Containers – namespaces and cgroups

--- Ajay Nayak

Why containers?

- Important use-case: implementing lightweight virtualization
  - Virtualization == isolation of processes
- Traditional virtualization: **Hypervisors**
  - Processes isolated by running in separate guest kernels that sit on top of host kernel
  - Isolation is “all or nothing”
- Virtualization via containers
  - Permit isolation of processes running on a single kernel be per-global-resource --- via namespaces
  - Restrict resource consumption --- via cgroups

Outline

- Motivation
- Concepts
- Linux Namespaces
  - UTS
  - UID
  - Mount
- C(ontrol) groups
- Food for thought

Concepts

- Isolation
  - Goal: Limit "**WHAT**" a process can use
  - "wrap" some global system resource to provide resource isolation
  - Namespaces jump into the picture

- Control
  - Goal: Limit "**HOW MUCH**" a process can use
  - A mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups
  - Assign specialized behaviour to the group
  - C(ontrol) groups jump into the picture

https://github.com/nayakajay/linux-namespaces

Linux namespaces

- Supports following NS types: (CLONE_FLAG; symlink)
  - Mount (CLONE_NEWNS; /proc/pid/ns/mnt)
  - UTS (CLONE_NEWUTS; /proc/pid/ns/uts)
  - IPC (CLONE_NEWIPC; /proc/pid/ns/ipc)
  - PID (CLONE_NEWPID; /proc/pid/ns/pid)
  - Network (CLONE_NEWNET; /proc/pid/ns/net)
  - User (CLONE_NEWUSER; /proc/pid/ns/user)
  - Cgroup (CLONE_NEWCGROUP; /proc/pid/ns/cgroup)
  - Time (CLONE_NEWTIME; /proc/pid/ns/time) <= very new!

```
# Magic symlinks, which tells the namespace the process is in
sh1$ readlink /proc/self/ns/uts
uts :[4026531838]
# Context that kernel uses to resolve values
```

New tools to use

syscalls

- *clone()* – associates a new child process with few NS(s)
- *unshare()* – new NS(s) with the current process
- *setns()* – move calling process to existing NS

Shell commands

- *unshare* - create new NS(s) and execute a command in the NS(s)
- *nsenter* - enter existing NS(s) and execute a command

Unix Timesharing System namespace

- Simplest Namespace
- Isolate two system identifiers
  - ***nodename*** – system hostname
  - ***domainname*** – NIS domain name
- Why is it needed?
  - ***nodename*** could be used with DHCP, to obtain IP address for container

Unix Timesharing System namespace (Demo)

Shell 1

```
# Show hostname of initial UTS NS
sh1$ hostname
wolverine



# Verify if changed?
sh1$ hostname
wolverine
```

Shell 2

```
# Create new (u)ts namespace
$ PS1 =’sh2% ’ sudo unshare -u bash
# Show hostname of initial UTS NS
sh2% hostname
wolverine
# Change hostname
sh2% hostname subzero
# Verify change
sh2% hostname
subzero
```

Need (CAP_SYS_ADMIN) capability to create a UTS NS

User namespace

- Isolate user and group ID number spaces
  - A process's UIDs and GIDs can be different inside and outside user namespace
- User NSs have a hierarchical relationship
- Maintain mapping:
  - User ID: /proc/PID/uid_map
  - Group ID: /proc/PID/gid_map
- Most interesting use case:
  - Outside user NS: process has normal unprivileged UID
  - Inside user NS: process has UID 0
  - Superuser privileges for operations inside user NS!

User namespace - hierarchy

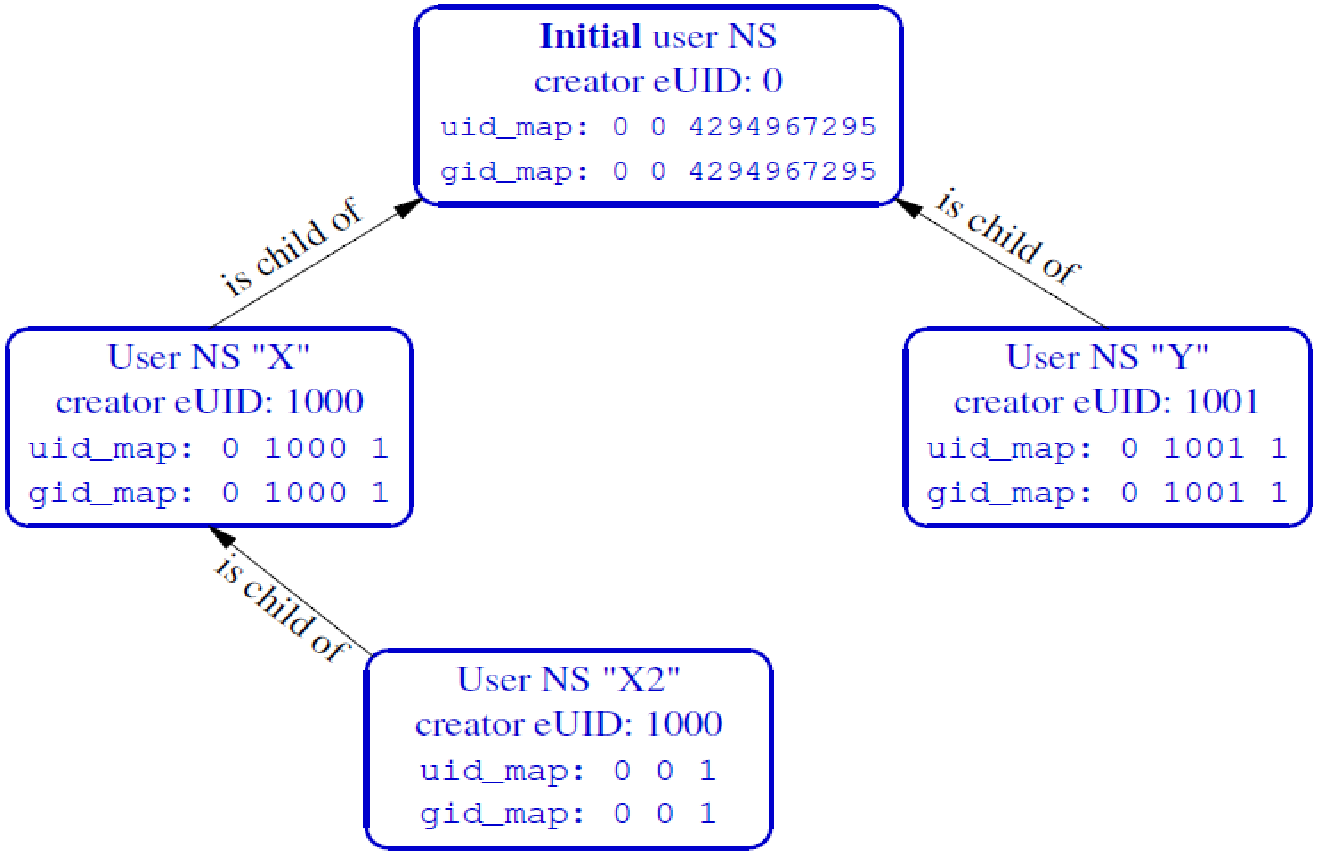

Image credits: Michael Kerrisk

User namespace (Demo)

Shell 1

```
# Get username and id
sh1$ whoami
ajayn
sh1$ id
uid=1008(ajayn) gid=1008(ajayn)


# Use Shell 2 process id
```

Shell 2

```
# Create new (U)ser namespace
$ PS1 =’sh2% ’ unshare -U bash
# Get ID inside new User NS
sh2% id
uid=65534(nobody) gid=65534(nogroup)
# Get PID
sh2% echo $$
4
```

User namespace (Demo)

Shell 1

```
# Edit uid_map file for shell 2

sh1$ echo "0 1008 1" >
/proc/4/uid_map
```

Shell 2

```
# Get ID inside new User NS

sh2% id

uid=0(root) gid=65534(nogroup)


# Yay, we are now root, but only
restricted in this shell!
```

uid_map for user ids, but group is still *nogroup* --- set something!

Mount namespace

- Isolation of set of mount points (MPs) seen by process(es)
  - MP is a tuple that includes:
    - Mount source (e.g., device)
    - Pathname
    - ID of parent mount
  - Process's view of filesystem (FS) tree is defined by (hierarchically related) set of MPs
- Mount NSs allow processes to have distinct sets of MPs
  - Processes in different mount NSs see different FS trees

Mount namespace – syscalls

- *mount()* and *umount()* affect processes in same mount NS as caller
- *pivot_root()*
  - Takes 2 arguments --- *new_root* and *put_old*
    - Mount root FS of calling process to *put_old*
    - Mount FS pointed by *new_root* as current root FS at "/"

Mount namespace (Demo)

Shell 1

```
# Get mount information
sh1$ cat /proc/$$/mounts
…
/dev/sda1
…
```

Shell 2

```
# Create new (m)ount namespace
$ PS1 =’sh2% ’ sudo unshare -m bash
# Get mount information
sh2% cat /proc/$$/mounts
…
/dev/sda1
…
```

Create a minimal installation “rootfs” >> next slide

Mount namespace (Demo)

```
# Create a minimal root installation

$> wget http://dl-cdn.alpinelinux.org/alpine/v3.10/releases/x86_64/alpine-minirootfs-3.10.1-x86_64.tar.gz

# Create rootfs directory, for new_root argument to pivot_root syscall

$> mkdir rootfs

# Untar the contents to rootfs directory

$> tar -xzf alpine-minirootfs-3.10.1-x86_64.tar.gz -C rootfs
```

Mount namespace (Demo)

Shell 1

```
# Get mount information
sh1$ cat /proc/$$/mounts
…
/dev/sda1
…
```

Shell 2

```
# Make rootfs directory as new root
sh2% mount --bind rootfs rootfs
sh2% cd rootfs
sh2% mkdir put_old
sh2% pivot_root . put_old
sh2% cd /
# Unmount put_old
sh2% umount -l put_old
# Get mount information?
sh2% cat /proc/$$/mounts
```

Mount namespace (Demo)

Shell 1

```
# Get mount information
sh1$ cat /proc/$$/mounts
…
/dev/sda1
…
```

Shell 2

```
# Make rootfs directory as new root
sh2% mount --bind rootfs rootfs
sh2% cd rootfs
sh2% mkdir put_old
sh2% pivot_root . put_old
sh2% cd /
# Unmount put_old
sh2% umount -l put_old
# Get mount information?
sh2% cat /proc/$$/mounts
```

What happens to */proc* now?

C(ontrol) groups

- Originally developed by Google
- The framework provides the following
  - Resource limiting, prioritization, accounting, control

- Two principle components
  - Group: processes bound to set of parameters or limits
  - (Resource) controller: kernel component that controls or monitors processes in a cgroup
    - *memory*: limits memory usage
    - *cpuacct*: accounts for CPU usage

C(ontrol) groups – more tools

```
# cgroup folder format /sys/group/cgroup/controller/group
$> sudo mkdir /sys/fs/cgroup/memory/foo

# Each file inside the group is controller.keyword
$> echo 500000 > /sys/fs/cgroup/memory/foo/memory.limit_in_bytes

# Verify the setting. Returns in multiple of 4KB. (Why?)
$> cat /sys/fs/cgroup/memory/foo/memory.limit_in_bytes
503808 << 492KB

# Add a process ID to foo group's memory controller
$> echo 12345 > /sys/fs/cgroup/memory/foo/cgroup.procs
```

C(ontrol) groups – [libcgroup-tools]

```
# Libraries make life easier (and difficult too!)
$> sudo cgcreate -g memory,cpu:limit_group

# Set some limits to these controllers
$> sudo cgset -r memory.limit_in_bytes=$((500*1024*1024)) limit_group

# Run an executable under the group and controllers
$> sudo cgexec -g memory,cpu:limit_group bash

# Add an existing process to the group, using process ID
$> sudo cgclassify -g memory,cpu:limit_group 12345
```

C(ontrol) groups – (Demo)

```
# Create a cgroup with memory controller (say, foo)
$> sudo cgcreate -g memory:foo

# Set some limits to these controllers (say, 10MB)
$> sudo cgset -r memory.limit_in_bytes=$((10*1024*1024)) foo

# Run an executable under the group and controller, within memory limit
$> sudo cgexec -g memory:foo exec
This program terminated happily!

# Now let’s try to restrict this program to very less memory!
```

C(ontrol) groups – (Demo)

```
# Set some small limits to these controllers (say, 4KB)
$> sudo cgset -r memory.limit_in_bytes=$((4*1024)) foo

# Run an executable under the group and controller, within memory limit
$> sudo cgexec -g memory:foo exec
Killed
Kernel's Out-of-Memory (OOM) Killer was invoked!

# That's all folks …… not really. There is tons more to explore!
```

Concept check

- The processor i.e., CPU is a global resource as it is used by all the processes sharing a host.
  - Why is processor control part of CGroup rather than namespace functionality, i.e., why is processor an accounting problem rather than a visibility problem?

- Is it a “HOW MUCH” problem or a “WHAT” problem?

Food for thought

- Imagine you need to create a system where:
  - You need to create a sandbox for an arbitrary program
  - Should have limited view of the system (filesystem, network, other processes)
  - Must use only few processors (say, 2) and not more than 100MB of memory, with restricted CPU time.
  - Restrict as few *syscalls* as possible

- Where can this kind of system be useful?
- What are the limitations?

** spoilers on the next slide **

Food for thought

- Imagine you need to create a system where:
  - You need to create a sandbox for an arbitrary program
  - Should have limited view of the system (filesystem, network, other processes)
  - Must use only few processors (say, 2) and not more than 100MB of memory, with restricted CPU time.
  - Restrict as few *syscalls* as possible
- Where can this kind of system be useful?
  - Programming platforms!
- What are the limitations?
  - Think dependencies!

Useful links

- **Namespaces:** https://lwn.net/Articles/531114/
- **Cgroups:** https://lwn.net/Articles/604609/
- **Seccomp:** https://lwn.net/Articles/656307/
- https://medium.com/@teddyking/linux-namespaces-850489d3ccf
- https://opensource.com/article/19/10/namespaces-and-containers-linux
- http://ifeanyi.co/posts/linux-namespaces-part-1/
- https://blog.lizzie.io/linux-containers-in-500-loc.html

Acknowledgements

- Michael Kerrisk (Linux man-pages contributor)