Control Groups (cgroups): Introduction

Michael Kerrisk, man7.org © 2024

January 2024

mtk@man7.org

Outline

16  Cgroups: Introduction 16-1
16.1 Preamble 16-3
16.2 What are control groups? 16-9
16.3 An example: the pids controller 16-16
16.4 Creating, destroying, and populating a cgroup 16-21
16.5 Enabling and disabling controllers 16-34
Goals

- 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
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
- 2012: work has already begun on cgroups v2...
Some history

- Sep 2015: **systemd** adds cgroup v2 support
  - (Based on kernel 4.2)
- Mar 2016: cgroups v2 officially released (Linux 4.5)
  - But, lacks feature parity with cgroups v1
- Jan 2018: **cpu** and **devices** controllers are 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 have passed the tipping point

- We have passed the v1-to-v2 tipping point:
  - **systemd**, Docker and other tools fully support cgroups v2, 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
- Resource accounting
  - Measure resources used by processes in group
- Limit device access
- Pin processes to CPU cores
- Shape network traffic
- 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
Filesistem interface

- 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`, if systemd is operating in cgroups “hybrid” mode)

```
# 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
Booting to cgroups v2

- 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, so that *systemd* abandons its hybrid mode, and uses just v2
  - If this shows a value > 1, then you need to reboot:
    
    ```
    $ 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
16 Cgroups: Introduction 16-1
16.1 Preamble 16-3
16.2 What are control groups? 16-9
16.3 An example: the pids controller 16-16
16.4 Creating, destroying, and populating a cgroup 16-21
16.5 Enabling and disabling controllers 16-34

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:

  ```
  # 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:

  ```
  # 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:

  ```
  # 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: 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`:

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

- Not possible from first shell (can’t create more processes)
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 (2020), a child process can be created in a specific v2 cgroup using `clone3()` `CLONE_INTO_CGROUP`
  - See `procexec/t_CLONE_INTO_CGROUP.c`
/proc/PID/cgroup file

- **/proc/PID/cgroup** shows cgroup memberships of PID
  - 8:cpu,cpuacct:/cpugrp3
  - 7:freezer: /
  - ... 
  - 0::/grp1

- **Hierarchy ID** (0 for v2 hierarchy)
  - Can be matched to hierarchy ID in another file, `/proc/cgroups` (but that file is not so interesting)

- **Comma-separated list of controllers bound to the hierarchy**
  - Field is empty for v2 hierarchy

- **Pathname of cgroup to which this process belongs**
  - Pathname is relative to cgroup root directory

- On a system booted in v2-only mode, there is just one line in this file (0:....)

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 (2021)

- Example use cases:
  - Service managers (e.g., `systemd`) can kill all processes in a service
  - User-space “out-of-memory” (OOM) handlers can quickly/easily kill 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

Shared screen etiquette

- **It may help your colleagues if you use a larger than normal font!**
  - In many environments (e.g., *xterm*, *VS Code*), we can adjust the font size with `Control+Shift+`+`"` and `Control+`-`
  - Or (e.g., *emacs*) hold down `Control` key and use mouse wheel
- **Long shell prompts** make reading your shell session difficult
  - Use `PS1='$ '` or `PS1='# '`
- **Low contrast** color themes are difficult to read; change this if you can
- Turn on **line numbering** in your editor
  - In *vim* use: `:set number`
  - In *emacs* use: `M-x display-line-numbers-mode` `<RETURN>`
    - `M-x` means `Left-Alt+x`
- For collaborative editing, **relative line-numbering is evil....**
  - In *vim* use: `:set nornu`
  - In *emacs*, the following should suffice:
    
    ```
    M-: (display-line-numbers-mode) <RETURN>
    M-: (setq display-line-numbers 'absolute) <RETURN>
    ```
    - `M-:` means `Left-Alt+Shift+:`
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’ll see the following:

  ```
  $ tmate
  ...
  Connecting to ssh.tmate.io...
  Note: clear your terminal before sharing readonly access
  web session read only: ...
  ssh session read only: ssh S0mErAnD0m5Tr1Ng@lon1.tmate.io
  web session: ...
  ssh session: ssh S0mEoTheRrAnD0m5Tr1Ng@lon1.tmate.io
  ```

- Share last “ssh” string with colleague(s) via Slack or another channel
  - Or: "ssh session read only" string gives others read-only access

- Your colleagues should paste that string into an X-term...

- Now, you are sharing an X-term session in which anyone can type
  - Any "mate" can cut the connection to the session with the 3-character sequence `<ENTER>~`.

- To see above message again: `tmate show-messages`

Booting to cgroups v2

- **In preparation for the following exercises**, if necessary reboot your system to use cgroups v2 only, as follows...

- First, check whether your system is already booted to use cgroups v2 only:

  ```
  $ grep cgroup /proc/mounts  # Is there a v2 mount?
  cgroup2 /sys/fs/cgroup cgroup2 ...
  $ grep cgroup /proc/mounts / grep -v name= | grep -vc cgroup2
  0  # 0 == no v1 controllers are mounted
  ```

  - If there is a v2 mount, and no v1 controllers are mounted, then you do not need to do anything further; otherwise:

  - From the GRUB boot menu, you can boot to cgroups v2–only mode by editing the boot command (select a GRUB menu entry and type “e”). In the line that begins with “`linux`”, add the following parameter:

    ```
    systemd.unified_cgroup_hierarchy
    ```
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 (/sys/fs/cgroup).
- 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.
Enabling and disabling controllers

- Each cgroup v2 directory contains two files:
  - `cgroup.controllers`: lists controllers that are **available** in this cgroup
  - `cgroup.subtree_control`: used to list/modify set of controllers that are **enabled** in this cgroup
    - Always a subset of `cgroup.controllers`

- Together, these files allow different controllers to be managed to **different levels of granularity** in v2 hierarchy
Available controllers: cgroup.controllers

- cgroup.controllers lists the controllers that are available in a cgroup
- Certain “automatic” controllers are available by default in every cgroup, and are not listed in cgroup.controllers
  - devices, freezer, network, perf_event

- A controller may not be available because:
  - Controller is not enabled in parent cgroup
    (Does not apply for “automatic” controllers)
  - The same controller is already in use in cgroups v1
  - Cgroups v1 and v2 can coexist, but a controller can be used in only one version
  - Kernel was built without support for that controller
  - Controller was disabled at boot time
    Using the boot option cgroup_disable=name[,..]
cgroup.subtree_control is used to show or modify the set of controllers that are enabled in a cgroup:

```
# cd /sys/fs/cgroup/
# cat cgroup.subtree_control
cpu io memory pids
```

- Set of controllers enabled in root cgroup will depend on distro and systemd configuration and version
- Contents of cgroup.subtree_control are always a subset of cgroup.controllers
- I.e., can't enable controller that is not available in a cgroup
- Controllers are enabled/disabled by writing to this file:

```
# echo '+cpuset' > cgroup.subtree_control # Enable a controller
# echo '-cpuset' > cgroup.subtree_control  # Disable a controller
```

Enabling a controller in cgroup.subtree_control:
- Allows resource to be controlled in child cgroups
- Causes controller-specific attribute files to appear in each child directory

Attribute files in child cgroups are used by process managing parent cgroup to manage resource allocation into child cgroups
- This is a significant difference from cgroups v1
Review situation in root cgroup:

```
# cd /sys/fs/cgroup/
# cat cgroup.controllers
cpuset cpu io memory hugetlb pids misc  
# cat cgroup.subtree_control
    cpu io memory pids
```

Create a small subhierarchy:

```
# mkdir -p grp_x/grp_y
```

Controllers available in `grp_x` are those that were enabled at level above; no controllers are enabled in `grp_x`:

```
# cat grp_x/cgroup.controllers
    cpu io memory pids
# cat grp_x/cgroup.subtree_control    # Empty...
```

Consequently, no controllers are available in `grp_y`:

```
# cat grp_x/grp_y/cgroup.controllers    # Empty...
```

List `cpu.*` files in `grp_y`:

```
# cd /sys/fs/cgroup/grp_x
# ls grp_y/cpu.*
    grp_y/cpu.pressure grp_y/cpu.stat
```

(These two files show CPU-related statistics and are present in every cgroup)

Enabling `cpu` controller in parent cgroup (`grp_x`) causes controller interface files to appear in child (`grp_y`) cgroup:

```
# echo '+cpu' > cgroup.subtree_control
# ls grp_y/cpu.*
    grp_y/cpu.idle          grp_y/cpu.max.burst  grp_y/cpu.stat
    grp_y/cpu.weight.nice   grp_y/cpu.max
    grp_y/cpu.weight
```

---

**cgroup.subtree_control example**

- **Review situation in root cgroup:**

```
# cd /sys/fs/cgroup/
# cat cgroup.controllers
 cpuset cpu io memory hugetlb pids misc
# cat cgroup.subtree_control
    cpu io memory pids
```

- **Create a small subhierarchy:**

```
# mkdir -p grp_x/grp_y
```

- **Controllers available in `grp_x` are those that were enabled at level above; no controllers are enabled in `grp_x`:**

```
# cat grp_x/cgroup.controllers
    cpu io memory pids
# cat grp_x/cgroup.subtree_control    # Empty...
```

- **Consequently, no controllers are available in `grp_y`:**

```
# cat grp_x/grp_y/cgroup.controllers    # Empty...
```

- **List `cpu.*` files in `grp_y`:**

```
# cd /sys/fs/cgroup/grp_x
# ls grp_y/cpu.*
    grp_y/cpu.pressure grp_y/cpu.stat
```

- **(These two files show CPU-related statistics and are present in every cgroup)**

- **Enabling `cpu` controller in parent cgroup (`grp_x`) causes controller interface files to appear in child (`grp_y`) cgroup:**

```
# echo '+cpu' > cgroup.subtree_control
# ls grp_y/cpu.*
    grp_y/cpu.idle          grp_y/cpu.max.burst  grp_y/cpu.stat
    grp_y/cpu.weight.nice   grp_y/cpu.max
    grp_y/cpu.weight
```
cgroup.subtree_control example

- After enabling controller in parent cgroup, we can limit resources in child cgroup...
- Set hard CPU limit of 50% in child cgroup (grp_y):
  
  ```sh
  # echo '50000 100000' > grp_y/cpu.max
  ```

- In another window, we start a program that burns CPU time and displays statistics; and we move it into grp_y:
  
  ```sh
  # echo 6445 > grp_y/cgroup.procs  # 6445 is PID of burner process
  ```

- In the other terminal, we see:

  ```
  $ ./cpu_burner
  [6445] %CPU = 99.86
  [6445] %CPU = 99.83
  ...
  [6445] %CPU = 83.52
  [6445] %CPU = 50.00
  [6445] %CPU = 50.00
  ...
  ```

Managing controllers to differing levels of granularity

- A controller is **available in child** cgroup only if it is **enabled in parent** cgroup:

  ```
  # cat cgroup.controllers
cpuset cpu io memory hugetlb pids
  # cat cgroup.subtree_control
  cpu_memory_pids
  # cat grp1/cgroup.controllers
cpuset
  ```

  - cpuset, io, and hugetlb are not available in grp1

- In grp1, none of the available controllers is initially enabled, so no controllers are available at next level:

  ```
  # cat grp1/cgroup.controllers
cpu memory pids
  # cat grp1/cgroup.subtree_control  # Empty
  # mkdir grp1/{grp10,grp11}  # Make grandchild cgroups
  # cat grp1/grp2/cgroup.controllers  # Empty
  ```
Managing controllers to differing levels of granularity

- If we enable `cpu` in `grp1`, it becomes available at next level

```
# echo '+cpu' > grp1/cgroup.subtree_control
# cat grp1/grp10/cgroup.controllers
cpu
```

- And `cpu` interface files appear in `grp1/{grp10,grp11}`

- Here, `cpu` is being managed at finer granularity than `memory`
  - We can make distinct `cpu` allocation decisions for processes in `grp10` vs processes in `grp11`
  - But we can’t make distinct `memory` allocation decisions
    - `grp10` and `grp11` will share `memory` allocation from `grp1`

- **We did this by design** (so we can manage different resources to different levels of granularity):
  - We want distinct CPU allocations in `grp10` and `grp11`
  - We want `grp10` and `grp11` to share a memory allocation

Top-down constraints

- Child cgroups are always subject to any resource constraints established by controllers in ancestor cgroups
  - ⇒ 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
Cgroups v2 enforces a rule often expressed as: “a cgroup can’t have both child cgroups and member processes”

- I.e., only leaf nodes can have member processes
- The “no internal tasks” rule

But the rule can be expressed more precisely...

- A cgroup can’t both:
  - distribute a resource to child cgroups (i.e., enable controllers in `cgroup.subtree_control`), and
  - 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...
- if 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
No internal tasks rule

Further details on the no internal tasks rule:

- The root cgroup is (necessarily) an exception to this rule
- The rule is irrelevant for “automatic” controllers
  - Because those controllers (e.g., freezer, devices) are always available (i.e., don’t need to be enabled)
- ⚠️ The rule changes for certain controllers in Linux 4.14
  - (The so-called “threaded controllers”)

---

Exercises

This exercise demonstrates that resource constraints apply in a top-down fashion, using the cgroups v2 `pids` controller.

- Check that the `pids` controller is visible in the cgroup root `cgroup.controllers` file. If it is not, reboot the kernel as described on slide 16-15.

- 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 will be `/sys/fs/cgroup`, or possibly `/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:

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

[Exercise continues on next page...]
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
```

In another terminal, use the supplied `cgroups/fork_bomb.c` program.

```
fork_bomb <num-children> [<child-sleep>]
# Default: 0 300
```

Run the program with the following command line, which (after the user presses `Enter`) will cause the program to create 30 children that sleep for (the default) 300 seconds:

```
$ ./fork_bomb 30
```

[Exercise continues on next page...]

---

The parent process in the `fork_bomb` program prints its PID. Return to the first terminal and place the parent process in the grandchild `pids` cgroup:

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

In the second terminal window, press `Enter`, so that the parent process now creates the child processes. How many children does it successfully create?

This exercise demonstrates what happens if we try to enable a controller in a cgroup that has member processes.

Under the cgroup root directory, create a new cgroup named `child`, and enable the `memory` controller in the root cgroup:

```
# cd /sys/fs/cgroup # or: cd /sys/fs/cgroup/unified
# mkdir child
# echo '+memory' > cgroup.subtree_control
```

[Exercise continues on the next slide]
Exercises

- Start a process running `sleep`, and place the process into the child cgroup:

  ```sh
  # sleep 1000 &
  # echo $! > child/cgroup.procs
  ```

- What happens if we now try to enable the `memory` controller in the child cgroup via the following command?

  ```sh
  # echo '+memory' > child/cgroup.subtree_control
  ```

- Does the result differ if we reverse the order of the preceding steps (i.e., enable the controller, then place a process in the cgroup)?