System Programming for Linux Containers

Control Groups (cgroups)

Michael Kerrisk, man7.org © 2020

mtk@man7.org

February 2020

Outline

19  Cgroups  19-1
19.1  Introduction to cgroups v1 and v2  19-3
19.2  Cgroups v1: hierarchies and controllers  19-17
19.3  Cgroups v1: populating a cgroup  19-24
19.4  Cgroups v1: release notification  19-33
19.5  Cgroups v1: a survey of the controllers  19-43
19.6  Cgroups /proc files  19-65
19.7  Cgroup namespaces  19-68
Goals

- Cgroups is a big topic
  - Many controllers
  - V1 versus V2 interfaces
- Our goal: understand fundamental semantics of cgroup filesystem and interfaces
  - Useful from a programming perspective
    - How do I build container frameworks?
    - What else can I build with cgroups?
  - And useful from a system engineering perspective
    - What’s going on underneath my container’s hood?
Focus

- We’ll focus on:
  - General principles of operation; goals of cgroups
  - The `cgroup` filesystem
  - Interacting with the `cgroup` filesystem using shell commands
  - Problems with cgroups v1, motivations for cgroups v2
  - Differences between cgroups v1 and v2
- We’ll look **briefly** at some of the controllers

Resources

- Kernel Documentation files
  - `Documentation/cgroup-v1/*.txt`
  - `Documentation/admin-guide/cgroup-v2.rst`
- `cgroups(7)` man page
- Neil Brown’s excellent (2014) LWN.net series on Cgroups: [https://lwn.net/Articles/604609/](https://lwn.net/Articles/604609/)
  - Thought-provoking commentary on the meaning of grouping and hierarchy
- [https://lwn.net/Articles/484254/](https://lwn.net/Articles/484254/) – Tejun Heo’s initial thinking about redesigning cgroups
- Other articles at [https://lwn.net/Kernel/Index/#Control_groups](https://lwn.net/Kernel/Index/#Control_groups)
2006/2007, “Process Containers”
   - Developed by engineers at Google
   - 2007: renamed “control groups” to avoid confusion with alternate meaning for “containers”

January 2008: initial release in mainline kernel (Linux 2.6.24)
   - Three resource controllers in initial mainline release

Fast-forward a few years…
   - Many new resource controllers added

Various problems arose from haphazard/uncoordinated development of cgroup controllers
   - “Design followed implementation” :-(

Sep 2012: work begins on cgroups v2
   - In-kernel changes, but marked experimental
   - Changes were necessarily incompatible with cgroups v1
     ∴ Create new/orthogonal filesystem interface for v2

March 2016, Linux 4.5: cgroups version 2 becomes official
   - Older version (cgroups v1) remains
     - A.k.a. “legacy cgroups”, but not going away in a hurry

Oct 2019: Fedora 31 is first distro to switch to v2-by-default
   - Boot with `systemd.unified_cgroup_hierarchy=0` to revert to v1/v2 “hybrid” mode

Cgroups v2 work is ongoing
   - For now, some functionality remains available only via v1
   - Conversely, v2 offers a number of advantages over v1
     - Subject to some rules, can use both versions at same time
**Cgroups overview**

- Two principle components:
  - A **mechanism for hierarchically grouping** processes
  - A set of **controllers** (kernel components) that manage, control, or monitor processes in cgroups
    - (Resources such as CPU, memory, block I/O bandwidth)
- 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 percentage of CPU available to group
- Prioritize group for resource allocation
  - E.g., some group might get greater proportion of CPU
- Resource accounting
  - Measure resources used by processes
- Freeze a group
  - Freeze, restore, and checkpoint a group
- And more...
Terminology and semantics

- **Control group**: group of processes bound to set of parameters or limits
- **(Resource) controller**: kernel component that controls or monitors processes in a cgroup
  - E.g., `memory` controller limits memory usage; `cpuacct` accounts for CPU usage
  - Also known as **subsystem**
    - (But that term is rather ambiguous)
- Child cgroups **inherit attributes** from parent

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)

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:
Example: the pids controller (cgroups v1)

- **pids** ("process number") controller allows us to limit number of PIDs in cgroup
  - Prevent *fork()* bombs!
- Use *mount* to attach **pids** controller to cgroup filesystem:
  ```
  # mkdir -p /sys/fs/cgroup/pids  # Create mount point
  # mount -t cgroup -o pids none /sys/fs/cgroup/pids
  
  "⚠️ May not be necessary
  - Some systems automatically mount filesystems with controllers attached
    - Specifically, *systemd* mounts the v1 controllers under subdirectories of `/sys/fs/cgroup`, a *tmpfs* filesystem mounted via:
      ```
      # mount -t tmpfs tmpfs /sys/fs/cgroup
      ```

Example: the pids controller (cgroups v1)

- Create new cgroup, and place shell's PID in that cgroup:
  ```
  # mkdir /sys/fs/cgroup/pids/g1
  # echo $$
  17273
  # echo $$ > /sys/fs/cgroup/pids/g1/cgroup.procs
  
  "cgroup.procs" defines/displays PIDs in cgroup
  - Which processes are in cgroup?
    ```
    # cat /sys/fs/cgroup/pids/g1/cgroup.procs
    17273
    20591
    ```
  - Where did PID 20591 come from?
    - PID 20591 is *cat* command, created as a child of shell
      - Child processes inherit parent’s cgroup membership(s)
Example: the pids controller (cgroups v1)

- Limit number of processes in cgroup, and show effect:

```
# echo 20 > /sys/fs/cgroup/pids/g1/pids.max
# for a in $(seq 1 20); do sleep 20 & done
...
[1] 20938
[18] 20955
bash: fork: retry: Resource temporarily unavailable
```

- `pids.max` defines/exposes limit on number of PIDs in cgroup

Applications

Cgroups (v1) is used in a range of applications

- Container frameworks such as Docker and LXC
- Firejail
- Flatpak
- `systemd` (also knows about cgroups v2)
- and more...
### Cgroup hierarchies

- **Cgroup** == collection of processes
- **Cgroup hierarchy** == hierarchical arrangement of cgroups
  - Implemented via a cgroup pseudo-filesystem
  - Structure and membership of cgroup hierarchy is defined by:
    - Mounting a cgroup 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
Attaching a controller to a hierarchy

- A controller is attached to a hierarchy by mounting a cgroup filesystem:

```bash
# mkdir -p /sys/fs/cgroup/pids # Create mount point
# mount -t cgroup -o pids none /sys/fs/cgroup/pids
```

- Here, `pids` controller was mounted
- `none` can be replaced by any suitable mnemonic name
  - Not interpreted by system, but appears in `/proc/mounts`

```
To see which cgroup filesystems are mounted and their attached controllers:

```bash
# mount | grep cgroup
none on /sys/fs/cgroup/pids type cgroup (rw,pids)
# grep cgroup /proc/mounts
none /sys/fs/cgroup/pids cgroup rw,...,pids 0 0
```

- Unmounting filesystem detaches the controller:

```bash
# umount /sys/fs/cgroup/pids
```

- But..., filesystem will remain (invisibly) mounted if it contains child cgroups
  - I.e., must move all processes to root cgroup, and remove child cgroups, to truly unmount
**Attaching controllers to hierarchies**

- A controller can be **attached to only one hierarchy**
  - Mounting same controller at different mount point simply creates second view of same hierarchy
- **Multiple** controllers can be attached to same hierarchy:
  ```
  # mkdir -p /sys/fs/cgroup/mem_cpu
  # mount -t cgroup -o memory,cpu none /sys/fs/cgroup/mem_cpu
  ```
  - In effect, resources associated with those controllers are being managed together
- Or, **all** controllers can be attached to one hierarchy:
  ```
  # mount -t cgroup -o all none /some/mount/point
  ```
  - `-o all` is the default if no controller is specified

---

**Creating cgroups**

- When a new hierarchy is created, all **tasks** on system are part of **root cgroup** for that hierarchy
- New cgroups are **created** by creating subdirectories under cgroup mount point:
  ```
  # mkdir /sys/fs/cgroup/memory/g1
  ```
- Relationships between cgroups are reflected by creating nested (arbitrarily deep) subdirectory structure
  - Meaning of hierarchical relationship depends on controller
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
To move a process to a cgroup, we write its PID to `cgroup.procs` file in corresponding subdirectory:

```
# echo $$ > /sys/fs/cgroup/memory/g1/cgroup.procs
```

- In multithreaded process, moves all threads to cgroup...
- ⚠️ Can write only one PID at a time
  - `write()` fails with `EINVAL`
- Writing 0 to `cgroup.procs` moves writing process to cgroup
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

- Within a hierarchy, 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
- A process can be a member of multiple cgroups, each of which is in a different hierarchy
- On `fork()`, **child inherits cgroup memberships** of parent
  - Afterward, cgroup memberships of parent and child can be independently changed
Placing a thread (task) in a cgroup

- Writing a PID to `cgroup.procs` moves all threads in thread group to a cgroup
- Cgroups v1 also supports notion of thread-level granularity for cgroup membership
  - I.e., individual threads in a multithreaded process can be placed in different cgroups
  - ⇒ threads can be subject to different control settings
- Each cgroup directory also has a `tasks` file...
  - Writing a thread ID (TID) to `tasks` moves that thread to cgroup
    - Thread ID == kernel thread ID (displayable with `ps -L`)
  - Reading `tasks` shows all TIDs in cgroup

Tasks?

- Cgroups v1 draws distinction between process and task
  - Task == kernel scheduling entity
    - From scheduler’s perspective, “processes” and “threads” are pretty much the same thing....
    - (Threads just share more state than processes)
  - Multithreaded process == set of tasks with same thread group ID (TGID)
    - TGID == PID!
    - Each thread has unique thread ID (TID)
  - Here, TID means kernel thread ID
    - I.e., value returned by `clone(2)` and `gettid(2)`
      - And displayed (as “LWP”) by `ps -L`
    - Not same as POSIX threads `pthread_t`
      - (But there is 1:1 relationship in NPTL implementation...)
Exercises

(If you have a recent distro that defaults to cgroups v2 only, reboot with
systemd.unified_cgroup_hierarchy=0 to revert to ‘‘hybrid’’ mode.)

In this exercise, we create a cgroup, place a process in the cgroup, and
then migrate that process to a different cgroup.

- If the memory cgroup is not already mounted, mount it:

```
# grep 'cgroup.*mem' /proc/mounts  # Is cgroup mounted?
# mkdir -p /sys/fs/cgroup/memory
# mount -t cgroup -o memory none /sys/fs/cgroup/memory
# cd /sys/fs/cgroup/memory
```

- Note: some systems (e.g., some Debian releases) provide a
  patched kernel that disables the memory controller by
default. If you can’t mount the controller, it may be
necessary to reboot with the cgroup_enable=memory
kernel command-line option. Alternatively, you could use a
different controller for this exercise.

[Exercise continues on the next slide]

Create two subdirectories, m1 and m2, in the memory 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?
- Remove the cgroups m1 and m2 using the rmdir command.