Containers unplugged: Linux namespaces

Michael Kerrisk, man7.org © 2019

mtk@man7.org

4 September 2019, Kongsberg
Outline

1 Overview 4
2 Namespace types 8
3 UTS namespaces 11
4 Namespace APIs and commands 15
5 Namespaces, containers, and virtualization 24
6 Mount namespaces 28
7 IPC namespaces 33
8 Cgroup namespaces 35
9 Network namespaces 37
10 PID namespaces 41
11 User namespaces (introduction) 49
Who am I?

- Maintainer to Linux *man-pages* project since 2004
  - $\approx 1050$ pages, mainly for system calls & C library functions
    - (I wrote a lot of those pages...)

- Author of a book on the Linux programming interface

- **Trainer**/writer/engineer
  - [http://man7.org/training/](http://man7.org/training/)

- Email: mtk@man7.org
  - Twitter: @mkerrisk
## Outline

1. **Overview**
2. Namespace types
3. UTS namespaces
4. Namespace APIs and commands
5. Namespaces, containers, and virtualization
6. Mount namespaces
7. IPC namespaces
8. Cgroup namespaces
9. Network namespaces
10. PID namespaces
11. User namespaces (introduction)
Namespaces: sources of further information

- See my LWN.net article series *Namespaces in operation*
  - https://lwn.net/Articles/531114/
  - Many example programs and shell sessions...

  *namespaces(7), cgroup_namespaces(7), mount_namespaces(7), network_namespaces(7), pid_namespaces(7), user_namespaces(7)*
  - Based on article series, but with further details, and updates for subsequent kernel versions

- “Linux containers in 500 lines of code”
Namespaces

- A namespace (NS) “wraps” some global system resource to provide resource isolation.
- Linux supports multiple NS types:
  - (Namespaces are a Linux-specific feature)
Namespaces

- For each NS type:
  - Multiple instances of NS may exist on a system
    - At system boot, there is one instance of each NS type—the so-called **initial namespace** of that type
  - Each process resides in one NS instance
  - To processes inside NS instance, it appears that only they can see/modify corresponding global resource
    - Processes are unaware of other instances of resource
  - When new process is created via `fork()`, it resides in same set of NSs as parent
<table>
<thead>
<tr>
<th></th>
<th>Overview</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Namespace types</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>UTS namespaces</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Namespace APIs and commands</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>Namespaces, containers, and virtualization</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Mount namespaces</td>
<td>28</td>
</tr>
<tr>
<td>7</td>
<td>IPC namespaces</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>Cgroup namespaces</td>
<td>35</td>
</tr>
<tr>
<td>9</td>
<td>Network namespaces</td>
<td>37</td>
</tr>
<tr>
<td>10</td>
<td>PID namespaces</td>
<td>41</td>
</tr>
<tr>
<td>11</td>
<td>User namespaces (introduction)</td>
<td>49</td>
</tr>
</tbody>
</table>
Linux supports following NS types:

- Mount (CLONE_NEWNS; 2.4.19, 2002)
- UTS (CLONE_NEWUTS; 2.6.19, 2006)
- IPC (CLONE_NEWIPC; 2.6.19, 2006)
- PID (CLONE_NEWPID; 2.6.24, 2008)
- Network (CLONE_NEWNET; ≈2.6.29, 2009)
- User (CLONE_NEWUSER; 3.8, 2013)
- Cgroup (CLONE_NEWCGROUP; 4.6, 2016)

Above list includes corresponding \texttt{clone()} flag and kernel release that “completed” implementation
Combining namespace types

- It’s possible to use individual NS types
  - E.g., mount NSs (first NS type) were invented to solve specific use cases

- But, often, several NS types are combined for an application
  - E.g., the use of PID, IPC, or cgroup NSs typically requires corresponding use of mount NSs
    - Because certain filesystems are commonly mounted for PID, IPC, and cgroup NSs

- In container-style frameworks, most or all NS types are used in concert
  - And cgroups (control groups) are thrown into the mix as well
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overview</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Namespace types</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>UTS namespaces</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Namespace APIs and commands</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>Namespaces, containers, and virtualization</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Mount namespaces</td>
<td>28</td>
</tr>
<tr>
<td>7</td>
<td>IPC namespaces</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>Cgroup namespaces</td>
<td>35</td>
</tr>
<tr>
<td>9</td>
<td>Network namespaces</td>
<td>37</td>
</tr>
<tr>
<td>10</td>
<td>PID namespaces</td>
<td>41</td>
</tr>
<tr>
<td>11</td>
<td>User namespaces (introduction)</td>
<td>49</td>
</tr>
</tbody>
</table>
UTS namespaces (CLONE_NEWUTS)

- UTS NSs are simplest NS, and so provide an easy example
- Isolate two system identifiers returned by `uname(2)`
  - `nodename`: system hostname (set by `sethostname(2)`)
  - `domainname`: NIS domain name (set by `setdomainname(2)`)
- Container configuration scripts might tailor their actions based on these IDs
  - E.g., `nodename` could be used with DHCP, to obtain IP address for container
- “UTS” comes from `struct utsname` argument of `uname(2)`
  - Structure name derives from “UNIX Timesharing System”
UTS namespaces (CLONE_NEWUTS)

- Running system may have multiple UTS NS instances
- Processes within single instance access (get/set) same `nodename` and `domainname`
- Each NS instance has its own `nodename` and `domainname`
  - Changes to `nodename` and `domainname` in one NS instance are invisible to other instances
Each UTS NS contains a set of processes (the circles) which see/modify same hostname (and domain name, not shown)
Outline

1  Overview 4  
2  Namespace types 8  
3  UTS namespaces 11  
4  Namespace APIs and commands 15  
5  Namespaces, containers, and virtualization 24  
6  Mount namespaces 28  
7  IPC namespaces 33  
8  Cgroup namespaces 35  
9  Network namespaces 37  
10 PID namespaces 41  
11 User namespaces (introduction) 49
Some “magic” symlinks

Each process has some symlink files in `/proc/PID/ns`

- `/proc/PID/ns/cgroup` # Cgroup NS instance
- `/proc/PID/ns/ipc` # IPC NS instance
- `/proc/PID/ns/mnt` # Mount NS instance
- `/proc/PID/ns/net` # Network NS instance
- `/proc/PID/ns/pid` # PID NS instance
- `/proc/PID/ns/user` # User NS instance
- `/proc/PID/ns/uts` # UTS NS instance

One symlink for each of the NS types
Some “magic” symlinks

- Target of symlink tells us which NS instance process is in:

```
$ readlink /proc/$$/ns/uts
uts:[4026531838]
```

- Content has form: `ns-type: [magic_inode-#]`

- Various uses for the `/proc/PID/ns` symlinks, including:
  - If processes show same symlink target, they are in same NS
APIs and commands

- Programs can use various system calls to work with NSs:
  - `clone(2)`: create new (child) process in new NS(s)
  - `unshare(2)`: create new NS(s) and move caller into it/them
  - `setns(2)`: move calling process to another (existing) NS instance

- There are analogous **shell commands**:
  - `unshare(1)`: create new NS(s) and execute a command in the NS(s)
  - `nsenter(1)`: enter existing NS(s) and execute a command
The *unshare(1)* and *nsenter(1)* commands

*unshare(1)* and *nsenter(1)* have flags for specifying each NS type:

```
unshare [options] [command [arguments]]
  -C Create new cgroup NS
  -i Create new IPC NS
  -m Create new mount NS
  -n Create new network NS
  -p Create new PID NS
  -u Create new UTS NS
  -U Create new user NS
```

```
nsenter [options] [command [arguments]]
  -t PID PID of process whose NSs should be entered
  -C Enter cgroup NS of target process
  -i Enter IPC NS of target process
  -m Enter mount NS of target process
  -n Enter network NS of target process
  -p Enter PID NS of target process
  -u Enter UTS NS of target process
  -U Enter user NS of target process
  -a Enter all NSs of target process
```
Privilege requirements for creating namespaces

- Creating **user** NS instances requires no privileges
- Creating instances of **other** (nonuser) NS types requires privilege
  - **CAP_SYS_ADMIN**
Two terminal windows (sh1, sh2) in initial UTS NS

```
sh1$ hostname  # Show hostname in initial UTS NS
antero
```

In sh2, create new UTS NS, and change hostname

```
sh2$ hostname  # Show hostname in initial UTS NS
antero
$ PS1='sh2#' sudo unshare -u bash
sh2# hostname bizarro  # Change hostname
sh2# hostname          # Verify change
bizarro
```

Used `sudo` because we need privilege (CAP_SYS_ADMIN) to create a UTS NS
In `sh1`, verify that hostname is unchanged:

```
sh1$ hostname
antero
```

Compare `/proc/PID/ns/uts` symlinks in two shells

```
sh1$ readlink /proc/$$/ns/uts
uts:[402653 1838]

sh2# readlink /proc/$$/ns/uts
uts:[402653 2855]
```

The two shells are in different UTS NSs
From *sh1*, use *nsenter*(1) to create a new shell that is in same NS as *sh2*:

```
sh2# echo $$       # Discover PID of sh2
5912

sh1$ PS1='sh3# ' sudo nsenter -t 5912 -u
sh3# hostname
bizarro
sh3# readlink /proc/$$/ns/uts
uts:[4026532855]
```

- Comparing the symlink values, we can see that this shell (*sh3#*) is in the second (*sh2#*) UTS NS
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overview</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Namespace types</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>UTS namespaces</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Namespace APIs and commands</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>Namespaces, containers, and virtualization</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Mount namespaces</td>
<td>28</td>
</tr>
<tr>
<td>7</td>
<td>IPC namespaces</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>Cgroup namespaces</td>
<td>35</td>
</tr>
<tr>
<td>9</td>
<td>Network namespaces</td>
<td>37</td>
</tr>
<tr>
<td>10</td>
<td>PID namespaces</td>
<td>41</td>
</tr>
<tr>
<td>11</td>
<td>User namespaces (introduction)</td>
<td>49</td>
</tr>
</tbody>
</table>
One important use of namespaces: implementing *lightweight virtualization* (AKA containers)

- 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 *namespaces* (containers)
- Permit isolation of processes *running on a single kernel*
- Isolation can be per-global-resource
Hypervisors

- (Relatively) simple to implement at kernel level
  - (Complete) isolation comes “for free” by having separate kernels
    - Can even employ guest kernels running a different OS
    - Strong isolation/security boundaries
  - First free Linux implementation appeared quite some time ago (Xen, 2003)
    - (Nonfree VMware came even earlier)
- But: separate kernel instance for each virtualization instance is an overhead
Namespaces/containers

- Cheaper in resource terms
- **Can selectively isolate** some global resources while not isolating others
- But: much **more work to implement** within kernel
  - Each global resource must be refactored inside kernel to support isolation (required changes are often extensive)
  - Mainline-kernel-based container systems much more recent
Mount namespaces (CLONE_NEWNS)

- First namespace type (merged into mainline in 2002)
  - CLONE_NEWNS: “new namespace”
    - No one foresaw that there might be further NS types...

- Isolation of set of mount points (MPs) seen by process(es)
  - Process’s view of filesystem (FS) tree is defined by (hierarchically related) set of MPs
  - MP is a tuple that includes:
    - Mount source (e.g., device)
    - Pathname
    - ID of parent mount

- Mount NSs allow processes to have distinct sets of MPs
  - \( \Rightarrow \) processes in different mount NSs see different FS trees

- \texttt{mount(2)} and \texttt{umount(2)} affect only processes in same mount NS as caller
Mount namespaces: use cases

- Per-process, private filesystem trees
- Jailing in the manner of *chroot*, but more flexible and secure
  - Can set process up with different root directory, and subset of available filesystems
- Mount new `/proc` FS without side effects
  - E.g., when also creating PID NS
  - Analogous use case when mounting `/dev/mqueue` for new IPC NS
Once upon a time (before Linux 2.4.19):
- Set of mount points (MPs) was a system-wide property shared by all processes
  - List of MPs viewable via /proc/mounts
- All kernel code that worked with MPs used same shared list
  - `mount()`, `umount()`
  - System calls that employ or resolve pathnames (`open()`, `stat()`, `link()`, `rename()`, and many, many others)

With mount namespaces:
- Each process is associated with one of multiple MP lists
  - (Now we need per-process /proc/PID/mounts)
- Inside kernel, every syscall that works with pathnames was refactored to handle fact that MP lists are per-namespace
- NS should automatically disappear when last process exits
And just a heads up

For time reasons, I’ll gloss over some key features related to mount NSs:

- **Shared subtrees and mount point propagation types**
  - See `Documentation/filesystems/sharedsubtree.txt` and `mount_namespaces(7)`
- Allow (controlled, partial) reversal of isolation provided by mount NSs
  - IOW: initial mount NS implementation provided too much isolation for many use cases
  - Permit automatic propagation of mount/unmount events in one mount NS to propagate to other mount NSs
    - Classic example use case: mount optical disk in one NS, and have mount appear in all NSs
IPC namespaces (CLONE_NEWIPC)

- Isolate certain IPC (*interprocess communication*) resources
  - System V IPC (message queues (MQs), semaphores, shared memory)
  - POSIX MQs
- Processes in an IPC NS instance share a set of IPC objects, but can’t see objects in other IPC NSs

Each NS instance has:

- Isolated set of System V IPC identifiers
- Its own POSIX MQ filesystem (/dev/mqueue)
- Private instances of various /proc files related to these IPC mechanisms
  - /proc/sysvipc, /proc/sys/fs/mqueue, etc.
- IPC objects automatically destroyed when NS is torn down
Cgroup namespaces (CLONE_NEWCGROUP)

- Difficult to describe without an understanding of cgroups (control groups)
  - But with that understanding, cgroup namespace concept is actually very simple
- See \texttt{cgroup_namespaces(7)} for full details
  - Essentially: virtualize pathnames exposed in certain \texttt{/proc/PID} files that show cgroup membership of a process
Network namespaces (CLONE_NEWNET)

- Isolate system resources associated with networking
  - IP addresses, IP routing tables, /proc/net & /sys/class/net directories, netfilter (firewall) rules, socket port-number space, abstract UNIX domain sockets
- Make containers useful from networking perspective
  - Each container can have virtual network device
  - Applications bound to per-NS port-number space
  - Routing rules in host system can direct network packets to virtual device of specific container
    - Virtual ethernet (veth) devices provide network connection between container and host system
Network namespaces use cases

- Containerized network servers
- Testing complex networking configurations on a single box
  - Instead of messing with HW to test network setup (routing and firewall rules), emulate in software
  - For example, Common Open Research Emulator, https://github.com/coreemu/core
Network namespaces use cases

Because network (NW) security is critical, many use cases revolve around isolation; some examples:

- Completely isolate process(es) from network
  - In initial state, network NS instance has no NW device
  - If compromised, process inside NS can’t access NW

- Isolate network service workers
  - Place server worker process in NS with no NW device
  - Can still pass file descriptors (e.g., connected sockets) via UNIX domain socket
    - FD passing example: `sockets/scm_rights_send.c` and `sockets/scm_rights_recv.c`
  
- Worker can provide NW service, but can’t access NW if compromised
Outline

1 Overview 4
2 Namespace types 8
3 UTS namespaces 11
4 Namespace APIs and commands 15
5 Namespaces, containers, and virtualization 24
6 Mount namespaces 28
7 IPC namespaces 33
8 Cgroup namespaces 35
9 Network namespaces 37
10 PID namespaces 41
11 User namespaces (introduction) 49
PID namespaces (CLONE_NEWPID)

- Isolate process ID number space
  - ⇒ processes in different PID NSs can have same PID

- Benefits:
  - Allow processes inside containers to maintain same PIDs when container is migrated to different host
  - Allows per-container *init* process (PID 1) that manages container initialization and reaping of orphaned children
Unlike (most) other NS types, PID NSs form a hierarchy

- Each PID NS has a parent, going back to initial PID NS
- **Parent** of PID NS is PID NS of caller of `clone()` or `unshare()`

- Maximum nesting depth: 32
- `ioctl(fd, NS_GET_PARENT)` can be used to discover parental relationship
  - Since Linux 4.9; see `ioctl_ns(2)` and
    http://blog.man7.org/2016/12/introspecting-namespace-relationships.html
PID namespace hierarchies

- A process is a member of its immediate PID NS, but is also visible in each ancestor PID NS.
- Process will (typically) have different PID in each PID NS in which it is visible!
- In **initial** PID NS, **can “see” all processes** in all PID NSs
  - See == employ syscalls on, send signals to, access via /proc, ...
- Processes in a NS will not be able to “see” any processes that are members only of ancestor NSs
  - Can see only peers in same NS + members of descendant NSs
A PID namespace hierarchy

A process is also visible in all ancestor PID namespaces

```
PID namespace

fork()

clone()

CLONE_NEWPID
```

Initial namespace

Child namespace

Child namespace

Grandchild namespace

PID

PID in ancestor namespace

PID namespace

fork()

clone()

CLONE_NEWPID
PID namespaces and PIDs

- `getpid()` returns caller’s PID inside caller’s PID NS
- When making syscalls and using `/proc` in outer NSs, process in a descendant NS is referred to by its PID in caller’s NS
- A caller’s parent might be in a different PID NS
  - `getppid()` returns 0!
- Fields in `/proc/PID/status` expose process’s/thread’s IDs in PID NSs of which it is a member
  - See `proc(5)` and `namespaces/pid_namespaces.go`
/proc/PID directories contain info about processes corresponding to a PID NS
- Allows us to introspect system
- Without /proc, many systems tools will fail to work
  - ps, top, etc.
- ⇒ create new mount NS at same time, and remount /proc

To mount /proc:

```
mount -t proc proc /proc
```
PID namespaces and *init*

First process inside new PID NS is special:

- Gets PID 1 (inside the NS)
- Fulfills role of *init*
  - Performs “system” initialization
  - Becomes parent of orphaned children
  - Can only be sent signals for which it has established a handler
- If killed/terminated, all other processes in NS are terminated (**SIGKILL**), and NS is torn down
- (Perfectly suits supporting containers as virtual systems)
User namespaces (CLONE_NEWUSER)

- Isolate user and group ID number spaces
  - IOW: a process’s UIDs and GIDs can be different inside and outside user namespace

- 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!

- Since Linux 3.8, no privilege is required to create a user NS
  - Unprivileged users now have access to functionality formerly available only to root
    - But only inside user NS...
User namespaces

Probably the most complex of the NS implementations:

- First kernel changes in Linux 2.6.23 (Oct 2007), more or less completed with 3.8 (Feb 2013)
  - More than five years!
- Required very wide-ranging changes in kernel
Thanks!

Michael Kerrisk, Trainer and Consultant
http://man7.org/training/

mtk@man7.org  @mkerrisk

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