example

- Let’s try to put a shell run by unprivileged user `mtk` into delegated hierarchy:

```bash
$ cd /sys/fs/cgroup/dlgt_grp
$ ls -ld grp0/cgroup.procs
-rw-r--r--. 1 mtk mtk [...] grp0/cgroup.procs
$ echo $$
2705
$ echo 2705 > grp0/cgroup.procs
bash: echo: write error: Permission denied
```

- What went wrong?
  - Already saw that `cgroup.procs` was writable by `mtk`...
  - But, this `shell was in root cgroup`, and
  - `mtk` doesn’t have “write access to `cgroup.procs` in common ancestor of source and destination cgroups”
    - (Common ancestor is the root cgroup)
Delegation example

- So, privileged process must insert initial process into delegated cgroup
  - Initial process then creates other processes inside cgroup
  - Unprivileged user/manager can move processes **within** delegated hierarchy
- In our example, we’ll use the shell as both initial process and manager in delegated cgroup
- So, our **privileged** user puts the unprivileged shell into delegated hierarchy:

```
# echo 2705 > dlgt_grp/grp0/cgroup.procs
```
Delegation example

Returning to unprivileged shell, let’s see how things look:

```
$ whoami
mtk
$ pwd
/sys/fs/cgroup/dlgt_grp
$ cat /proc/self/cgroup | grep '0::'
0::/dlgt_grp/grp0
```

- The shell is now inside the delegated cgroup
- 0:: entry shows process’s membership in v2 hierarchy

Let’s create a child process and see what cgroup it’s in:

```
$ sleep 1000 &
[1] 25591
$ cat /proc/25591/cgroup | grep '0::'
0::/dlgt_grp/grp0
```

- (Child process inherits parent’s cgroup membership)
Delegation example

- We can move the child process to another cgroup in the delegated hierarchy:

  ```
  $ echo 25591 > grp1/cgroup.procs
  $ cat /proc/25591/cgroup | grep '0::'
  0::/dlgt_grp/grp1
  ```

- But we can’t move it to cgroup outside delegated hierarchy:

  ```
  $ echo 25591 > /sys/fs/cgroup/cgroup.procs
  bash: /sys/fs/cgroup/cgroup.procs: Permission denied
  ```
Background

- Original design goal in v2: all threads in multithreaded (MT) process are always in same cgroup
- By contrast, v1 permitted threads to be split across cgroups
  - But, this made no sense for some controllers (e.g., memory)
- Despite the initial v2 design decision, 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
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, cpuset, perf_event, pids

- **Domain (nonthreaded)** controllers: support only process-granularity control
  - All other controllers...
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
    - (Recall that all threads of a process must be in same threaded subtree)
New interface files for thread mode

- **cgroup.threads**: define/view thread membership of cgroup
  - Write thread ID to this file to move thread to cgroup
  - Read file to get list of threads in cgroup
- **cgroup.type**: defines type of cgroup, and contains one of:
  - **domain**: normal group providing process-granularity control
    - (i.e., the original cgroup v2 behavior)
  - **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
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, Trainer and Consultant
http://man7.org/training/

mtk@man7.org @mkerrisk

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