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
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`

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**
  - 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
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
- Not necessary (or possible) to delete attribute files inside cgroup directory before deleting it
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/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
- **⚠️** Some shells don’t check for error return from `write(2)`
  - So we might not see error for an invalid PID...
  - ⇒ Use `/bin/echo` since it checks for errors from `write(2)`
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

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 (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

- `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

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. Alternatively, you could use a different controller for this exercise.
- 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.

Linux Security and Isolation APIs ©2017, Michael Kerrisk Cgroups

4-35 §4.3

Exercises

2. In this exercise, we demonstrate the operation of a release_agent.
   - Create a shell script, `/tmp/release.sh`, with the following content:

```
#!/bin/sh
date > /tmp/release.log
echo $* >> /tmp/release.log
```

- Make the script executable: `chmod +x /tmp/release.sh`
- 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
```

- Change your working directory to the memory root cgroup and write the pathname of the script to the release_agent file:

```
# cd /sys/fs/cgroup/memory
# echo /tmp/release.sh > release_agent
```

Linux Security and Isolation APIs ©2017, Michael Kerrisk Cgroups

4-36 §4.3
Create two subdirectories, `x1` and `x2`, and write the value 1 to the `notify_on_release` file in `x1`:

```bash
# mkdir x1 x2
# echo 1 > x1/notify_on_release
```

Execute the following command, and note the PID assigned to the resulting process:

```bash
# sleep 300 &
```

Write the PID of the process created in the previous step into the file `x1/cgroup.procs`.

Now write the PID of the process into the file `x2/cgroup.procs`.

Inspect the contents of the file `/tmp/release.log`. You should see one line containing the date and time, and a second line containing the pathname `/x1`.

(If `/tmp/release.log` does not exist, review the above steps, kill the `sleep` process, and repeat the exercise.)