NDC Security 2025

Linux containers in (less than) 100 lines of shell

Michael Kerrisk, man7.org © 2025

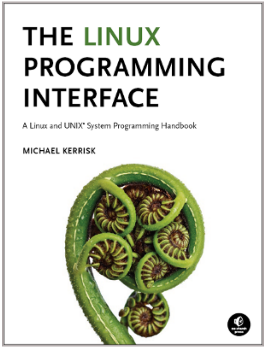
22 January 2025, Oslo, Norway

mtk@man7.org

Outline

Who?

- Linux *man-pages* project
  - `https://www.kernel.org/doc/man-pages/`
    - Manual pages pages documenting syscalls and C library
  - Contributor since 2000
  - Maintainer 2004-2020
  - Comaintainer 2020-2021
- I wrote a book
- Trainer/writer/engineer
  `http://man7.org/training/`
- `mtk@man7.org`, `@mkerrisk`



Outline

One day I wondered...

Can I create a (decent) container with shell commands?

man7.org

Building a container with shell commands

- So, it is possible (opinions on “decent” might differ...)
  - (And can be automated in a few scripts)
- It’s not a perfect container
  - Some untidy corners
  - Some set-up steps are omitted or need to be done manually
    - E.g., defining cgroup settings
  - And other limitations...
    - Only root UID/GID maps for user namespaces
    - No seccomp syscall filtering (no shell command for this)
- But, on the plus side:
  - Built using “simple” shell commands; and
  - Provides a fair approximation of the isolation of a Docker container

Building a container with shell commands

- We'll use a few standard commands:
  - *unshare(1)*, *mount(8)*, *pivot_root(8)*
    - Each of which is a layer on a system call of the same name
- And we'll simplify things by using *busybox(1)*
  - Emulates core functionality of ≈400 shell commands
  - We can avoid copying many individual binaries into our container filesystem
  - Statically linked!
    - No need to copy shared library dependencies into filesystem

man7.org

Building a container with shell commands

- We'll automate much of the set-up using some scripts
  - `create_lowerfs.sh`: **constructs** (lower layer of) **container filesystem** (FS)
    - Creates a suitable set of directories that should appear under a root FS, and places *busybox* in `/bin`
  - `consh_setup.sh`: **initial set-up of container**
    - Mount container FS; create a cgroup; launch container *init* process (a shell) in a set of new namespaces
  - `consh_post_setup.sh`: (automatically) launched in *init* shell to **complete the container setup**
    - Switch to container root FS; mount a set of pseudo-filesystems; create some devices; set host name
- Here goes...

man7.org

Outline

The container root filesystem

- A container needs a root filesystem (FS)
- That FS should be private to the container
  - So that FS changes don't have an effect outside container
- Each container will have **some files that are unique** to it
- But, **much of FS tree is the same across all containers**
  - E.g., each container has a `/bin`, containing same binaries

How do we efficiently provide a container filesystem?

- **If each container image stored copies of all files:**
  - **Disk space would be wasted**
    - Because many files are same across all containers
  - **Container start-up would be slow**
    - Because of need to copy all of the files to create image at container start-up
- These problems can be solved with a **union mount**

Union mounts

- A union mount
  - Combines contents of multiple directories ("layers")
  - **Provides a merged view** of those layers
- Merged view is taken from:
  - One or more **read-only lower layers**
  - A **read-write upper layer** that contains the differences from combined lower layers
  - If a file with same name appears in multiple layers, merged view shows file from uppermost layer
- From a container perspective:
  - Lower layer(s) contain FS content shared by all containers
  - Upper layer contains FS content that is private to container

OverlayFS

- In Docker and Podman, union mounts are provided using **OverlayFS**
  - https://docs.kernel.org/filesystems/overlayfs.html
  - https://wiki.archlinux.org/title/Overlay_filesystem
  - https://docs.docker.com/storage/storagedriver/overlayfs-driver/
- There are other possibilities, including:
  - Btrfs
  - UnionFS, aufs (both older)

OverlayFS

```
mount -t overlay overlay ./merged \
        -o lowerdir=./lower1:./lower2,upperdir=./upper,workdir=./work
```

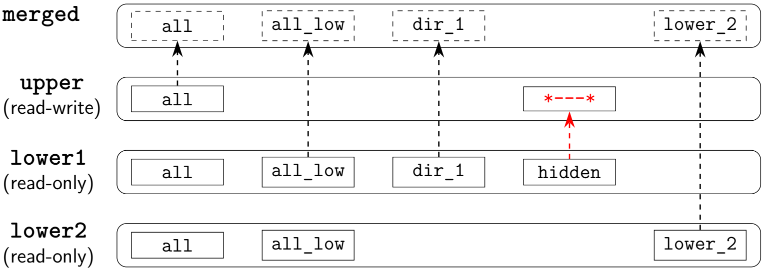
- Creates overlay FS mount at "`merged`" that combines two lower layers (`lower1`, `lower2`) and an upper layer (`upper`)
- `workdir` is a directory used internally by OverlayFS
  - Used internally to prepare files before they are atomically switched to `upperdir` [*]
  - Must be empty directory on same FS as `upperdir`

[*] E.g., consider how this must be implemented: `echo >> file-in-lower-layer`

- While doing operations in OverlayFS, try watching *workdir* (from outside container):
  `sudo inotifywait -m -r -format '%:e %f' work`

OverlayFS

```
mount -t overlay overlay ./merged \
        -o lowerdir=./lower1:./lower2,upperdir=./upper,workdir=./work
```



- Read-write upper layer is "diff" applied to lower layers
  - Diff may include "whiteouts" to represent deletion of a file from a lower layer (e.g., `hidden` above)

Constructing the root filesystem

- To create the container FS, we'll use a **union mount** constructed with OverlayFS, **with two layers**:
  - **Lower layer** containing a base image of files that are common to all containers
  - **Upper layer** containing the files that are unique to/modified in a container instance

Constructing the root filesystem

- We'll build lower layer with a script:

```
create_lowerfs.sh <dir>
```

  - `<dir>` is directory where base image is to be created

consh/create_lowerfs.sh

```
mkdir $1
cd $1
mkdir bin dev etc home proc root sys tmp usr var

cd bin
cp $(which busybox) .
$PWD/busybox --install .
```

- Create a reasonable set of directories that should appear in a root FS
  - (We won't actually populate all of those directories)
- Prepopulate `bin` with binaries to be used inside container:
  - Copy *busybox* into `bin` directory

    ```
    $ which busybox
    /usr/sbin/busybox
    ```

  - Use *busybox --install* to create all of the associated links
    - After this step, there will be ≈400 links in `bin`

man7.org

Outline

A container provides an illusion

- A **container provides an illusion** for the processes inside:
  - That the processes are in a “mini-system”: a world of their own and no other processes exist on the system

A container provides an illusion

To support the illusion, each container should have:

- Its own set of **mounted filesystems**
- Its own **hostname**
- Its own **network infrastructure**
    - E.g., own virtual NW devices, own socket port numbers
- A **private set of PIDs**
    - PIDs of container should not be visible outside
        - Allows each container to have its own *init* (PID 1)
    - Outside PIDs shouldn't be visible inside container
- The concept of "**superuser** inside the container"
    - I.e., processes that have privilege inside the container, but not outside
- And so on...

Implementing the illusion: namespaces

- The container illusion of “a world of their own” is primarily created via use of namespaces (NSs)
- A NS provides a virtual instance of some global resource that is private to a group of processes

Implementing the illusion: namespaces

- There are various types of NS, including:
  - **PID NS**s: make PIDs private to container; hide outside PIDs
    - Each container can have its own PID 1!
  - **Mount NSs**: provide a **private set of mounts**
    - Each container can have its own set of mounted filesystems
  - **UTS NSs**: allow each container to have its own **hostname**
  - **Network NSs**: provide a **private instance of NW infrastructure** (devices, routing rules, socket port #s, etc.)
    - Each container can have its own (virtual) NW device that provides connectivity to outside world
- For our container, we'll create one instance of each NS type

Implementing the illusion: superuser

- Concept of superuser-in-a-container is provided via **user NSs** + capabilities
- Capabilities break power of superuser into (mostly) small pieces
  - Currently, 41 different capabilities exist
    - E.g., `CAP_KILL`, send signals to arbitrary processes; `CAP_SETUID`, make arbitrary changes to process's UIDs
  - Traditional superuser == process with all capabilities
- We'll create a new user NS for our container
  - Kernel automatically grants **all capabilities to first process** in new user NS
    - I.e., superuser powers inside container

Creating namespaces

- At the shell level, a NS is created using *unshare(1)*
  - At kernel level, NSs are created using *unshare(2)* or *clone(2)* syscall
- Example:

```
$ unshare --user --pid --fork sh -c 'echo "My PID is $$!"'
My PID is 1!
```

  - Create new user and PID NSs, and run a new shell that displays its PID
    - First process in a new PID NS gets PID 1

Outline

Limiting container resource usage

- Isolation also means **limiting container's use of resources**
- For example, we want to:
  - **Prevent a container from overwhelming system** with excessive resource demands
  - Be assured that **other containers can't overwhelm system**
    - So that our container obtains reasonable share of resources
  - **Limit access to resources** such as devices
  - **Measure resource consumption** of container

Control groups (cgroups)

- On Linux, resource isolation/limitation is done via control cgroups (cgroups)
  - Key point: resource allocation is specified at level of **group** of processes
    - Older *ulimit* mechanism sets **per-process** limits
- Interface is a pseudo-filesystem (FS)
  - Mounted at `/sys/fs/cgroup`
  - Cgroup manipulation is done using FS commands
    - Creating a cgroup == creating directory on FS
    - Limits are set by writing values into files inside cgroup directory

Outline

consh/consh_setup.sh

- We'll use a script to do the container set up:

```
consh_setup.sh [options] <lower-dir> <overlay-dir>
    # Options: -c <cgroup-path> -h <hostname>
```

- *<lower-dir>*: directory to be used as lower layer in union mount used for container root FS
- *<overlay-dir>*: location (pathname) in which to create other pieces needed for union mount
  - I.e., `upper`, `work`, and the mount point, `merged`
- *<cgroup-path>*: [optional] pathname of cgroup into which container should be placed
- *<hostname>*: hostname to use in container
- Script places these **values into shell variables**: *lower*, *ovly_dir*, *cgroup* (possibly empty), and *host*

consh/consh_setup.sh

```
mkdir -p $ovly_dir/upper $ovly_dir/work
mkdir -p $ovly_dir/merged

sudo mount -t overlay -o lowerdir=$lower \
                      -o upperdir=$ovly_dir/upper \
                      -o  workdir=$ovly_dir/work \
                          overlay $ovly_dir/merged
cd $ovly_dir
```

- Create directories used in the OverlayFS union mount
  - `upper` will be upper layer of union mount
  - `work` is a directory used internally by OverlayFS
- Create mount point (`merged`)
- **Create union mount** at “`merged`”
  - *$lower* is directory we created with `create_lowerfs.sh`
- Change current directory to *$ovly_dir*

(After container terminates, we need to manually remove the mount and the directories)

consh/consh_setup.sh

```
manifest=merged/MANIFEST
echo "Created at:  $(date)"   > $manifest
echo "Creator UID: $(id -u)" >> $manifest
echo "Creator PID: $$"       >> $manifest
```

- As a demo, create a file that is private to this container
  - (File is created in `upper` layer of the union mount)

consh/consh_setup.sh

```
if test "X$cgroup" != "X"; then
    echo "Using cgroup: $cgroup" >> $manifest

    cgpath="/sys/fs/cgroup/$cgroup"
    sudo mkdir -p $cgpath

    sudo sh -c "echo $$ > $cgpath/cgroup.procs"
    ...
fi
```

- If a cgroup pathname was specified:
    - **Create cgroup**
    - **Move this shell into cgroup**
        - Children of this shell will also be in this cgroup
    - ...

man7.org

consh/consh_setup.sh

```
if test "X$cgroup" != "X"; then
    ...
    sudo sh -c 'cd '$cgpath'
        dlgt_files=$(ls $(cat /sys/kernel/cgroup/delegate) 2> /dev/null)
        chown '$(id -u):$(id -g)' . $dlgt_files'
fi
```

- If a cgroup pathname was specified:
  - ...
  - **Delegate the cgroup** to the user invoking this script
    - Delegation == changing ownership of cgroup directory and certain files inside that directory
    - Allows (unprivileged) user to manage subhierarchy (e.g, create child cgroups)
    - `/sys/kernel/cgroup/delegate` provides a list of files whose ownership must be changed (*if they exist*)
      (Not all of those files might exist; hence use of *ls* above)

consh/consh_setup.sh

```
exec env -i HOME="/root" PATH="/usr/sbin:/usr/bin:/sbin:/bin" \
            HOSTNAME="$host" \
            ENV="$(dirname $0)/consh_post_setup.sh" \
        unshare --user --map-root-user --pid --fork \
                --mount --net --ipc --uts --cgroup \
            busybox sh
```

- *exec*: replace the shell with the *env* command
  - Rather than executing in a child process
    - Reduces number of excess processes in container's cgroup
- Use *env* to clear environment (*–i*) and set a minimal set of environment variables
  - Use of *ENV* is explained shortly
- *env* in turn does an *exec* to replace itself with *unshare*

man7.org

consh/consh_setup.sh

```
exec env -i HOME="/root" PATH="/usr/sbin:/usr/bin:/sbin:/bin" \
          HOSTNAME="$host" \
          ENV="$(dirname $0)/consh_post_setup.sh" \
      unshare --user --map-root-user --pid --fork \
              --mount --net --ipc --uts --cgroup \
          busybox sh
```

- Use *unshare* to **create child process that runs in new NSs**
- *--user --map-root-user*: create a user NS with root mappings
  - This user NS will own all of the other NSs created here
  - (Recent (2022) versions of *unshare* have *--map-users* and *--map-groups*, to allow creation of more complex mappings)
- *--pid --fork*: create a PID NS and a child process
  - The child process will have PID 1 in new PID NS
- Remaining options specify creation of the other NS types

man7.org

consh/consh_setup.sh

```
exec env -i HOME="/root" PATH="/usr/sbin:/usr/bin:/sbin:/bin" \
            HOSTNAME="$host" \
            ENV="$(dirname $0)/consh_post_setup.sh" \
        unshare --user --map-root-user --pid --fork \
                --mount --net --ipc --uts --cgroup \
            busybox sh
```

- Run a shell in child process created by *unshare*
  - Run a *busybox* shell, in order to have a shell that is the same as that in `/bin` of the container FS

Outline

Performing initialization steps inside the container

- After the child process has been created, there are still some set-up steps to be done
- We perform those steps in another script
  - `consh/consh_post_setup.sh`
- Execution of that script is automated using the *ENV* environment variable
  - If *ENV* is defined, then a newly launched shell will execute the script it points to on start-up
- ⇒ Child shell launched by *unshare* automatically executes `consh/consh_post_set.sh`
  - As its first step, that script unsets *ENV*, so the script won't be executed by future shells run within container:

```
unset ENV
```

Setting up the container root FS: *pivot_root(8)*

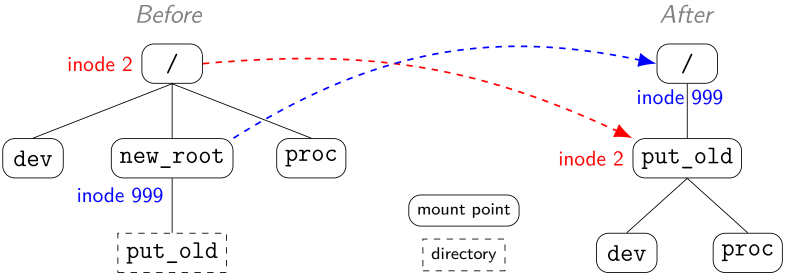
- Our "container" shell inherited the list of mounts from the initial mount NS
- We want to d**rop those mounts**, and use our **overlay mount as the root FS**
- Can do this with *pivot_root(8)* command:

```
pivot_root new_root put_old
```

  - Moves existing root mount of mount NS (and all descendant mounts) to *put_old*
  - Makes *new_root* the new root mount
  - Later, we will unmount old root FS

  - (*pivot_root(8)* is built on *pivot_root(2)* syscall)

The effect of *pivot_root(8)*



- *new_root* is made the new root mount
- Old root mount (along with all descendant mounts) is shifted to *put_old*

- Background notes: the root directory on a FS is always at inode 2; here, hypothetically, *new_root* has inode number 999

pivot_root(8) rules

- There are many rules governing use of *pivot_root*...
  - (See *pivot_root(2)* manual page)

man7.org

pivot_root(8) rules

- *new_root* and *put_old* must be directories and must not be on same mount as the current root mount
- *put_old* must be at or underneath *new_root*
  - This allows us to subsequently unmount old root FS
- *new_root* must be a path for an existing mount
  - (*pivot_root()* is essentially shuffling entries in mount list, so *new_root* must be a mount)
  - We can ensure *new_root* is a mount by bind mounting that path onto itself

pivot_root(8) rules

- To ensure that *pivot_root(8)* does not propagate changes to any other mount NS:
  - (Propagation is a mechanism whereby mounts in one NS automatically propagate to other NSs; we don't want this)
  - Propagation type of parent mount of *new_root* and parent of current root must not be "shared"
  - If *put_old* is an existing mount, its propagation type must not be "shared"

consh/consh_post_setup.sh

Again, we'll make a script, consh/consh_post_setup.sh:

```
mount --make-rprivate /

mount --bind merged merged
mkdir merged/oldrootfs

pivot_root merged merged/oldrootfs

cd /
```

- Ensure that no mounts have shared propagation
- Ensure that new root (merged) is a mount point
- Create a directory under new root (oldrootfs), so that current root can be moved there
- **Pivot the root directory**
- At this point, the root current directory of our shell is outside (above) the new root directory; fix that

consh/consh_post_setup.sh

```
mount -t proc proc /proc

mount -t sysfs sysfs /sys
mount -t cgroup2 cgroup2 /sys/fs/cgroup

mkdir -p /dev/mqueue
mount -t mqueue mqueue /dev/mqueue
```

- **Mount** `/sys` + some NS-related **pseudofilesystems**
  - So that we have mounts that are consistent with PID, IPC, and cgroup NSs of our container
    - In particular, `/proc` **mount ensures that** ***ps(1)*** **works!**

man7.org

consh/consh_post_setup.sh

```
for name in full null random tty urandom zero; do
    touch /dev/$name
    mount --bind oldrootfs/dev/$name /dev/$name
done
```

- **Add some useful devices**, by bind mounting to devices under old root FS

consh/consh_post_setup.sh

- **Unmount old root mount**:

```
umount -l oldrootfs
rmdir oldrootfs          # Remove now-unused mount point
```

  - This does a lazy unmount of the old root, and all of its descendant mounts
    - See description of `MNT_DETACH` in *umount(2)*
  - For obscure reasons, must be done after mounting `/proc`
    - `https://lore.kernel.org/lkml/87tvsrjai0.fsf@xmission.com/T/`
- **Set hostname** using value passed via environment variable:

```
hostname $HOSTNAME
```

- **And that's it!**

man7.org

Outline

Demo

- Let's use our scripts to create a container
- We do the following as an unprivileged user:

```
$ id
uid=1000(mtk) gid=1000(mtk) groups=1000(mtk),10(wheel)
```

- Create a directory for our work; inside that directory we create the base image for the union mount:

```
$ cd consh
$ mkdir demo
$ cd demo
$ ../create_lowerfs.sh lower
```

man7.org

Demo

- Start the container, creating overlay mount at `./merged`:

```
$ ../consh_setup.sh -c consh_cgrp -h tekapo lower .
```

  - We are now running a shell in our "container"
  - The shell is in the cgroup `consh_cgrp`

- Because we'll be hopping between shells, make prompt of *this* shell more distinctive:

```
/ # PS1="bbsh# "                    # Change the shell prompt
bbsh#
```

man7.org

Outline

PID namespaces

- From inside container, show PID of shell; use *ps*:

```
bbsh# echo $$
1
bbsh# ps ax                          # List all processes
PID   USER     TIME  COMMAND
    1 0         0:00 busybox sh
   15 0         0:00 ps
```

  - Shell was first process in a new PID NS, and so got PID 1
  - Processes outside the container are not visible
- From outside container, show PID of shell in initial PID NS:

```
$ ps -C busybox
    PID TTY          TIME CMD
  26926 pts/3    00:00:00 busybox
```

- What's going on?

man7.org

PID namespaces

- PID NSs exist in hierarchies
  - Each PID NS has a parent, which has a parent... back to initial PID NS
- A process that is member of a PID NS is also visible (i.e., has a PID in) in all ancestor NSs
  - `/proc/PID/status` shows shell's PID in each PID NS:

```
$ grep NStgid /proc/26926/status
NStgid: 26926 1
```

man7.org

Mount namespaces

- From outside the container (because *busybox* doesn't provide *findmnt*), view the mount tree of the container:

```
$ findmnt -o 'target,source,fstype' -N 26926
TARGET                 SOURCE       FSTYPE
/                      overlay      overlay
├─/dev                 tmpfs        tmpfs
│ └─/dev/mqueue        mqueue       mqueue
├─/sys                 sysfs        sysfs
│ └─/sys/fs/cgroup     cgroup2[...] cgroup2
└─/proc                proc         proc
```

  - This is a different (and smaller) set of mounts than is seen outside the container
  - The container has its own mount NS
  - (*–N <pid>* == show mounts in mount NS of *<pid>* rather than `/proc/self/mountinfo`)

User namespaces

- From inside container, show credentials of shell:

```
bbsh# id
uid=0 gid=0 groups=65534,65534,65534,0
```

  - The supplementary groups are messy, but it's the best we can do from a script
    - (One of the untidy corners of our container...)

- From outside the container, show credentials of the shell:

```
$ grep '[UG]id' /proc/26926/status
Uid:   1000   1000   1000   1000
Gid:   1000   1000   1000   1000
```

- UID 1000 outside container was mapped to 0 inside via creation of a **UID map** for container's user NS:

```
$ cat /proc/26926/uid_map
         0       1000          1
```

  - Mapping was created by *unshare --map-root-user*

UTS namespaces

- From inside container, view the hostname, and change it:

```
bbsh# hostname
tekapo
bbsh# hostname langwied
bbsh# hostname
langwied
```

  - Container is in a new UTS NS, so user can change hostname

man7.org

Outline

Superuser inside a container

- In previous demo, we changed the container's hostname
- How is that possible?
    - (Since privilege is required)
- And could a process inside container do superuser-y things outside the container?
    - (We certainly hope not, since *unprivileged* users can create containers)
- **How can a process be privileged inside a container while being unprivileged outside the container?**

Namespace relationships

Some things we need to know:

- Each non-user NS governs some type of global resource
  - Mount NS: mounts
  - UTS NS: hostname
  - Network NS: NW resources
  - etc.
- **Each non-user NS is owned by a user NS**
  - Ownership is established when non-user NS is created
- When our container was created, new instances of each NS type were created, including a new user NS
- Because all NSs were created at same time, **kernel made the new user NS the owner of the other new NSs**

Capabilities and superuser powers inside a container

- Kernel (automatically) grants **all capabilities to first process in a new user NS**
    - All capabilities == superuser powers
- Show capabilities of our container shell:

```
bbsh# grep -E 'Cap(Prm|Eff)' /proc/$$/status
CapPrm: 000001ffffffffff
CapEff: 000001ffffffffff
```

- All permitted and effective capabilities...
    - "`=ep`" as would be shown by *getpcaps(8)*

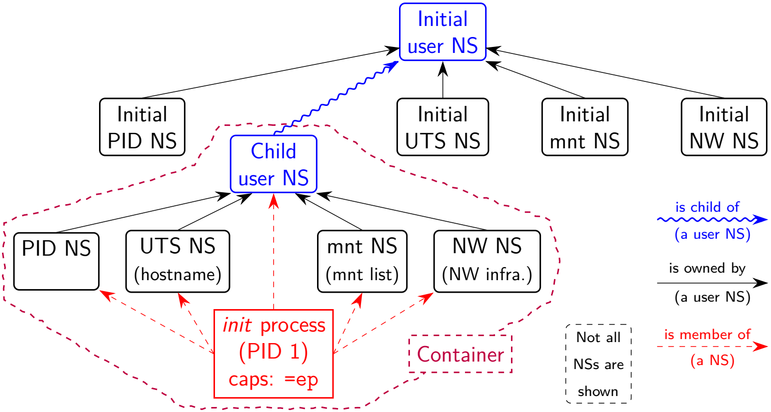
man7.org

What does it mean to be superuser inside a NS?

- But those superuser powers have effect only inside container, because...
- Root power in a user NS == **privilege over resources governed by non-user NSs owned by the user NS**

man7.org

Containers and namespaces



- "Superuser" process in a container has **root power over resources governed by non-user NSs owned by container's user NS**
- And does **not** have privilege in outside user NS
  - (E.g., can't change mounts seen by processes outside container)

Namespace relationships

From a shell outside container, use my `namespaces_of.go` to compare (some) NSs of that shell with NSs of container shell:

```
$ echo $$
28736
$ sudo go run namespaces/namespaces_of.go 28736 26926
user {4 4026531837} <UID: 0>            # Initial user NS
        [ 28736 ]
    cgroup {4 4026531835}
            [ 28736 ]
    ipc {4 4026531839}
            [ 28736 ]
    mnt {4 4026531841}
            [ 28736 ]
    [...]
    user {4 4026534280} <UID: 1000>     # User NS of the container
            [ 26926 ]
        cgroup {4 4026534285}           # Indentation indicates ownership
                [ 26926 ]
        ipc {4 4026534283}
                [ 26926 ]
        mnt {4 4026534281}
                [ 26926 ]
        [...]
```

- The container has its own user NS, which owns other NSs

Outline

Demo: cgroups

- From a shell outside the container, let's look at the container's cgroup:

```
$ cat /sys/fs/cgroup/consh_cgrp/cgroup.procs
26911
26926
$ ps 26911 26926
    PID TTY      STAT   TIME COMMAND
  26911 pts/1    S      0:00 unshare --user --map-root --pid ...
  26926 pts/1    S+     0:00 busybox sh     # Our container shell
```

  - Another small untidiness: *unshare* process shouldn't be in the cgroup; we can manually move it out if we care

man7.org

Demo: cgroups

- Inside the container, show cgroup membership of the shell:

```
bbsh# cat /proc/1/cgroup
0::/
```

  - Shell is in cgroup `consh_cgrp`...
  - But remount of `cgroup2` FS ensured a correctly virtualized path when looking from inside container
    - I.e., in cgroup NS of our container, `consh_cgrp` is the root cgroup
- How does cgroup membership of the container shell look from a shell in the outside world?

```
$ cat /proc/26926/cgroup
0::/consh_cgrp
```

  - This (different) path is consistent with the fact that we are looking from a different cgroup NS

Demo: cgroup delegation

- Let's look at cgroup directory and some files inside to see the **effect of delegation**:

```
$ ls -ld /sys/fs/cgroup/consh_cgrp
drwxr-xr-x. 3 mtk  mtk  0 Feb  1 15:20 /sys/fs/cgroup/consh_cgrp
$ ls -l /sys/fs/cgroup/consh_cgrp
total 0
-r--r--r--. 1 root root 0 Feb  1 15:19 cgroup.controllers
-r--r--r--. 1 root root 0 Feb  1 00:11 cgroup.events
...
-rw-r--r--. 1 mtk  mtk  0 Feb  3 10:38 cgroup.procs
-r--r--r--. 1 root root 0 Feb  1 15:19 cgroup.stat
-rw-r--r--. 1 mtk  mtk  0 Feb  1 15:19 cgroup.subtree_control
-rw-r--r--. 1 mtk  mtk  0 Feb  1 15:19 cgroup.threads
-rw-r--r--. 1 root root 0 Feb  1 15:19 cgroup.type
...
```

  - Cgroups created under `consh_cgrp` will also be owned by `mtk`

Demo: setting cgroup limits

- From a shell outside container, set a CPU limit for cgroup:

```
$ sudo sh -c 'echo 5000 10000 > /sys/fs/cgroup/consh_cgrp/cpu.max'
```

  - 50% of one CPU

- And copy a (statically linked) program that burns CPU into the container FS:

```
$ cd consh/demo
$ cp ../../timers/cpu_burner upper/
```

- From inside container, run that program:

```
bbsh# /cpu_burner
[17]  %CPU = 51.36
[17]  %CPU = 50.00
[17]  %CPU = 50.00
...
```

Outline

Demo: networking

- Let's use a virtual NW device to achieve NW connectivity into our container
- All steps are done using the standard *ip netns* command
  - See also the script, `consh/consh_nw_setup.sh`

ip netns

- One hurdle: normally, we create a NW NS using *ip netns add <name>*
  - **Creates a bind mount for NS** in `/var/run/netns`
  - That **mount is used in subsequent *ip netns* commands** in order to reach the NS
- Our container's NW NS has already been created, but we still need the bind mount for our *ip netns* commands
- ⇒ we create the bind mount manually from a shell outside the container:

```
$ sudo mkdir -p /var/run/netns          # Ensure directory exists
$ sudo touch /var/run/netns/consh       # Create the mount point
$ sudo mount --bind /proc/26926/ns/net /var/run/netns/consh
```

  - Our bind mount is named `consh`
  - `/proc/26926/ns/net` is NW NS of our container shell

man7.org

Setting up network infrastructure

From a *root* shell outside the container, we now set up some NW infrastructure:

- Create a pair of connected virtual Ethernet (`veth`) devices:

```
sudo bash
# ip link add veth0 type veth peer name veth1
```

  - We named the two devices `veth0` and `veth1`

- Move the `veth1` device into our container:

```
# ip link set veth1 netns consh
```

- Assign IP addresses to both `veth` devices & bring them up:

```
# ip address add 10.0.0.1/24 dev veth0
# ip link set veth0 up
# ip netns exec consh ip address add 10.0.0.2/24 dev veth1
# ip netns exec consh ip link set veth1 up
```

Setting up network infrastructure

Returning to our container shell:

- Show that the `veth1` device is present in the container:

```
bbsh# ip link show veth1
282: veth1@if283: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> ...
    link/ether 2e:c2:13:c5:4e:b8 brd ff:ff:ff:ff:ff:ff
```

Demonstrating network connectivity

- How can we easily fire up a NW server inside the container?
  - *busybox* does Netcat (*¡Genial!*)
- Inside our container, start a listening server on port 50000:

```
bbsh# nc -l -p 50000 -e sh -c \
              's=; while true; do s=x$s; echo $s; sleep 1; done'
```

  - After accepting a connection, server script sends strings of ever-increasing length
- From a shell outside the container, we connect to the server and see:

```
# nc 10.0.0.2 50000
x
xx
xxx
xxxx
...
```

Outline

If this is a "decent" container, we should be able to do one more thing...

Can we create a container inside a container?

A container inside a container

What are the hurdles?

- The *busybox unshare* applet doesn't support *--cgroup*
  - ⇒ We'll use a statically linked version of the standard *unshare* program provided by *util-linux*
- `consh_setup.sh` uses *sudo*, but *busybox* has no *sudo* applet
  - But, we don't need *sudo* because container shell already has all capabilities
  - ⇒ We'll edit the script
- We need a union mount for inner container, but **upper layer in OverlayFS can't itself be an OverlayFS mount**
  - IOW: we can't use FS of outer container in upper layer of inner container's FS
  - ⇒ Mount a new *tmpfs* + create union mount layers there

Creating the outer container

- First, we create the outer container:

```
$ cd consh
$ mkdir demo
$ cd demo
$ ../create_lowerfs.sh lower
$ ../consh_setup.sh -c cgrp -h tekapo lower .
/ # PS1="bbsh# "                    # Change the shell prompt
bbsh#
```

Preparations for the inner container

- Now we need to copy the files into outer container that will be used to set up inner container...
- From a shell outside the container, we copy our scripts into the container FS:

```
$ cd consh
$ cp  *.sh demo/upper/
```

- And edit the scripts to remove the *sudo* strings:

```
$ sed -i 's/sudo //' demo/upper/*.sh
```

- And copy in a statically linked version of the standard *unshare* command:

```
$ rm demo/lower/bin/unshare
        # Steps to build unshare.static omitted
$ cp /some/path/util-linux/unshare.static demo/lower/bin/unshare
```

Starting the inner container

- Returning to our outer container, we mount a *tmpfs* FS where we will create the components of the union mount for the inner container:

```
bbsh# mkdir demo_inner
bbsh# mount -t tmpfs tmpfs demo_inner
```

- Create the lower layer for the union mount:

```
bbsh# cd demo_inner
bbsh# ../create_lowerfs.sh lower
```

- Start the inner container:

```
bbsh# ../consh_setup.sh -c cgrp_2 -h pukaki lower .
/ #
```

  - "`/ #`" is prompt of *busybox* shell in inner container...

man7.org

Examining the inner container

- Start a *sleep* process in the inner container:

```
/ # /bin/sleep 1000          # Full path to avoid 'sleep' built-in
```

  - We use absolute pathname to avoid use of *sleep* built-in command (which would not create separate process)

- From a shell in the initial NS, obtain PID of *sleep*:

```
$ pidof sleep
69884
```

Examining the inner container

- Let's use my `namespaces/namespaces_of.go` to compare some NSs of a shell in initial NSs with NSs of *sleep*:

```
$ go run namespaces_of.go --namespaces=user,cgroup,pid,uts $$ 69884
user {4 4026531837} <UID: 0>                   # Initial user NS
        [ 56734 ]
    cgroup {4 4026531835}
            [ 56734 ]
    pid {4 4026531836}
            [ 56734 ]
    uts {4 4026531838}
            [ 56734 ]
    user {4 4026534072} <UID: 1000>            # User NS of outer container
        user {4 4026532574} <UID: 1000>        # User NS of inner container
                [ 69884 ]
            cgroup {4 4026534004}
                    [ 69884 ]
            pid {4 4026534003}
                    [ 69884 ]
            uts {4 4026534001}
                    [ 69884 ]
```

- *sleep* is in grandchild user NS that owns various other NSs
    - *sleep* is also a member of those other NSs

man7.org

Examining the inner container

- Display PID of *sleep* in all PID NSs where it is present:

```
$ grep NStgid /proc/69884/status
NStgid: 69884 54  17
```

  - Three PIDs ⇒ *sleep* is in a grandchild PID NS
  - Verify by using my program to examine PID NS hierarchy:

```
$ sudo go run namespaces_of.go --pidns 69884
pid {4 4026531836}                 # Initial PID NS
    pid {4 4026535236}             # PID NS of outer container
        pid {4 4026534583}         # PID NS of inner container
                [ 69884 ]
```

- Display cgroup membership of *sleep*:

```
$ cat /proc/69884/cgroup
0::/cgrp/cgrp_2
```

  - It is in a child cgroup of the outer container's cgroup

man7.org

Sure does look like a container inside a container!

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/`