Control Groups (cgroups): Introduction

Michael Kerrisk, man7.org © 2022

mtk@man7.org

January 2022

Outline

15 Cgroups: Introduction 15-1
15.1 Preamble 15-3
15.2 What are control groups? 15-9
15.3 An example: the pids controller 15-16
15.4 Creating and destroying cgroups 15-21
15.5 Populating a cgroup 15-25
15.6 Enabling and disabling controllers 15-33
15.7 Cgroups /proc files 15-50
Cgroups is a big topic
- Many controllers
- V2 versus V1 interfaces

We’ll focus on:
- General principles of operation; goals of cgroups
- The cgroup2 filesystem
- Interacting with cgroup2 filesystem using shell commands
- Origin of cgroups v2 (i.e., problems with cgroups v1)
- Differences between cgroups v2 and v1

We’ll look briefly at some of the controllers
Resources

- Kernel documentation files
  - V2: Documentation/admin-guide/cgroup-v2.rst
  - V1: Documentation/admin-guide/cgroup-v1/*.rst
    - Before Linux 5.3: Documentation/cgroup-v1/*.txt
- `cgroups(7)` man page
- Neil Brown’s (2014) LWN.net series on cgroups: https://lwn.net/Articles/604609/
  - Thought-provoking commentary on the meaning of grouping and hierarchy
- https://lwn.net/Articles/484254/ – Tejun Heo’s initial thinking about redesigning cgroups
- Other articles at https://lwn.net/Kernel/Index/#Control_groups

Some history

- 2006/2007, “Process Containers” @ Google ⇒ Cgroups v1
- Jan 2008: initial mainline kernel release (Linux 2.6.24)
  - Three resource controllers (all CPU-related) in initial release
- Subsequently, other controllers are added
  - memory, devices, freezer, net_cls, blkio...
- But a few years of uncoordinated design leads to a mess
  - Decentralized design fails us... again
- Sep 2012: work has already begun on cgroups v2...
Some history

- Sep 2015: systemd adds cgroup v2 support
- Mar 2016: cgroups v2 officially released (Linux 4.5)
  - But, lacks feature parity with cgroups v1
- Jan 2018: cpu controller is released for cgroups v2
  - (Absence had been major roadblock to adoption of v2)
- Oct 2019: Fedora 31 is first distro to move to v2-by-default
- 2020: Docker 20.10 gets cgroups v2 support
- 2021: other distros move to v2-by-default
  - Debian 11.0 (Aug 2021); Ubuntu 21.10 (Oct 2021); Arch

We are at a tipping point

- A lot of existing infrastructure depends on cgroups v1
- But:
  - A lot of migration work has already been done, and the distros have migrated to v2
  - Cgroups v2 offers a number of advantages over v1
  - ⇒ we’ll focus on cgroups v2, and later look at how v1 is different
What are control groups?

- Two principal components:
  - A mechanism for hierarchically grouping processes
  - A set of controllers (kernel components) that manage, control, or monitor processes in cgroups
- Interface is via a pseudo-filesystem
- Cgroup manipulation takes form of filesystem operations, which might be done:
  - Via shell commands
  - Programmatically
  - Via management daemon (e.g., systemd)
  - Via your container framework’s tools (e.g., LXC, Docker)
What do cgroups allow us to do?

- Limit resource usage of group
  - E.g., limit % of CPU available to group; limit amount of memory that group can use
- Prioritize group for resource allocation
  - E.g., favor the group for network bandwidth
- Resource accounting
  - Measure resources used by processes
- Freeze a group
  - Freeze, restore, and checkpoint a group
- And more...

Terminology

- **Control group**: a group of processes that are bound together for purpose of resource management
- **(Resource) controller**: kernel component that controls or monitors processes in a cgroup
  - E.g., memory controller limits memory usage; cpu controller limits CPU usage
  - Also known as **subsystem**
    - (But that term is rather ambiguous because so generic)
- Cgroups are arranged in a **hierarchy**
  - Each cgroup can have zero or more child cgroups
  - Child cgroups **inherit** control settings from parent
Cgroup filesystem directory structure defines cgroups + cgroup hierarchy

- I.e., use `mkdir(2)` / `rmdir(2)` (or equivalent shell commands) to create cgroups

- Each subdirectory contains automagically created files
  - Some files are used to manage the cgroup itself
  - Other files are controller-specific

- Files in cgroup are used to:
  - Define/display membership of cgroup
  - Control behavior of processes in cgroup
  - Expose information about processes in cgroup (e.g., resource usage stats)

---

The cgroup2 filesystem

- On boot, `systemd` mounts v2 hierarchy at `/sys/fs/cgroup`
  - (or `/sys/fs/cgroup/unified`)

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

- The (pseudo)filesystem type is “cgroup2”
  - In cgroups v1, filesystem type is “cgroup”

- The cgroups v2 mount is sometimes known as the “unified” hierarchy
  - Because all controllers are associated with a single hierarchy
  - By contrast, in v1 there were multiple hierarchies
You may be on a distro that uses *systemd*'s “hybrid” mode by default
- Hybrid mode combines use of cgroups v1 and v2
- Problem: can’t simultaneously use a controller in both v1 and v2
- Simplest solution is usually to reboot
  - If this shows a value > 1, then you need to reboot:

```bash
$ grep -c cgroup /proc/mounts  # Count cgroup mounts
```

- **Either**: use kernel boot parameter, `cgroup_no_v1`:
  - `cgroup_no_v1=all` ⇒ disable all v1 controllers
- **Or**: use `systemd.unified_cgroup_hierarchy` boot parameter
  - ⇒ *systemd* abandons its hybrid mode, uses just v2
Example: the pids controller

- **pids** ("process number") controller allows us to limit number of PIDs in cgroup (prevent `fork()` bombs!)
- Create new cgroup, and place shell’s PID in that cgroup:

```bash
# mkdir /sys/fs/cgroup/mygrp
# echo $$
17273
# echo $$ > /sys/fs/cgroup/mygrp/cgroup.procs
```

- `cgroup.procs` defines/displays PIDs in cgroup
- (Note ‘#' prompt ⇒ all commands done as superuser)
- Moving a PID into a group automatically removes it from group of which it was formerly a member
  - I.e., a process is always a member of exactly one group in the hierarchy
Example: the pids controller

- Can read `cgroup.procs` to see PIDs in group:

```shell
# cat /sys/fs/cgroup/mygrp/cgroup.procs
17273
20591
```

- Where did PID 20591 come from?
- PID 20591 is `cat` command, created as a child of shell
  - Child process inherits cgroup membership from parent

- `pids.current` shows how many processes are in group:

```shell
# cat /sys/fs/cgroup/mygrp/pids.current
2
```

- Two processes: shell + `cat`

---

Example: the pids controller

- We can limit number of PIDs in group using `pids.max` file:

```shell
# echo 5 > /sys/fs/cgroup/mygrp/pids.max
# for a in $(seq 1 5); do sleep 60 & done
[1] 21283
[2] 21284
[3] 21285
[4] 21286
bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
```

- (The shell retries a few times and then gives up)
- `pids.max` defines/exposes limit on number of PIDs in cgroup
- From a **different** shell, examine `pids.current`:

```shell
$ cat /sys/fs/cgroup/mygrp/pids.current
5
```

- Not possible from first shell (can’t create more processes)
Managing cgroups

- **Cgroup** === collection of processes
- **Cgroup hierarchy** === hierarchical arrangement of cgroups
  - Implemented via a pseudo-filesystem
  - Structure and membership of cgroup hierarchy is defined by:
    - **Mounting** `cgroup2` filesystem
    - **Creating a subdirectory structure** that reflects desired cgroup hierarchy
    - **Moving processes within hierarchy** by writing their PIDs to special files in cgroup subdirectories
      - E.g., `cgroup.procs`
Creating cgroups

- Initially, all processes on system are members of root cgroup
- New cgroups are created by creating subdirectories under cgroup mount point:
  
  ```
  # mkdir /sys/fs/cgroup/mygrp
  ```

- Relationships between cgroups are reflected by creating nested (arbitrarily deep) subdirectory structure

Destroying cgroups

An empty cgroup can be destroyed by removing directory

- Empty == last process in cgroup terminates or migrates to another cgroup and last child cgroup is removed
  - Presence of zombie process does not prevent removal of cgroup directory
    - (Notionally, zombies are moved to root cgroup)

- Not necessary (or possible) to delete attribute files inside cgroup directory before deleting it
Placing a process in a cgroup

- To move a **process** to a cgroup, we write its PID to **cgroup.procs** file in corresponding subdirectory
  
  ```
  # echo $$ > /sys/fs/cgroup/mygrp/cgroup.procs
  ```

- In multithreaded process, moves all threads to cgroup
- ⚠️ Can write only one PID at a time
  - Otherwise, **write()** fails with **EINVAL**
Viewing cgroup membership

- To see PIDs in cgroup, read `cgroup.procs` file
  - PIDs are newline-separated
  - Zombie processes do not appear in list
- ⚠️ List is **not guaranteed to be sorted or free of duplicates**
  - PID might be moved out and back into cgroup or recycled while reading list

Cgroup membership details

- A **process can be member of just one cgroup**
  - That association defines attributes / parameters that apply to the process
- Adding a process to a different cgroup automatically removes it from previous cgroup
- On `fork()`, **child inherits cgroup membership(s) of parent**
  - Afterward, cgroup membership(s) of parent and child can be independently changed
  - Since Linux 5.7, a child process can be created in a specific v2 cgroup using `clone3()` `CLONE_INTO_CGROUP`
Killing all processes in a cgroup

- Writing “1” to cgroup.kill kills all processes in a cgroup
  - Action is recursive
    - I.e., processes in descendant cgroups are also killed
  - Processes are killed using SIGKILL
  - File is write-only, and available only in non-root cgroups :-)
- Available since Linux 5.14
- All threads in a multithreaded process are killed
  - (Can’t write to cgroup.kill in a “threaded” cgroup)
- Example use cases:
  - Service managers (e.g., systemd) can kill all processes in a service
  - User-space “out-of-memory” (OOM) handlers can quickly kill/easily an entire cgroup
  - Handle some kill-container use cases that can’t be handled by killing container PID 1

Notes for online practical sessions

- Small groups in breakout rooms
  - Write a note into Slack if you have a preferred group
- We will go faster, if groups collaborate on solving the exercise(s)
  - You can share a screen in your room
- I will circulate regularly between rooms to answer questions
- Zoom has an “Ask for help” button...
- Keep an eye on the #general Slack channel
  - Perhaps with further info about exercise;
  - Or a note that the exercise merges into a break
- When your room has finished, write a message in the Slack channel: “***** Room X has finished *****”
  - Then I have an idea of how many people have finished
Using *tmate* in in-person practical sessions

In order to share an X-term session with others, do the following:

- Enter the command *tmate* in an X-term, and you will see the following:

  $$
  \text{tmate}
  $...
  \text{Connecting to ssh.tmate.io...}
  \text{Note: clear your terminal before sharing readonly access}
  \text{web session read only: ...}
  \text{ssh session read only: ...}
  \text{web session: ...}
  \text{ssh session: ssh S0mErAnD0m5Tr1Ng@lon1.tmate.io}
  $$

- Share the last “ssh” string with your colleagues via Slack or another channel

- Your colleagues should paste that string into an X-term...
  - After that, you will be sharing an X-term session in which anyone can type

Exercises

1. In this exercise, we create a cgroup, place a process in the cgroup, and then migrate that process to a different cgroup.
   - Create two subdirectories, *m1* and *m2*, in the cgroup root directory.
   - Execute the following command, and note the PID assigned to the resulting process:

```
# sleep 300 &
```

- Write the PID of the process created in the previous step into the file *m1/cgroup.procs*, and verify by reading the file contents.
- Now write the PID of the process into the file *m2/cgroup.procs*.
- Is the PID still visible in the file *m1/cgroup.procs*? Explain.
- Try removing cgroup *m1* using the command *rm -rf m1*. Why doesn’t this work?
- If it is still running, kill the *sleep* process and then remove the cgroups *m1* and *m2* using the *rmdir* command.