NDC TechTown

An introduction to control groups (cgroups) v2

Michael Kerrisk, man7.org

mtk@man7.org

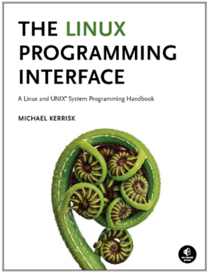
20 October 2021, Kongsberg, Norway

Outline

Outline

Who am I?

- Maintainer of Linux *man-pages* project since 2004
  - ≈1060 pages, mainly for system calls & C library functions
    - https://www.kernel.org/doc/man-pages/
    - (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:
  - What are control groups?
  - An example (`pids` controller)
  - A survey of the controllers
  - Enabling and disabling controllers
  - Managing controllers to different levels of granularity
- Questions: at the end

Outline

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
- So, let's ignore v1 and focus on v2

Booting to cgroups v2

- You may be on a distro that uses cgroups v1 by default; if so, you need to reboot....
  - Because we can't simultaneously use a controller in both v1 and 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
  - ⇒ *systemd* abandons its "hybrid" mode, uses just v2
    - (Hybrid mode uses a mixture of cgroups v1 and v2)

The cgroup2 filesystem

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

Outline

What are control groups?

- Two principle 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

Filesystem 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)

Outline

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:

```
# 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)
- 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)

Discovering a process's cgroup membership

- `/proc/PID/cgroup` shows cgroup membership(s) of a process:

```
$ cat /proc/17273/cgroup
0::/mygrp
```

  - Membership is shown as pathname relative to mount point
  - `0::` is entry for cgroup v2 hierarchy
    - (In *systemd*'s hybrid mode, we would also see entries for memberships in v1 hierarchies)

Destroying a cgroup

- A cgroup that has no child cgroups and no member processes can be destroyed by removing directory
- Returning to our first shell:

```
# rmdir mygrp
rmdir: failed to remove 'mygrp/': Device or resource busy
# echo $$ > /sys/fs/cgroup/cgroup.procs      # Move to root cgroup
# rmdir mygrp                                # Succeeds
```

  - First attempt failed because shell is a member of cgroup we are trying to remove
  - So, we move shell to root cgroup and repeat
  - **Note**: it is not necessary (or possible!) to delete files inside directory beforehand

Outline

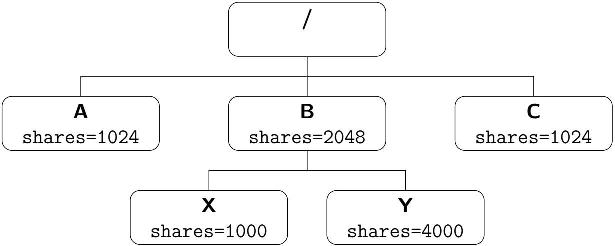
Cgroups v2 controllers

- Let's get a flavor of what kinds of control are possible
- `Documentation/admin-guide/cgroup-v2.rst` documents v2 controllers

Controllers available in cgroups v2

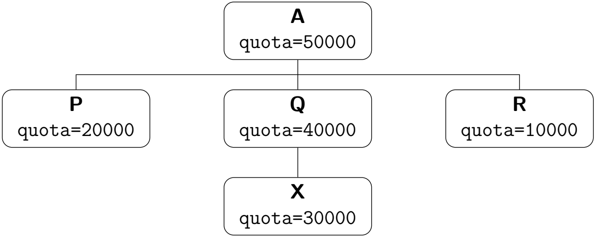
- `cpu`: limit and measure CPU usage by a group of processes; two modes of operation:
  - **Proportional-weight division** (default)
  - **Bandwidth control**
  - Can intermingle these modes at different levels in hierarchy

cpu controller: proportional-weight division



- `cpu.weight` file in each group defines relative share of CPU received by that group
- Processes in B get $\frac{2048}{1024+2048+1024} = \frac{1}{2}$ of CPU time
- Processes in A and C each get $\frac{1024}{1024+2048+1024} = \frac{1}{4}$ of CPU time
- Processes in X get $\frac{2048}{1024+2048+1024} \cdot \frac{1000}{1000+4000} = \frac{1}{2} \cdot \frac{1}{5} = \frac{1}{10}$ of CPU time
- Processes in Y get $\frac{2048}{1024+2048+1024} \cdot \frac{4000}{1000+4000} = \frac{1}{2} \cdot \frac{4}{5} = \frac{4}{10}$ of CPU time

cpu controller: bandwidth control



- Bandwidth control strictly limits CPU (`quota/period`) granted to a group (even if no other competitors for CPU)
- Assume that `period` is 100’000 in all cgroups
- Processes under A will get maximum of 50% of (one) CPU
- Processes under Q will get maximum of 40% of CPU
- Processes under X will get maximum of 30% of CPU
- Sibling cgroups under A are oversubscribed (won’t get 70% of CPU)

Controllers available in cgroups v2

- `cpuset`: control CPU and memory affinity
  - Pin cgroup to one CPU/subset of CPUs (or memory nodes)
  - Dynamically manage placement of application components on systems with large numbers of CPUs and memory nodes
    - Non-uniform memory access (NUMA) systems

Controllers available in cgroups v2

- `memory`: limit memory usage per cgroup + memory usage accounting
  - **Soft** limits influence page reclaim under memory pressure
  - **Hard** limits trigger per-cgroup OOM killer
  - Alternatively, can arrange for notifications to user-space supervisor process in event of low-memory situation
- `io`: limit I/O on block devices
  - HDDs, SSDs, USB, etc.
  - Policies:
    - Proportional-weight division of device bandwidth
    - Bandwidth control (throttling/hard limit)
    - Can set up per-device policies

Controllers available in cgroups v2

- `devices`: limitwhich devices members of cgroup may access
    - No interfaces files; instead control is done by attaching eBPF program to cgroup
        - Each attempt to open/create a device is gated by decision that eBPF program returns to kernel
    - Example use: inside container, disallow access to devices other than `/dev/{null,zero,full,random,tty}`
- Control of network traffic
    - *iptables* allows eBPF filters that hook on cgroup v2 pathnames to manage NW traffic on a per-cgroup basis

Controllers available in cgroups v2

- `pids`: limit number of PIDs in cgroup
  - Prevent fork bombs
- `freezer`: freeze (suspend) and thaw (resume) a group of processes
  - Useful for container migration and checkpoint/restore
- And the rest:
  - `perf_event`: carry out per-cgroup *perf* monitoring
    - Allows *perf* monitoring of a container...
  - `rdma`: control use of RDMA resources per cgroup
  - `hugetlb`: limit usage of “huge pages” per cgroup

Outline

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:

```
$ cat /sys/fs/cgroup/cgroup.controllers
cpuset cpu io memory hugetlb pids
```

- A controller may not be available because:
  - The same controllers is **already in use in cgroups v1**
    - Cgroups v1 and v2 can coexist, but a controller can be used in only one version
    - Must unmount controller in v1 (often easier to reboot...)
  - The controller is **not enabled in the parent cgroup**
- Certain so-called **implicit controllers** are always available, and are not listed in cgroup.controllers
  - E.g., freezer, perf_event

man7.org

Enabling controllers: cgroup.subtree_control

- cgroup.subtree_control is used to show or modify the set of controllers that are available in a cgroup:

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

- 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 '+cpu' > cgroup.subtree_control  # Enable 'cpu' controller
# cat cgroup.subtree_control
cpu memory pids
# echo '-cpu' > cgroup.subtree_control  # Disable 'cpu' controller
# cat cgroup.subtree_control
memory pids
```

Enabling controllers: `cgroup.subtree_control`

- 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 into child cgroups
  - This is a significant difference from cgroups v1

cgroup.subtree_control example

- Currently, cpu controller is not enabled in root cgroup:

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

- Create child cgroup and list cpu.* files:

```
# mkdir grp1
# ls grp1/cpu.*
grp1/cpu.pressure  grp1/cpu.stat
```

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

- Enabling cpu controller in parent cgroup causes controller interface files to appear in child cgroup:

```
# echo '+cpu' > cgroup.subtree_control
# ls grp1/cpu.*
grp1/cpu.max       grp1/cpu.stat    grp1/cpu.weight.nice
grp1/cpu.pressure  grp1/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:

```
# echo '50000 100000' > grp1/cpu.max
```

- In another window, we start a program that burns CPU time and displays statistics; and we move it into grp1:

```
# echo 6445 > grp1/cgroup.procs    # 6445 is PID of burner process
```

- In the other terminal, we see:

```
$ ./cpu_burner
[6445] 1: elapsed/cpu = 1.001; %CPU = 99.862
[6445] 2: elapsed/cpu = 1.002; %CPU = 99.835
...
[6445] 6: elapsed/cpu = 1.197; %CPU = 83.522
[6445] 7: elapsed/cpu = 2.000; %CPU = 50.000
[6445] 8: elapsed/cpu = 2.000; %CPU = 50.000
...
```

Outline

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
cpu memory pids
```

  - 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
```

man7.org

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 (we don't want to manage every resource to same level of granularity):
  - We want distinct CPU allocations in grp10 and grp11
  - We want grp10 and grp11 to share a memory allocation

man7.org

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/