LinuxCon Europe 2016

Control Groups (cgroups)

man7.org Training and Consulting
http://man7.org/training/
@mkerrisk   mtk@man7.org

4 October 2016
Berlin, Germany

Outline

Who am I?

- Maintainer of Linux man-pages (since 2004)
  - Documents kernel-user-space + C library APIs
    - ~1000 manual pages
    - http://www.kernel.org/doc/man-pages/
- API review, testing, and documentation
  - API design and design review
  - Lots of testing, lots of bug reports, a few kernel patches
- "Day job": programmer, trainer, writer
  - http://man7.org/

Outline

Goals

- Cgroups is a big topic
  - Many controllers
  - V1 versus V2 interfaces
- Our goal: understand fundamental semantics of cgroup filesystem and interfaces
  - ("Containers are too high level for me")
  - 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/cgroup-v2.txt`
- *cgroups(7)* man page
- Neil Brown's excellent (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*

History

- 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)
- Fast-forward a few years...
  - Many new resource controllers added
- Various problems arose from haphazard/uncoordinated development of cgroup controllers
  - "Design followed implementation" :-(

History

- Sep 2012: work begins on cgroups v2
  - In-kernel changes, but marked experimental
  - Changes were necessarily incompatible with cgroups v1
    - $\Rightarrow$ 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
- Cgroups v2 work is ongoing
  - For now, some functionality remains available only via cgroups 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** 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
  - E.g., can use shell commands

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)
- Cgroups for each controller can be arranged in a **hierarchy**
  - Child cgroups may **inherit attributes** 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 for purposes such as:
  - **Defining/displaying membership** of cgroup
  - **Controlling behavior** of processes in cgroup
  - **Exposing information** about processes in cgroup (e.g., resource usage stats)

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
    - E.g., *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

Outline

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:
  1. **Mounting** a `cgroup` filesystem
  2. **Creating a subdirectory structure** that reflects desired cgroup hierarchy
  3. **Moving processes within hierarchy** by writing their PIDs to special files in cgroup subdirectories

Attaching a controller to a hierarchy

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

```
# mkdir -p /sys/fs/cgroup/mem   # Create mount point
# mount -t cgroup -o memory none /sys/fs/cgroup/mem
```

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

Attaching a controller to a hierarchy

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

```
# mount | grep cgroup
none on /sys/fs/cgroup/mem type cgroup (rw,memory)
# cat /proc/mounts | grep cgroup
none /sys/fs/cgroup/mem cgroup rw,relatime,memory 0 0
```

- Unmounting filesystem detaches the controller:

```
# umount /sys/fs/cgroup/mem
```

  - 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

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/mem/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
  - Not necessary (or possible) to delete attribute files inside cgroup directory before deleting it

Outline

Files for managing cgroup membership

- To manage cgroup membership, each subdirectory in a hierarchy includes two automagically created files:
  - `cgroup.procs`
  - `tasks`

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 (MT) 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)*
  - Not same as POSIX threads `pthread_t`
    - (But there is 1:1 relationship in NPTL implementation...)

Placing a process in a cgroup

- To move a **process** to a cgroup, write its PID to `cgroup.procs` file in corresponding subdirectory

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

Viewing cgroup membership

- **To see PIDs in cgroup**, read `cgroup.procs` file
  - PIDs are newline-separated
- ⚠ 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

Placing a thread (task) in a cgroup

- Writing a PID to `cgroup.procs` **moves all threads in thread group** to a cgroup
- Each cgroup directory also has a `tasks` file...
  - Writing a TID to **`tasks`** **moves that thread** to cgroup
    - This feature goes away in cgroups v2...
  - Reading `tasks` shows all TIDs in cgroup

Cgroup membership details

- Within a hierarchy, a **task can be member of just one cgroup**
  - That association defines attributes / parameters that apply to the task
- Adding a task to a different cgroup automatically removes it from previous cgroup
- A task 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

Cgroup release

- Consider the following scenario:
  - We create a cgroup subdirectory
  - Some processes are moved into cgroup
  - Eventually, all of those processes terminate
- Who cleans up/gets notified when last process leaves cgroup?
  - We might want cgroup subdirectory to be removed
  - Manager process might want to know when all workers have terminated

Cgroup release

- `release_agent` in cgroup **root** directory
  - Contains pathname of binary/script that is executed **when cgroup becomes empty**
    - E.g., this program might remove cgroup subdirectory
  - Release agent gets one **command-line argument**: pathname of cgroup subdirectory that has become empty
- `notify_on_release` in each cgroup subdirectory
  - Should `release_agent` be run when cgroup becomes empty? (0 == no, 1 == yes)
  - Initial setting for this file is inherited from cgroup parent

Mounting a *named* hierarchy with no controller

- Can mount a *named* hierarchy with no attached controller:

```
# mount -t cgroup cgroup -o none,name=somename \
           /some/mount/point
```

- Named hierarchies can be used to organize and track processes
  - E.g., PIDs can be moved into `cgroup.procs`, and will automatically disappear on process termination
    - (And we can use `release_agent`, etc.)
  - *systemd* creates such a hierarchy for its management of processes
    - Mounted at `/sys/fs/cgroup/systemd`
- Cgroups v1 only

Exercises

1. 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:

```
# cat /proc/mounts | grep cgroup   # 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., Debian) provide a patched kernel that disables the `memory` controller by default. If you find that you can't mount the memory controller, it may be necessary to reboot the kernel with the `cgroup_enable=memory` command-line option.
   - 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.

Outline

Cgroups v1 controllers

- Each of following controllers is selectable via a **kernel configuration option**
  - And there is an overall option, `CONFIG_CGROUPS`
- For each controller, there are controller-specific files in each cgroup directory
  - Names are prefixed with controller-specific string
    - E.g., `cpuacct.stat`, `pids.max`, `freezer.state`
- Individual **documentation** files for most controllers can be found in `Documentation/cgroup-v1`
  - $\Rightarrow$ Following slides give just a brief picture...

Cgroups v1 controllers

- cpu (2.6.24): control distribution of CPU cycles to cgroups
  - cpu.cfs_period_us: measurement period for CFS scheduler (μs; default: 100000)
  - cpu.cfs_quota_us: allowed run-time within period (μs; default: –1 [no limit])
  - Constraints propagate into child cgroups
- cpuacct (2.6.24): expose CPU usage of cgroup
  - cpuacct.usage: CPU usage by this cgroup (nanoseconds)
  - cpuacct.stat: user vs system mode CPU time (measured in USER_HZ [centiseconds])
  - Statistics include CPU consumed in descendant cgroups

Cgroups v1 controllers

- `memory` (2.6.25): control memory usage of cgroups
  - Limit memory usage per cgroup
    - Soft limits influence page reclaim under memory pressure
    - Hard limits trigger per-cgroup OOM killer
  - Memory-usage accounting (optionally hierarchical)
  - Disable knob for OOM killer
  - Kernel-to-user-space notification for low-memory and OOM situations
    - E.g., instead of OOM killing, freeze processes, notify user space, remedy situation, thaw processes
  - And more; see `Documentation/cgroup-v1/memory.txt`
    - (but "this document is hopelessly outdated")

Cgroups v1 controllers

- `freezer` (2.6.28): freeze (suspend) and resume processes in a cgroup
  - Gets round some limitations of using `SIGSTOP`/`SIGCONT` for this purpose
    - `SIGSTOP` is observable by waiting/ptracing parent
    - `SIGCONT` can be caught by application!
  - Cgroup is frozen / resumed by writing `FROZEN` / `THAWED` to `freezer.state`
    - Operations propagate to child cgroups
- `blkio` (2.6.33): limit I/O on block devices
  - HDDs, SSDs, USB, etc.
  - Policies:
    - Proportional-weight division of device bandwidth
    - Throttling/upper-limit

Cgroups v1 controllers

- `pids` (4.3): limit number of tasks in a cgroup
  - Prevent fork bombs
  - `pids.max`: maximum number of tasks in cgroup (and cgroup descendants)
    - Writing "max" into this file means no limit
    - Limit affects *fork()*/*clone()*
    - Doesn't affect attempts to move processes into cgroup
  - `pids.current`: number of PIDs currently in cgroup
  - ⚠ `pids.current` & `pids.max` **count tasks** not processes
  - Limit on a cgroup == most stringent limit on any ancestor cgroup (and descendants)

Other cgroups v1 controllers

- `cpuset` (2.6.24): assign CPUs & memory nodes to cgroups
- `devices` (2.6.26): whitelist controller to permit/deny access to device by members of cgroup
- `perf_event` (2.6.39): carry out *perf* monitoring per cgroup
- `net_cls` (2.6.29), `net_prio` (3.3): traffic shaping and priority control of cgroup's network traffic
- `hugetlb` (3.6): limit hugeTLB usage per cgroup

Exercises

1. The `freezer` controller can be used to suspend and resume execution of all of the processes in a cgroup hierarchy. Create a cgroup hierarchy containing two child cgroups (thus three cgroups in total) as follows:

```
# mkdir /sys/fs/cgroup/freezer/mfz
# mkdir /sys/fs/cgroup/freezer/mfz/sub1
# mkdir /sys/fs/cgroup/freezer/mfz/sub2
```

   Then run four separate instances of the `timers/cpu_burner.c` program, and place two of the resulting processes in the `mfz/sub1` cgroup, and one each of the remaining processes in `mfz` and `mfz/sub2`. Observe what happens to these processes as the following commands are executed.
   Freeze the processes in the `mfz/sub1` cgroup:

```
# echo FROZEN > /sys/fs/cgroup/freezer/mfz/sub1/freezer.state
```

   Freeze all of the processes in all cgroups under the `mfz` subtree:

```
# echo FROZEN > /sys/fs/cgroup/freezer/mfz/freezer.state
```

Exercises

Thaw all of the processes in the `mfz` subtree, so that they resume execution:

```
# echo THAWED > /sys/fs/cgroup/freezer/mfz/freezer.state
```

Once more freeze the entire subtree, and then try thawing just the processes in the `mfz/sub1` cgroup:

```
# echo FROZEN > /sys/fs/cgroup/freezer/mfz/freezer.state
# echo THAWED > /sys/fs/cgroup/freezer/mfz/sub1/freezer.state
```

Do the processes in the `mfz/sub1` cgroup resume execution? Why not? For a clue, view the status of the cgroup parent of this cgroup using the following command:

```
# cat /sys/fs/cgroup/freezer/mfz/sub1/freezer.parent_freezing
```

Outline

/proc/cgroups file

- /proc/cgroups describes controllers available on system

```
#subsys_name    hierarchy       num_cgroups     enabled
cpuset          4               1               1
cpu             8               1               1
cpuacct         8               1               1
blkio           6               1               1
memory          3               1               1
devices         10              84              1
freezer         7               1               1
net_cls         9               1               1
perf_event      5               1               1
net_prio        9               1               1
hugetlb         0               1               0
pids            2               1               1
```

1. Controller name
2. Unique hierarchy ID (0 for v2 hierarchy)
   - Multiple controllers may be bound to same hierarchy
3. Number of cgroups in hierarchy
4. Controller enabled? 1 == yes, 0 == no
   - Kernel cgroup_disable boot parameter

/proc/PID/cgroup file

- /proc/PID/cgroup shows cgroup memberships of PID

```
3:cpu,cpuacct:/memgrp3
2:freezer:/
0::/grp1
```

1. Hierarchy ID (0 for v2 cgroup)
    - Can be matched to hierarchy ID in /proc/cgroups
2. Comma-separated list of controllers bound to the hierarchy
    - Field is empty for v2 cgroup
3. Pathname of cgroup to which this process belongs
    - Pathname is relative to cgroup root directory

Outline

Cgroups version 2

- Designed to address perceived problems with cgroups v1
  - Section "R" of `Documentation/cgroup-v2.txt` details the problems
- Cgroups v2 officially released in Linux 4.5 (March 2016)
  - After extended experimental development phase...
- Both cgroups v1 and cgroups v2 can be used on same system
  - **But** can't mount same controller in both filesystems

Cgroups v2 controllers

- V2 currently implements only a subset of equivalents of v1 controllers
  - Work in progress...
- `Documentation/cgroup-v2.txt` documents v2 controllers
  - `memory`: control distribution of memory
    - Successor of v1 `memory` controller
  - `io`: regulate distribution of I/O resources
    - Successor of v1 `blkio` controller
  - `pids`: control number of processes
    - Exactly the same as v1 `pids` controller
  - ⚠ `cpu`: documented in `Documentation/cgroup-v2.txt`, but not yet merged (as at Linux 4.8)
  - `freezer`: work in progress (late 2016/early 2017?)

Problems with cgroups v1

- **V1 hierarchy scheme** was supposed to allow great flexibility
  - V1: arbitrary number of hierarchies, with each hierarchy hosting any number of controllers
- But, that flexibility was **less useful than originally envisaged**

Problems with cgroups v1

Problems with the v1 hierarchy scheme:

- ☹ Utility controllers (e.g., `freezer`) that might be useful in all hierarchies could be used in only one
- ☹ Controllers bound to same hierarchy were forced to have same hierarchical view
  - Could not vary granularity according to controller
- These problems meant apps commonly put most controllers on separate, but highly similar, hierarchies
  - ☹ Same hierarchical management operations needed to be repeated on multiple hierarchies
  - ☹ Cooperation between controllers becomes complex
- ⇒ **v2 uses single hierarchy for all controllers**
  - Establish common resource domain across different resource types, so controllers (e.g., `memory` and `io`) can cooperate

Problems with cgroups v1

Allowing **thread-granularity** for cgroup membership proved problematic

- Didn't make sense for some controllers
  - E.g., `memory` controller (all threads share memory...)
- Writing TIDs to `tasks` file is a **system-level activity**, but only **applications** well understand their thread topology
- $\Rightarrow$ **v2 allows only process-granularity** membership

Problems with cgroups v1

- There may yet be some backtracking on process-vs-thread granularity for `cpu` controller
  - Some users are pushing back strongly for thread granularity
- Further info
  - "Resource groups"; https://lwn.net/Articles/656115/
    https://lwn.net/Articles/679940/
    https://lkml.org/lkml/2016/1/5/366

    https://lwn.net/Articles/697369/ ("[Documentation] State of CPU controller in cgroup v2", Aug 2016)
    https://lwn.net/Articles/697366/ "The case of the stalled CPU controller"

Problems with cgroups v1

- Allowing a cgroup to contain both tasks and child cgroups is problematic
  - Two different types of entities–*tasks* and *groups* of tasks–compete for distribution of same resources
    - Different controllers dealt with this in differing ways...
    - which could cause difficulties if trying to generically combine multiple controllers on same hierarchy
  - $\Rightarrow$ **In v2, only leaf cgroups can contain processes**
    - (The story is a little more subtle...)

Problems with cgroups v1

- **Inconsistencies** between controllers (“design followed implementation”)
  - In some hierarchies, new cgroups inherit parent’s attributes; in others, they get defaults
  - Some controllers have controller-specific interfaces in root cgroup; others don’t
  - v2: **consistent names and values** for interface files, **consistent inheritance rules** for all controllers
    - With some clearly documented guidelines!
- V1 cgroup release mechanism (firing up a process) has problems:
  - Firing up a process is expensive
  - Can’t delegate release handling to process inside a container
  - $\Rightarrow$ v2 has a lightweight solution that supports delegation

Outline

Mounting the cgroups v2 filesystem

- To use cgroups v2, we mount new filesystem type:

```
# mount -t cgroup2 none /path/to/mount
```

- All **v2 controllers are automatically available** under single hierarchy
  - No need to explicitly bind controllers to mount point
    - `cgroup2` filesystem allows/needs no `-o` mount options

The cgroup.controllers file

- Each v2 cgroup has a cgroup.controllers file, which lists **available controllers** this cgroup can enable
- But, if we look in cgroups v2 root directory, we might find cgroup.controllers is empty:

```
# mkdir /mnt/cgroup2
# mount -t cgroup2 none /mnt/cgroup2
# cd /mnt/cgroup2
# cat cgroup.controllers
# wc -l cgroup.controllers
0 cgroup.controllers
```

A v2 controller is available only if not mounted in v1

- V2 controller is available only if not bound in v1 hierarchy

```
# cat /proc/mounts | grep pids
cgroup /sys/fs/cgroup/pids cgroup rw,..,pids 0 0
pids
```

  - That's why we didn't see `pids` in v2 `cgroup.controllers`
- ⇒ May need to unmount controller in v1 hierarchy to have it available in v2 hierarchy:

```
# umount /sys/fs/cgroup/pids
# cat cgroup.controllers
pids
```

- (Since Linux 4.6) kernel boot parameter, `cgroup_no_v1`:
  - `cgroup_no_v1=all`, to disable all v1 controllers
  - `cgroup_no_v1=controller,...`, to disable selected v1 controllers

Enabling and disabling controllers

- Controllers are enabled/disabled by writing some subset of available controllers to `cgroup.subtree_control`

```
# echo "+pids -memory" > cgroup.subtree_control
```

  - `+` $\Rightarrow$ enable controller, `-` $\Rightarrow$ disable controller

Enabling and disabling controllers

- 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 across child cgroups
  - Different from v1...

Example: enabling a controller

- In the cgroup root directory, list available controllers:

```
# cat cgroup.controllers
io memory pids
```

- Create a child cgroup; see what files are in subdirectory:

```
# mkdir grp1
# ls grp1
cgroup.controllers  cgroup.events  cgroup.procs
cgroup.subtree_control
```

- Enable pids controller for child cgroups; new control files have been created in child cgroup:

```
# echo '+pids' > cgroup.subtree_control
# ls grp1
cgroup.controllers  cgroup.procs           pids.current
cgroup.events       cgroup.subtree_control  pids.max
```

Example: enabling a controller

- In grp1 cgroup, only available controller is `pids`:

```
# cat grp1/cgroup.controllers
pids
```

- In child of grp1, we can enable `pids` controller:

```
# mkdir grp1/sub
# echo '+pids' > grp1/cgroup.subtree_control
# cat grp1/cgroup.subtree_control
pids
```

- But `io` controller is not available:

```
# echo '+io' > grp1/cgroup.subtree_control
sh: echo: write error: No such file or directory
```

Top-down constraints

- Child cgroups are always subject to any resource constraints established by controllers in ancestor cgroups
  - $\Rightarrow$ 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

Exercises

1. This exercise demonstrates that resource constraints apply in a top-down fashion, using the cgroups v2 `pids` controller.
    - Mount the `cgroup2` filesystem if it is not already mounted and check that the `pids` controller is visible in the cgroup root `cgroup.controllers` file. If it is not, unmount the cgroup v1 `pids` filesystem. (See the steps at the start of this section.)
        - In some cases, unmounting the cgroup v1 `pids` filesystem may not be enough, since the controller is in use (e.g., by *systemd*). Therefore, it may be necessary to reboot the system with the `cgroup_no_v1=pids` kernel boot parameter.
    - To simplify the following steps, change your current directory to the cgroup root directory (i.e., the location where the `cgroup2` filesystem is mounted).

Exercises

- Create a child and grandchild directory in the cgroup filesystem and enable the PIDs controller in the root directory and the first subdirectory:

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

- 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 with the following command line, which will cause the program to first sleep 60 seconds and then create 30 children:

Exercises

```
$ ./fork_bomb 30 60
```

- The parent process in the fork_bomb program prints its PID before sleeping. While it is sleeping, return to the first terminal and place the parent process in the grandchild pids cgroup:

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

- When the parent finishes sleeping, how many children does it successfully create?

Outline

Organizing cgroups and processes

Broadly similar to cgroups v1:

- Hierarchy organized as set of subdirectories
- All processes initially in root cgroup
- Move process into group by writing PID into `cgroup.procs`
- Read `cgroup.procs` to discover process membership
  - ⚠ Returned list is not sorted
  - ⚠ List may contain duplicate PIDs
    - E.g., if PID moved out and then back into cgroup, or PID recycled, while reading
- Child of *fork()* inherits parent's cgroup membership
- Cgroup directory with no process members or child cgroups can be removed

Organizing cgroups and processes

Differences between v1 and v2:

- Cgroup can't both control cgroup children and have member processes
  - $\Rightarrow$ Place member processes in leaf nodes
- No `tasks` file
  - Granularity for cgroup membership is process
  - Writing TID of any thread to `cgroup.procs` moves all of process's threads to cgroup
- Root cgroup does not contain controller interface files

"Only leaf nodes can have member process"

- Earlier statement: cgroup can't have both child cgroups and member processes
- Let's refine that...
- A cgroup can't both:
  - distribute a resource to child cgroups, **and**
  - have child processes
  - (**Note**: root cgroup is an exception to this rule)
- Conversely (1):
  - A cgroup **can** have member processes and child cgroups...
  - **iff** 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

Cgroup (un)populated notification

- Cgroups v1: firing up a process is an expensive way of get notification of an empty cgroup!
- Cgroups v2:dispenses with `release_agent` and `notify_on_release` files
- Instead, each (non-root) cgroup has a file, `cgroup.events`, with a `populated` field:

```
# cat grp1/cgroup.events
populated 1
```

  - 1 == subhierarchy contains live processes
    - I.e., live process in any descendant cgroup
  - 0 == no live processes in subhierarchy

Cgroup (un)populated notification

- Can monitor `cgroup.events` file, to get notification of transition between populated and unpopulated states
  - *inotify*: transitions generate `IN_MODIFY` events
  - *poll()*: transitions generate `POLLPRI` events
- One process can monitor multiple `cgroup.events` files
  - Much cheaper notification!
  - **Notification can be delegated** per container
    - I.e., one process can monitor all `cgroup.events` files in a subhierarchy

Exercise

For the following exercises, you'll need to mount the cgroup2 filesystem if it is not already mounted and check that the pids controller is visible in the cgroup root cgroup.controllers file. If it is not, unmount the cgroup v1 pids filesystem. (Details can be found at the start of section *Cgroups v2: enabling and disabling controllers*.) In the exercises below, we assume that the cgroup2 filesystem is mounted at /mnt/cgroup2.

1. This exercise demonstrates what happens if we try to enable a controller in a cgroup that has member processes.
   - Under the cgroup2 mount point, create a new cgroup, and enable the pids controller in the root cgroup:

```
# cd /mnt/cgroup2
# mkdir child
# echo '+pids' > cgroup.subtree_control
```

Exercise

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

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

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

```
# echo '+pids' > child/cgroup.subtree_control
```

Thanks!

mtk@man7.org @mkerrisk
Slides at http://man7.org/conf/

Linux System Programming,
System Programming for Linux Containers,
and other training at http://man7.org/training/
The Linux Programming Interface, http://man7.org/tlpi/