Open Source Summit Europe 2018

An introduction to control groups (cgroups)

Michael Kerrisk, man7.org © 2018
mtk@man7.org

Open Source Summit Europe
21 October 2018, Edinburgh, Scotland

Outline

Outline

Who am I?

- Contributor to Linux *man-pages* project since 2000
  - Maintainer since 2004
    - Maintainer email: `mtk.manpages@gmail.com`
  - Project provides ≈1050 manual pages, primarily documenting system calls and C library functions
    - *https://www.kernel.org/doc/man-pages/*
- Author of a book on the Linux programming interface
  - *http://man7.org/tlpi/*
- Trainer/writer/engineer
  - Lots of courses at *http://man7.org/training/*
- Email: `mtk@man7.org`
  Twitter: `@mkerrisk`

Who am I?

- Contributor to Linux *man-pages* project since 2000
  - Maintainer since 2004
    - Maintainer email: `mtk.manpages@gmail.com`
  - Project provides ≈1050 manual pages, primarily documenting system calls and C library functions
    - *https://www.kernel.org/doc/man-pages/*
- Author of a book on the Linux programming interface
  - *http://man7.org/tlpi/*
- Trainer/writer/engineer
  - Lots of courses at *http://man7.org/training/*
- Email: `mtk@man7.org`
  Twitter: `@mkerrisk`

Outline

Goals

- Cgroups is a big topic
  - Many controllers
  - V1 versus V2 interfaces
- We can't do everything; we'll focus on:
  - General principles of operation; goals of cgroups
  - The `cgroup` filesystem
  - Interacting with the `cgroup` filesystem using shell commands
- And later...
  - 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

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)
  - 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” :-(

History

- Sep 2012: work begins on cgroups v2
  - 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
- 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** (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)
- Cgroups for each controller are arranged in a **hierarchy**
  - Child cgroups **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 to:
  - **Define/display membership** of cgroup
  - **Control behavior** of processes in cgroup
  - **Expose 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
    - Specifically, *systemd* mounts the v1 controllers under subdirectories of `/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...

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
     - E.g., `cgroup.procs`

Attaching a controller to a hierarchy

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

```
# 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
- Most distros these days use *systemd*, which automatically mounts all cgroups v1 resource controllers during boot-up

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

- Unmounting filesystem detaches the controller:

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

Outline

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/memory/g1/cgroup.procs
```

  - In multithreaded process, moves all threads to cgroup...
- ⚠ Can write only one PID at a time
  - *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

- Within a hierarchy, a **process can be member of just one cgroup**
- 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
  - $\Rightarrow$ **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

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:

```
# 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.
   - Create two subdirectories, `m1` and `m2`, in the `memory` cgroup root directory.
     [Exercise continues on the next slide]

Exercises

- 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.

Outline

Cgroups v1 controllers

- 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 flavor of what controllers are available

Cgroups v1 controllers

- `cpuset` (2.6.24): assign CPUs & memory nodes to cgroups
  - Pin cgroup to one CPU/subset of CPUs (or memory nodes)
  - Reserve a CPU for a high priority application
  - Dynamically manage placement of application components on systems with large numbers of CPUs
    - And systems with non-uniform memory access
- `cpuacct` (2.6.24): expose CPU usage of cgroup
  - `cpuacct.usage`: CPU usage by this cgroup (nanoseconds)
  - Statistics include CPU consumed in descendant cgroups

Cgroups v1 controllers

- `cpu` (2.6.24): control distribution of CPU cycles to cgroups
  - Two modes: **proportional-weight division** and **bandwidth control**
  - Proportional-weight division:
    - `cpu.shares` file defines proportion of CPU given to cgroup
    - Proportion of CPU given to cgroup = (`cpu.shares` / [sum of all `cpu.shares` at same level])
    - Constraints have effect only if there is competition for CPU
  - Bandwidth control:
    - Quotient `cpu.cfs_quota_us` / `cpu.cfs_period_us` defines upper limit on CPU consumption by cgroup
    - Quota applies even if no other competitors for CPU
  - Proportional-weight division is used, unless `cpu.cfs_quota_us` > 0

Cgroups v1 controllers

- `memory` (2.6.25): memory controller
  - 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)
- `devices` (2.6.26): controller that white lists devices that may be accessed by members of a cgroup
  - Can control open for read, open for write, and *mknod*
  - Example use: inside container, allow access to `/dev/{null,zero,random,tty}`, disallow everything else

Cgroups v1 controllers

- `freezer` (2.6.28): freeze (suspend) and resume processes in a cgroup
  - Cgroup is frozen / resumed by writing `FROZEN` / `THAWED` to `freezer.state`
    - Operations propagate to child cgroups
  - Use cases: container migration; checkpoint-restore
- `pids` (4.3): limit number of tasks in a cgroup
  - Prevent fork bombs
  - `pids.max`: writable file that defines limit on number of tasks that can be created in cgroup
  - Tasks in child cgroups count against limit

Cgroups v1 controllers

- `net_cls` (2.6.29): tag outgoing network packets emitted by processes in a cgroup with class ID
  - Class ID can be used by *tc(8)* ("traffic control") for network traffic shaping
- `net_prio` (3.3): control priority of cgroup's outgoing network traffic
  - Can control on per-interface basis (unlike `net_cls` controller)

Cgroups v1 controllers

- `blkio` (2.6.33): limit I/O on block devices (HDDs, SSDs)
  - Policies:
    - Proportional-weight division of device bandwidth
    - Throttling/upper-limit
- `perf_event` (2.6.39): carry out *perf* monitoring per cgroup
  - Do *perf* monitoring of a container...
- `hugetlb` (3.6): limit usage of "huge pages" per cgroup
- `rdma` (4.11): control and accounting of RDMA resources
  - RDMA == remote direct memory access

Exercises

1. The `cpu` controller implements bandwidth-based throttling of CPU usage. Throttling is specified via two files:
   - `cpu.cfs_period_us`: the period used for allocating CPU bandwidth (μsec; default 100000)
   - `cpu.cfs_quota_us`: the portion of the period available to this cgroup (μsec; default -1, meaning no limit)

   Create two sibling CPU cgroups, named `fast` and `slow`. (You might find that the CPU controller is co-mounted with the CPU accounting controller under `/sys/fs/cgroup/cpu,cpuacct`.) In one cgroup, set `cpu.cfs_quota_us` to 30000, and in the other set it to 10000. Run two instances of the `timers/cpu_burner.c` program, which consumes CPU time, printing a message as each second is consumed. Place the two instances in the different CPU cgroups, and observe the effect on the rate of execution of the two programs. What happens if you adjust `cpu.cfs_quota_us` to 50000 in the `slow` cgroup?
   ⚠This exercise requires a kernel configured with the `CONFIG_CFS_BANDWIDTH` option. (`grep CFS_BANDWIDTH /lib/modules/$(uname -r)/build/.config`)

Exercises

2. 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 each of 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
```

   Thaw the `mfz` subtree (which processes resume execution?):

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

Exercises

Once more freeze the entire `mfz` 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
```

Try moving one of the processes in the frozen `mfz` cgroup into the root cgroup. What happens?

Use the `kill -KILL` command to send a `SIGKILL` signal to a process in a frozen cgroup? Is the process killed immediately?

3. Among other features, the `memory` controller can be used to set an upper limit on the amount of memory consumed by a cgroup. Create a memory cgroup named `mx` and set an upper limit on the memory that may be consumed by the cgroup, and disable use of swap space in the cgroup, using the following commands:

```
# echo 100M > /sys/fs/cgroup/memory/mx/memory.limit_in_bytes
# echo 0 > /sys/fs/cgroup/memory/mx/memory.swappiness
```

Exercises

Place the shell of another terminal window into the cgroup, and in that shell use the `cgroups/alloc_mem.c` program to allocate more than 100 MB in the cgroup. The following command line will make 5000 calls to *malloc()* requesting 0x10000 bytes (64 kiB) on each call:

```
$ ./alloc_mem 0x10000 0 5000
```

What happens?

Demonstrate that the limit applies across all processes in the cgroup by running the following commands:

```
$ ./alloc_mem 0x10000 0 1000 &
$ ./alloc_mem 0x10000 0 1000 &
$ jobs
```

Try writing to the `memory.limit_in_bytes` file in the cgroup root directory. What happens?

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

```
8:cpu,cpuacct:/cpugrp3
7: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

Cgroup release

- Consider the following scenario:
  - We create a cgroup subdirectory
  - Some processes are moved into cgroup
  - Eventually, all of those processes terminate (or leave the cgroup)
- 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

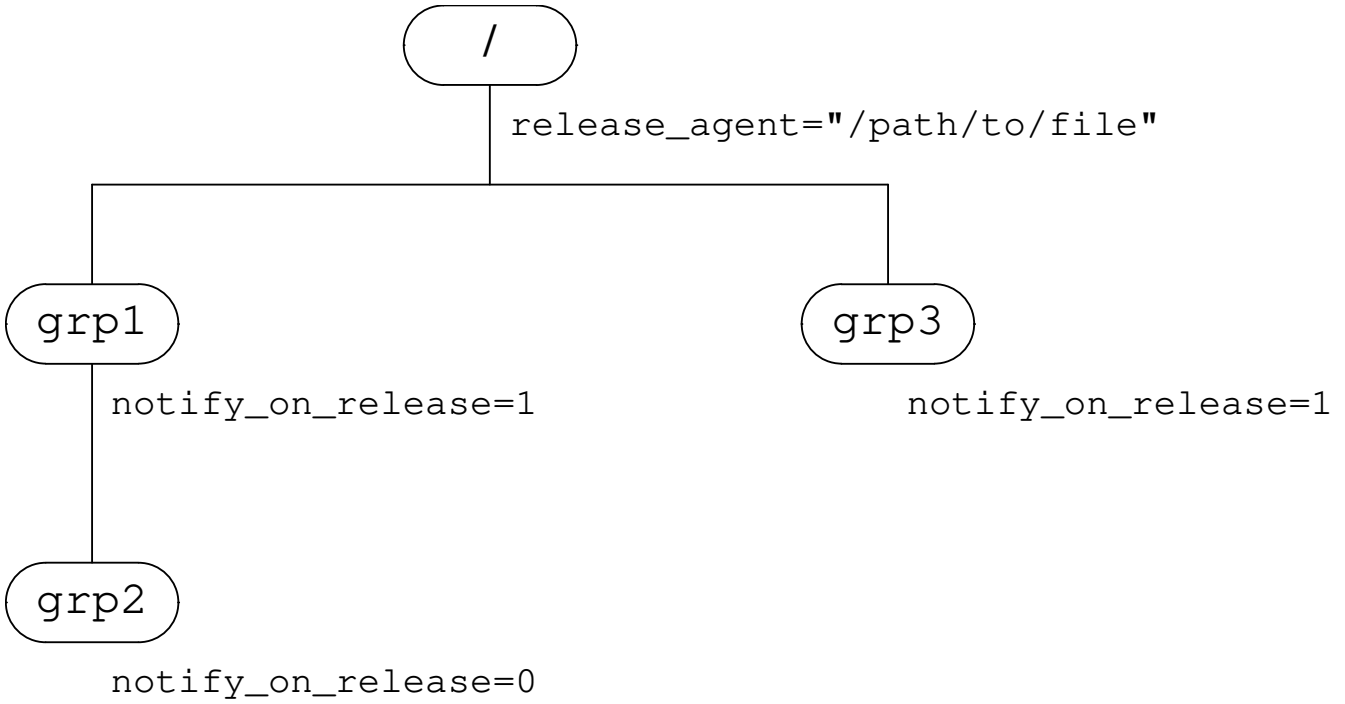
Cgroup release notification

- `release_agent` in **cgroup root directory**
  - Contains pathname of binary/script that is executed (as root) 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
  - Can also be specified as mount option

```
mount -o release_agent=/path/to/executable
```

- `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
    - Initial value of `notify_on_release` in root cgroup is 0

Cgroup release notification



- One `release_agent` file resides in cgroup root
- Each nonroot cgroup has `notify_on_release` file indicating whether `release_agent` will be executed when that cgroup becomes empty
  - `release_agent` is executed with cgroup path as argument

Mounting a named hierarchy with no controller

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

```
# mount -t cgroup -o none,name=somename \
            none /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`
    - (More recent *systemd* versions use an alternate cgroups v2 feature for same purpose)

Thanks!

Michael Kerrisk mtk@man7.org @mkerrisk

Slides at http://man7.org/conf/
Source code at http://man7.org/tlpi/code/

Training: Linux system programming, security and isolation APIs, and more; http://man7.org/training/

The Linux Programming Interface, http://man7.org/tlpi/