Container Technology

Jose L. Muñoz, Oscar Esparza, Juanjo Alins, Jorge Mata
Telematics Engineering
Universitat Politècnica de Catalunya (UPC)

Outline

Introduction

- Functionalities in the Linux kernel make easy to isolate Linux processes into their own “little environments”.
- Isolation allow building containers, which are a lightweight virtualization technology.
- **A single Linux kernel is shared between the host and the containers (virtual machines).**
- Containers can achieve higher densities of isolated environments than when using virtual machines.
- This concept is not new, as it was implemented a few years ago in BSD jails, Solaris Zones and other open-source projects.

Outline

chroot and mount

- During many years Unix systems have offered different tools to play with mounted parts of the filesystem and offer different views of the filesystem to processes.
- This has allowed some kind of isolation:
  - A remarkable tool is `chroot`, which allows to run a command or an interactive shell with a special root directory.
  - In addition, the `mount` command has several interesting options for isolation.
    - We can mount filesystems in files (`loop` option).
    - We can remount part of the file hierarchy somewhere else (`--bind` option).
    - Examples:

```
# mount -o loop debian7.fs image/
# mount --bind olddir newdir
```

    - After the last command call the same contents are accessible in two places.

proot

- The problem of the previous commands is that you **have to be root** to use them.
- There is an alternative called `proot`:
  - It is a user-space implementation of `chroot`, `mount --bind`, and `binfmt_misc`.
  - This means that users don't need any special privileges with `proot`.

  `binfmt_misc` is used to execute programs in user-space (for example by a special virtual machine).
- In the following examples the directories /mnt/slackware-8.0 and /mnt/armslack-12.2/ contain a Linux distribution respectively made for x86 CPUs and ARM CPUs.

chroot with proot

- To execute a command inside a given Linux distribution (chroot equivalent), just give proot the path to the guest rootfs followed by the desired command.
- The example below executes `cat` to print the content of a file:

```
$ proot -r /mnt/slackware-8.0/
$ cat /etc/motd
Welcome to Slackware Linux 8.0

$ proot -r /mnt/slackware-8.0/ cat /etc/motd
Welcome to Slackware Linux 8.0
```

chroot with proot I

- The bind mechanism enables one to relocate files and directories.
- This is typically useful to trick programs that perform access to hard-coded locations, like some installation scripts:

```
$ proot -b /tmp/alternate_opt:/opt
$ cd to/sources
$ make install
[...]
install -m 755 prog "/opt/bin"
[...] # prog is installed in "/tmp/alternate_opt/bin" actually
```

chroot with proot II

- As shown in this example, it is possible to bind over files not even owned by the user.
- This can be used to “overlay” system configuration files, for instance the DNS setting:

```
$ ls -l /etc/hosts
-rw-r--r-- 1 root root 675 Mar  4  2011 /etc/hosts
$ proot -b ~/alternate_hosts:/etc/hosts
$ echo '1.2.3.4 google.com' > /etc/hosts
$ resolveip google.com
IP address of google.com is 1.2.3.4
$ echo '5.6.7.8 google.com' > /etc/hosts
$ resolveip google.com
IP address of google.com is 5.6.7.8
```

chroot + mount bind with proot I

- chroot & mount bind can be combined to make any file from the host rootfs accessible in the confined environment just as if it were initially part of the guest rootfs.
- For example, it is sometimes required to run programs that rely on some specific files:

```
$ proot -r /mnt/slackware-8.0/
$ ps -o tty,command
Error, do this: mount -t proc none /proc
```

- Works with:

```
$ proot -r /mnt/slackware-8.0/ -b /proc
$ ps -o tty,command
TT        COMMAND
?         bash
?         proot -b /proc /mnt/slackware-8.0/
?         sh
?         ps -o tty,command
```

chroot + mount bind with proot II

- Actually there's a bunch of such specific files.
- That's why `proot` provides the option `-R`.
- This option automatically binds a pre-defined list of recommended paths:

```
$proot -R /mnt/slackware-8.0/
$ ps -o tty,command
TT       COMMAND
pts/6    bash
pts/6    proot -R /mnt/slackware-8.0/
pts/6    sh
pts/6    ps -o tty,command
```

chroot + mount bind + su I

- Some programs will not work correctly if they are not run by the root user, this is typically the case with package managers.
- PRoot can fake the root identity and its privileges when the -0 (zero) option is specified:

```
$ proot -r /mnt/slackware-8.0/ -0
# id
uid=0(root) gid=0(root) [...]

# mkdir /tmp/foo
# chmod a-rwx /tmp/foo
# echo 'I bypass file-system permissions.' > /tmp/foo/bar
# cat /tmp/foo/bar
I bypass file-system permissions.
```

chroot + mount bind + su II

- The previous option is typically required to create or install packages into the guest rootfs.
- However, it is not recommended to use the `-R` option when installing packages since they may try to update bound system files, like /etc/group.
- Instead, it is recommended to use the `-S` option.
- This latter enables the -0 option and binds only paths that are known to not be updated by packages:

```
$ proot -S /mnt/slackware-8.0/
# installpkg perl.tgz
Installing package perl...
```

chroot+mntbind+binfmt_misc I

- PRoot uses QEMU user-mode to execute programs built for a CPU architecture incompatible with the host one.
- From users' point-of-view, guest programs handled by QEMU user-mode are executed transparently, that is, just like host programs.
- To enable this feature users just have to specify which instance of QEMU user-mode they want to use with the option `-q`:

```
$ proot -R /mnt/armslack-12.2/ -q qemu-arm
$ cat /etc/motd
Welcome to ARMedSlack Linux 12.2
```

- PRoot allows one to mix transparently the emulated execution of guest programs and the native execution of host programs in the same file-system namespace.
- It's typically useful to extend the list of available programs and to speed up build-time significantly.

chroot+mntbind+binfmt_misc II

- This mixed-execution feature is enabled by default when using QEMU user-mode, and the content of the host rootfs is made accessible through /host-rootfs:

```
$ proot -R /mnt/armslack-12.2/ -q qemu-arm
$ file /bin/echo
[...] ELF 32-bit LSB executable, ARM [...]
$ /bin/echo 'Hello world!'
Hello world!
$ file /host-rootfs/bin/echo
[...] ELF 64-bit LSB executable, x86-64 [...]
$ /host-rootfs/bin/echo 'Hello mixed world!'
Hello mixed world!
```

- Since both host and guest programs use the guest rootfs as /, users may want to deactivate explicitly cross-filesystem support.
- To compile with the ARM gcc:

```
$ proot -R /mnt/armslack-12.2/ -q qemu-arm
$ export CC=/host-rootfs/opt/cross-tools/arm-linux/bin/gcc
$ ./configure; make
```

chroot+mntbind+binfmt_misc III

- As with regular files, a host instance of a program can be bound over its guest instance.
- Here is an example where the guest binary of make is overlaid by the host one:

```
$ proot -R /mnt/armslack-12.2/ -q qemu-arm -b /usr/bin/make
$ which make
/usr/bin/make
$ make --version # overlaid
GNU Make 3.82
Built for x86_64-slackware-linux-gnu
```

- It's worth mentioning that even when mixing the native execution of host programs and the emulated execution of guest programs, they still believe they are running in a native guest environment.
- QEMU user-mode (package qemu-user) is required only if the guest rootfs was made for a CPU architecture incompatible with the host one.

Outline

VETH Interfaces I

- VETH interfaces are virtual Ethernet interfaces that always exist in pairs.
- Whatever enters on one interface, exits from the other one, and vice-versa.

- We can create veth interfaces as follows:

```
# ip link add veth0 type veth peer name veth1
# ip link show
...
23: veth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN...
link/ether ee:c0:0e:d6:ae:09 brd ff:ff:ff:ff:ff:ff
24: veth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN...
link/ether 4e:e8:84:bd:01:f0 brd ff:ff:ff:ff:ff:ff
```

VETH Interfaces II

- We created a pair of veth interfaces called `veth0` and `veth1` (these names can be chosen by the user).
- Now, let's send some traffic to show the interfaces in action.

```
# ip link set veth0 up
# ip link set veth1 up
# ip addr add 10.0.0.1/24 dev veth0
# ping -c 3 10.0.0.2
```

- Note. For an interface to be up, the other one must be up too.
- While the ping is running, you will be able to observe traffic on `veth1` (most likely ARP).
- We can delete veth interfaces as follows:

```
# ip link del veth0
```

- Notice that when removing a veth peer the other peer is automatically removed too.

Outline

Concept

- The key tool to build containers (LXC, Docker, Rocket etc.) are kernel namespaces:
  - Provide a way to put applications in isolated environments with separate process lists, network devices, user lists and filesystems.
  - Such a functionality is implemented inside the kernel without the need to run hypervisors or virtualization.
  - The OS needs to keep separate internal structures and make sure they remain isolated.

Implementation I

- **Kernel namespaces are manipulated using 3 syscalls**:
  1. `clone()` which is used to create a new children processes (this low-level function sits behind the well known `fork()`).
     When used with namespaces, clone() can create a new process in the specified namespace.
  2. `unshare()` which allows to modify the execution context of a process without spawning a new child process.
     Allows changing the namespaces of a running process.
  3. `setns()`. Instead of creating a new namespace for a process (like clone and unshare), allows attaching a process to an already created namespace.

Implementation II

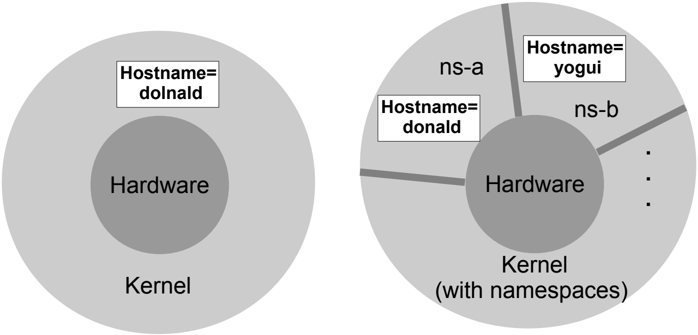
- Relationship of namespaces and containers:
  - The `clone()` allows to specify that you want the new process to run within one or more new namespaces.
  - When creating a container, this is exactly what happens.
  - A new process with new namespace is created.
  - Its network interfaces (including the special pair of interfaces to talk with the outside world) are configured.
  - It executes an init-like process.
- **Each namespace is materialized by a special file in /proc/PID/ns**:
  - When the last process within a namespace exits, the associated resources: network interfaces, etc. are automatically reclaimed.
  - It is also possible to "enter" a namespace, by attaching a process to an existing namespace.
  - This is generally used to run an arbitrary command within the namespace.

uts namespace I

- The uts namespace deals with one little detail:
  - The hostname that will be “seen” by a group of processes.
  - Each uts namespace will hold a different hostname.
  - Changing the hostname will only change it for processes running in the same namespace.
- You can create uts namespaces with `unshare -u`.
- You can use the `hostname` command to change the host name.

uts namespace II

With namespaces different processes can "see" different hostnames:



/proc/PID/ns files I

- Each process has a /proc/PID/ns directory that contains one file (symbolic link) for each type of namespace.
- One use of these symbolic links is to discover whether two processes are in the same namespace:
  - The kernel ensures that if two processes are in the same namespace, then the inode numbers reported for the corresponding symbolic link are equal.
  - The inode numbers can be obtained using `stat` but the kernel also constructs symbolic links:

```
$ ls -l /proc/$$/ns
total 0
lrwxrwxrwx 1 user1 user1 0 jul 19 16:42 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 user1 user1 0 jul 19 16:42 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 user1 user1 0 jul 19 16:42 net -> net:[4026531956]
lrwxrwxrwx 1 user1 user1 0 jul 19 16:42 pid -> pid:[4026531836]
lrwxrwxrwx 1 user1 user1 0 jul 19 16:42 user -> user:[4026531837]
lrwxrwxrwx 1 user1 user1 0 jul 19 16:42 uts -> uts:[4026531838]
```

/proc/PID/ns files II

- Let's check symbolic links at /proc/PID/ns:

```
t1# ls -l /proc/$$/ns
ls -l /proc/$$/ns
total 0
lrwxrwxrwx 1 root root 0 jul 19 16:54 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 jul 19 16:54 mnt -> mnt:[4026532289]
lrwxrwxrwx 1 root root 0 jul 19 16:54 uts -> uts:[4026530269]
...

t2# ls -l /proc/$$/ns
ls -l /proc/$$/ns
total 0
lrwxrwxrwx 1 root root 0 jul 19 16:54 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 jul 19 16:54 mnt -> mnt:[4026532289]
lrwxrwxrwx 1 root root 0 jul 19 16:54 uts -> uts:[8026537890]
...
```

- The two processes are in different uts namespaces.
- The /proc/PID/ns symbolic links also serve other purposes:
  - If we open one of these files, then the namespace will continue to exist as long as the fd remains open.
  - Even if all processes in the namespace terminate.

mnt namespace I

- The mnt namespace was the first implemented.
- This namespace deals with mountpoints.
- Processes living in different mnt namespaces can see different sets of mounted filesystems and different root directories.
- If a filesystem is mounted in a mnt namespace:
  - It will be accessible only to those processes within that namespace.
  - It will remain invisible for processes in other namespaces.
- The mnt namespace allows each container:
  - To have its own mountpoints.
  - See only those mountpoints.
  - With the path correctly translated to the actual root of the namespace.

mnt namespace II

- We can test this mnt namespaces with the `unshare` command (which implements the `unshare()` syscall):

```
t1# unshare -m /bin/bash
```

- The option `-m` is used to unshare the mnt namespace.
- Now, let's create a new mount point using a filesystem in memory (tmpfs):

```
t1# mount -n -o size=1m -t tmpfs tmpfs  /tmp/mytmpfs
```

  - The option `-o size=1m` specifies that the system size is 1 megabyte.
  - The option `-n` is to use `mount` without writing in /etc/mtab (we look at /proc/mounts).

mnt namespace III

- To check our mount point using /proc/mounts:

```
t1# grep mytmpfs /proc/mounts
tmpfs /tmp/mytmpfs tmpfs rw,relatime,size=1024k 0 0
```

- Now, let's create some files:

```
t1# cd /tmp/mytmpfs
t1# touch hello.txt
t1# touch hola.txt
```

- Open another terminal now (terminal 2) and execute the following commands:

```
t2# ls -la /tmp/mytmpfs
```

- Files hello.txt and hola.txt are not visible because they were written in another mnt namespace inside a tmpfs mounted only in that namespace.

IPC namespace

- IPC namespaces isolate certain interprocess communication (IPC) resources.
- The common characteristic of these IPC mechanisms is that IPC objects are identified by mechanisms other than filesystem pathnames.

Process Namespace I

- When computer with Linux boots up:
  - It starts with just one process (with process identifier PID=1).
  - This process is the root of the process tree.
  - All the other processes start below this process in the tree.
- The PID namespace allows one to spin off a new tree with its own PID 1 process:
  - The process that does this remains in the parent namespace, in the original tree.
  - But makes the child the root of its own process tree.
  - We can now have multiple “nested” process trees.
  - Each process tree can have an entirely isolated set of processes.
- A “parent” PID namespace can see its children namespaces:
  - And it can affect them (for instance, with signals).
  - A child namespace cannot do anything to its parent namespace.

Process Namespace II

- /proc pseudo-filesystem:
    - If a pseudo-filesystem like /proc is mounted by a process within a pid namespace, it will only show the processes belonging to the namespace.
    - Note that the numbering is different in each namespace.
- A process in a child namespace will have multiple PIDs:
    - One PID in its own namespace, and a different PID in its parent namespace.
    - From the top-level pid namespace, you will be able to see all processes running in all namespaces.
- Unshare the PID namespace (on Ubuntu 16.04):

```
# unshare -f --mount-proc -p /bin/bash
# ps
PID TTY          TIME CMD
  1 pts/14   00:00:00 bash
  11 pts/14   00:00:00 ps
```

User Namespace

- Is the only NS that can be created by unprivileged users:
  - Using the user namespace unprivileged users can create a namespace where they can be root.
  - From this namespace can start other namespaces.
- The design is based on a 1-1 uid mapping (by ranges) from uids in the namespace to uids on the parent:
  - For instance, uid 0 in the namespace may really be uid 999990 on the host.
  - Users can be pre-allocated their own private ranges.
  - The uid and gid mappings are exposed and manipulated through /proc/pid/uid_map and /proc/pid/gid_map.
- Regarding security:
  - An unprivileged user can only access to his existing privileges in the parent user namespace.
  - E.g. an unprivileged user might create a new filesystem tree and `chroot` into it, but he will not be able to mount over the filesystem on the parent namespace (e.g. /etc/passwd)

Net Namespace I

- A network namespace is logically:
  - Another copy of the network stack.
  - With its own routes, firewall rules, and network devices.
  - Even the loopback interface (`lo`) will be different in each different net namespace.
  - Each net namespace has its own meaning for INADDR_ANY, a.k.a. 0.0.0.0.
    - When a web server process binds to *:80 within its namespace.
    - It will only receive connections directed to the IP addresses and interfaces of its namespace.
    - This allows running multiple web server instances, with their default configuration listening on port 80.
- We can use the `unshare` command to create a network namespace.

Net Namespace II

- But it is more comfortable to use the `ip` command to work with network namespaces:
  - In the kernel, namespaces do not have names.
  - They are identified by the processes that are running inside these namespaces.
  - However, the `ip` command uses the concept of “named network namespace”.
  - By convention is an object at /var/run/netns/NAME that can be opened.
  - The file descriptor (fd) resulting from opening /var/run/netns/NAME refers to the specified network namespace:
    - This fd can be used to change the network namespace associated with a process (with the syscall `setns()`).

Playing with netns I

- Create/list a network namespace:

```
# ip netns add myns
# ip netns list
myns
# ls /var/run/netns/
myns
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:1e:4f:f9:76:ac
          ...
lo        Link encap:Local Loopback
          ...
```

- Start a bash in that namespace:

```
# ip netns exec myns bash
# ifconfig
# exit
exit
```

Playing with netns II

- Send a physical interface to a netns:

```
# ip link set eth0 netns myns
```

  - The previous command:
    - Triggers changing the network namespace of the net_device to myns.
    - The syscall used is dev_change_net_namespace().
  - An important fact is that **a network interface** can only **exist in one namespace at a time**:
    - So a physical NIC passed into the container is not usable on the host.
    - Also sockets belong to a single namespace.
    - After deleting a namespace, all its migratable network devices are moved to the default network namespace.

Playing with netns III

- Non-root processes that are assigned to a namespace via `clone()`, `unshare()`, or `setns()` only have access to the networking devices and configuration that have been set up in that namespace.
- Only the root user can add new devices and configure them.
- There are two ways to address a netns with `ip`:
  - By its name.
  - By the process ID of a process in that namespace.
- For example:

```
# ip netns exec myns bash
# ip link add vethMYTEST type veth peer name eth0
# ip link set vethMYTEST netns 1
# exit
```

Playing with netns IV

- If you want to send again `eth0` to the original network namespace type:

```
# ip netns exec myns bash
# ip link set eth0 netns 1
# exit
```

- Where 1 is used as the PID of init, which is a process on the original network namespace.
- Rather than using physical interfaces, a more typical approach is to use VETH interfaces.
- To send one leg of a veth interface to a network namespace:

```
# ip link add veth0 type veth peer name veth1
# ip link set veth1 netns myns
```

- The first command sets up a veth interface and the second assigns `veth1` to myns.

lxc-unshare I

- This command, provided by LXC allows us to run a task in a new set of namespaces.
- The syntax is:

```
lxc-unshare {-s namespaces } [-u user ] [-H hostname ] [-i ifname ] \
[-d] [-M] {command}
```

- Its options are:

```
-s namespaces     Specify the namespaces to attach to,
                  as a pipe-separated list, e.g. NETWORK|IPC .
                  Allowed values are MOUNT,PID,UTSNAME,IPC,
                  USER and NETWORK.
-u user           Specify a userid which the new task should become.
-H hostname       Set the hostname in the new container.
                  Only allowed if the UTSNAME namespace is set.
-i interfacename  Move the named interface into the container.
                  Only allowed if the NETWORK namespace is set.
-d                Daemonize (do not wait for the container to exit
                  before exiting)
-M                Mount default filesystems(/proc /dev/shm and /dev/mqueue)
                  in the container. Only allowed if MOUNT namespace is set.
```

lxc-unshare II

- For example, to spawn a new shell with its own UTS (hostname) namespace:

```
# lxc-unshare -s UTSNAME /bin/bash
```

- The previous command is equivalent to `unshare -u /bin/bash`.
- Another example, to spawn a shell in a new network, pid, and mount namespace:

```
# lxc-unshare -s "NETWORK|PID|MOUNT" /bin/bash
```

- The resulting shell will have pid 1 and will see no network interfaces.
- After re-mounting /proc in that shell:

```
# mount -t proc proc /proc
```

lxc-unshare III

- `ps` output will show there are no other processes in the namespace.
- To spawn a shell in a new network, pid, mount, and hostname namespace.

```
# lxc-unshare -s "NETWORK|PID|MOUNT|UTSNAME" -M -H slave -i veth1 /bin/bash
```

- The resulting shell will have pid 1 and will see two network interfaces (lo and veth1).
- The hostname will be "slave" and /proc will have been remounted.
- `ps` output will show there are no other processes in the namespace.

Outline

cgroups I

- Control groups, or “cgroups“, facilitate the management and administration of the system resources.
- They consist of a set of mechanisms to measure and limit resource usage for groups of processes.
- LXC relies on the Linux kernel cgroups, which are a feature to limit, control and isolate resource usage of process groups (CPU, memory, disk I/O, etc.).
- Conceptually, it works a bit like the `ulimit` shell command or the `setrlimit()` system call;
- but instead of manipulating the resource limits for a single process, they allow to set them for groups of processes.
- These groups are hierarchical, beginning with the top group which all processes are located in unless set otherwise.

cgroups II

- You can then define groups, subgroups, sub-subgroups, etc. to facilitate your limitation needs.
- In cgroups, subsystems, also known as resource controllers, are used to apply a limits to certain parts of the system.
- Some of the main subsystems present is most Linux distributions are:
  - **blkio**. This subsystem sets limits on input/output access to and from block devices such as physical drives (disk, solid state, USB, etc.).
  - **cpu**. This subsystem uses the scheduler to provide cgroup tasks access to the CPU.
  - **cpuset**. This subsystem assigns individual CPUs (on a multicore system) and memory nodes to tasks in a cgroup.
  - **cpuacct**. This subsystem generates automatic reports on CPU resources used by tasks in a cgroup.

cgroups III

- **devices**. This subsystem allows or denies access to devices by tasks in a cgroup.
- **freezer**. This subsystem allows to suspend or resume tasks in a cgroup.
- **memory**. This subsystem sets limits on memory use by tasks in a cgroup, and generates automatic reports on memory resources used by those tasks.

- Some examples of the previous subsystems are shown next.
- It is also worth to mention that cgroups are not dependent upon namespaces.
- That is to say, they are independent concepts and it is possible to build cgroups without namespaces kernel support.

Pseudo-FS Interface I

- From userspace, the way to manipulate control groups is through the cgroup filesystem.
- All cgroups actions can be performed via filesystem actions.
- This means creating, removing, reading or writing into directories.
- It is worth to mention that all cgroup entries created are deleted after rebooting (are not persistent).
- Depending on your Linux distribution you may or may not have cgroups already set up.
- The easiest way to check is to run the mount and look for lines starting with cgroup.
- In the event that you do not have cgroup mount points,

Pseudo-FS Interface II

- you need to mount them by adding the following line to your /etc/fstab file:

```
cgroup  /sys/fs/cgroup  cgroup  defaults  0   0
```

- Some distributions (like Ubuntu) mount the separate subsystems (cpu, memory, blkio, etc) into separate directories.
- Other distributions, just use a flat structure.
- In any case, if you are configuring cgroups manually, you modify/view the cgroups configuration with the typical commands `echo` and `cat` used on the files of /sys/fs/cgroups (or its subdirectories).
- These files are not regular files though, just like /proc they are actually configuration interfaces for the Linux kernel's internal workings.

Pseudo-FS Interface III

- Another possibility to manage cgroups is to use the tools provided by the package cgroup-bin.
- You can install these utilities with the following command:

```
$ sudo apt-get install cgroup-bin
```

- Next, we show some configuration examples.

Creating a cgroup

- To create a cgroup simply create a directory in /sys/fs/cgroup or if you have a per-subsystem setup, in the appropriate directory for the subsystem.
- The kernel automatically fills the cgroup's directory with the settings file nodes.
- If you want to use the toolkit-way, use `cgcreate` and provide the subsystems you wish to add as a parameter:

```
# cgcreate -g cpu,cpuset,memory:/my_group
```

- To view the cgroup files for the cpu subsystem (in Ubuntu) you can type:

```
# cd /sys/fs/cgroup/cpu/my_group ; ls
cgroup.clone_children  cgroup.event_control  cgroup.procs
cpu.cfs_period_us  cpu.cfs_quota_us  cpu.shares
cpu.stat  notify_on_release  tasks
```

Attaching processes

- To attach a process just echo the processes ID into the tasks file of the group.
- Note, that you can only inject one task at a time:

```
# echo 1234 >/sys/fs/cgroup/cpu/my_group/tasks
```

- Alternatively, you can use the `cgclassify` command to classify multiple processes:

```
# cgclassify -g cpu,cpuset,...:/my_group 1234 1235 ...
```

Deleting a cgroup

- Deleting a cgroup is a little more tricky as you cannot directly use `rm -rf` to remove the files in that directory.
- Instead just use the following command, which removes the directory with a depth of one:

```
# find my_group -depth -type d -print -exec rmdir {} \;
```

- Again, there is a utility for that:

```
# cgdelete cpu,cpuset,...:/my_group
```

Limiting CPU usage I

- CPU limits come in two flavors:
  1. Binding cgroups to certain CPU cores (called affinity).
  2. Limiting the actual usage.
- For CPU affinity we use the cpuset subsystem:
  - Let's look at our CPU affinity before modifying it:

```
# cat cpuset.cpus
0-7
```

  - As you can see the CPU affinity for this group is set to all 8 cores.
  - To adjust the affinity:

```
# echo "0-2,4" > cpuset.cpus
```

  - The limit is then applied immediately (observe using the `top` command).
  - Be careful though, setting the CPU affinity of the root cgroup will affect all processes.

Limiting CPU usage II

- we can also adjust the CPU usage (CPU bandwidth) if only using the CPU affinity is not enough:
  - Set the weight of a group with the process scheduler.
  - This will still give the process all free CPU.
  - But will give other processes a higher priority when considering CPU allowance.
  - Done via the cpu.shares option (defaults to 1024):
    - Leaving one group (e.g. lesscpulimited) at the default of 1024.
    - Setting another group (e.g. cpulimited) to 512.
    - We are telling the kernel to split the CPU resources using a 2:1 ratio.
    - We can do this with the following command:

```
# echo 512 > cpu.shares
```

    - Or with:

```
# cgset -r cpu.shares=512 cpulimited
```

Limiting CPU usage III

- The final and most strict setting is the realtime CPU quota a process gets.
- This only works on realtime scheduling groups.
- You should not use it unless you know what you are doing.
- There are two configuration options:
  - cpu.rt_runtime_us limits how long the process can keep the CPU continuously at most.
  - cpu.rt_period_us sets the period length for the former setting.
- Thus, if you want a process to access the CPU 4 seconds out of 5:
  - you need to set cpu.rt_runtime_us to 4000000.
  - and cpu.rt_period_us to 5000000.
- We must remark that setting very small values in either option can result in an unstable system.

Limiting memory usage

- For limiting memory usage you have two parameters:
  - memory.limit_in_bytes limits the total memory usage of a cgroup including file cache.
  - memory.memsw.limit_in_bytes limits the amount of memory plus swap that a cgroup can use.
- Pay attention however, memory.limit_in_bytes should be set first, otherwise you will receive an error.
- Also note that you do not need to specify the amount in bytes, you can use the shorthand multipliers k or K for kilobytes, m or M for Megabytes, and g or G for Gigabytes.

Limiting Disk IO I

- Last but not least among the major limiting features there is the IO.
- Disk IO was a long time pain in the neck, since no really reliable methods existed for limiting it.
- With cgroups however, we have a couple of parameters available:
  - The parameter blkio.weight behaves just like the CPU shares.
  - When weights are not enough, you can use fixed limits.
  - You can either limit by bytes-per-second.
  - Or by IOPS (IO Operations Per Second).

Limiting Disk IO II

- The configuration options are called:

```
blkio.throttle.read_bps_device for read limits in BPS
blkio.throttle.read_iops_device for read limits in IOPS.
blkio.throttle.write_bps_device for write limits in BPS.
blkio.throttle.write_iops_device for write limites in IOPS.
```

- To adjust them you need to figure out the minor and major number of the device:
    - Easily done though, just use ls -la /dev and look at the line with your device:
    - The numbers in just before the date will be the two numbers you are looking for.
- To place a limit run the following with your major, minor and byte limits replaced:

```
# echo "252:2 10485760" > blkio.throttle.write_bps_device
```

Limiting Access to Devices I

- The devices cgroup provides enforcing restrictions on opening and mknod operations on device files.
- It contains three main files:
  - devices.allow. This file is a whitelist of devices.
  - devices.deny. This file is a blacklist of devices.
  - devices.list. Using cat over this file shows the available devices for the cgroup.
- Each entry of the previuos files has 4 fields:
  - **type:** can be a (all), c (char device), or b (block device). All means all types of devices, and all major and minor numbers.
  - **Major number.** The major number of the device (you can figure it with `ls -l /dev`).
  - **Minor number.** The minor number of the device.
  - **Access.** A composition of '`r`' (read), '`w`' (write) and '`m`' (mknod). For example, `rwm` means that you can read, write and create these device.
- For example, the special device /dev/null has as major number the number 1 and its minor number is 3.

Limiting Access to Devices II

- Create a group and view the permissions:

```
# cat /sys/fs/cgroup/devices/my_group/devices.list
a *:* rwm
```

- Now, we deny rmw access to /dev/null:

```
# echo 'c 1:3 rmw' > /sys/fs/cgroup/devices/my_group/devices.deny
```

- If you create a bash inside my_group and try:

```
$ echo "test" > /dev/null
bash: /dev/null: Operation not permitted
```

- To restore the access to /dev/null, we add 'a *:* rwm' entry to the whitelist (now there is no error):

```
# echo a > /sys/fs/cgroup/devices/0/devices.allow
$ echo "test" > /dev/null
```

Permanent Configuration

- Now that all limits are configured, you might want to make sure they are applied upon restart as all contents of /sys.
- cgroup configurations are volatile and a reboot deletes them.
- To persist the configuration, you can use the `cgconfigparser` utility.
- The `cgconfigparser` takes a configuration file and builds the corresponding cgroup structure.