Faculty of Engineering
Department of Computer Engineering
CENG536 – Advanced UNIX
Final Project Report – **Tuğca Eker / 1942010**

ORTA DOĞU TEKNİK ÜNİVERSİTESİ
MIDDLE EAST TECHNICAL UNIVERSITY

January 22, 2018

Linux Namespaces – mount

Content List

1. Abstract

Linux Namespaces provides isolation for processes which work on same environment by separating the system resources. Mount Namespace was the firstly developed type of the Linux Namespace group. It allow processes to create their own private area. This feature makes Mount Namespace critical to use on multi-user environments. This report gives three test cases about the mount namespace and mentions two critical usage area by looking over two open source projects which are developed by Google and CloudLinux.

2. Linux Namespaces

2.1 What are Linux Namespaces?

Namespaces are implemented to provide isolation and lightweight virtualization of the global system resources between processes. Currently, Linux Kernel provides seven different namespaces (Kerrisk, namespaces, n.d.).

- Cgroup
- IPC (Inter Process Communication)
- Network
- Mount
- PID (Process ID)
- User
- UTS

Namespaces allow operating system to implement container structure. Some significant parts of the LXC (LinuX Containers) relies on isolation feature of the Linux Kernel provided by namespace structure.

Linux Namespaces are similar to Zones in Solaris Operating Systems. Unlike KVM or Xen, there is no hypervisor layer on the Linux Namespaces structure.

2.2 What is "Isolation"?

Isolation enables programs or processes to have different and separate views of the system plane than others.

2.3 Why to isolate?

For single-user usage of the operating system, there is no urgency for the isolation of processes. But for the multi-user systems or for the servers it is crucial to separate process planes from each other as possible. In this way each program or process run on completely different s/w plane.

3. Mount Namespace

3.1 Introduction about the Mount Namespace

Mount namespace was the first implemented namespace type in the Linux (Kerrisk, LWN.net, 2016). The aim of the mount namespace is creating different filesystem trees for each user and container by isolating the list of mount points of each processes. In other words, each mount namespace has own particular list of mount points and has different view of the file system hierarchy. Also, processes under different namespace groups are able to change the mountpoints while others not affected.

3.2 Usage Details of Mount Namespace and executables

After the boot of the operating system, there is only "initial namespace" mounted. Processes may share initial name space or may create their private mount areas. Built-in userland *mount, umount* and *unshare* executables are used to manage mount operations. In general, these commands need root privileges to run successfully.

The tables below show attributes and functionalities of mount and share executables (HelpManual, n.d.).

Operations for **mount** executable:

-B, --bind	mount a subtree somewhere else (same as -o bind)
-M, --move	move a subtree to some other place
-R, --rbind	mount a subtree and all submounts somewhere else
--make-shared	mark a subtree as shared
--make-slave	mark a subtree as slave
--make-private	mark a subtree as private
--make-unbindable	mark a subtree as unbindable
--make-rshared	recursively mark a whole subtree as shared
--make-rslave	recursively mark a whole subtree as slave
--make-rprivate	recursively mark a whole subtree as private
--make-runbindable	recursively mark a whole subtree as unbindable

Related options for **unshare** executable:

-m, --mount[=<file>]	unshare mounts namespace
-U, --user[=<file>]	unshare user namespace
-f, --fork	fork before launching <program>
--mount-proc[=<dir>]	mount proc filesystem first (implies --mount)
-r, --map-root-user	map current user to root (implies --user)
--propagation slave|shared|private|unchanged	modify mount propagation in mount namespace

3.3. Too great isolation was problematic! – Shared Subtrees

Naïve(initial) implementation of the mount namespaces was providing too great isolation between the namespaces which makes hard to propagate new mounted objects. For example, in the naïve implementation each namespace should mount the newly added object (for example when USB drive plugged) separately. To solve that developers added "shared subtrees feature" in Linux 2.6.15 (KernelNewbises, 2006). This fresh feature provides automatic and controlled propagation of updates about mount point events for each namespace.

On shared subtree design, each mount point has "propagation type" tag which shows how new mount operations will be propagated to other namespaces. There are 4 propagation types;

MS_SHARED: It shares any mount or unmount events with same-group peers. Additionally, it emits events from other same-group peers.

MS_PRIVATE: New mount points do not propagate to other peers or does not emit an event from other peers

MS_SLAVE: Mixture of the share and private types. It has master peer group. If any master peer propagates mound or unmount event to the slave but not vice-versa. A mount point can be the slave of another peer group while at the same timesharing mount events with a group of which it is a member. (Debian, 2017)

MS_UNBINDABLE: It is similar to MS_PRVATE with a single difference. Mount in this type can not be selected as a source of bind.

4. Tests and Use Cases

4.1 Basic Test – Two separate Mount Namespaces using shell

Following shell commands creates two different mount namespaces by using. But to imitate block devices (sda1 etc.) we should create fake ones.

```
$ dd if=/dev/zero of=/root/fake_file1 bs=1M count=100
# creates 100MB file with random content
$ mkfs.ext4 /root/fake_file1
# change filesystem for image
$ losetup /dev/loop1 fake_file1
# assigns fake file to loopback device
```

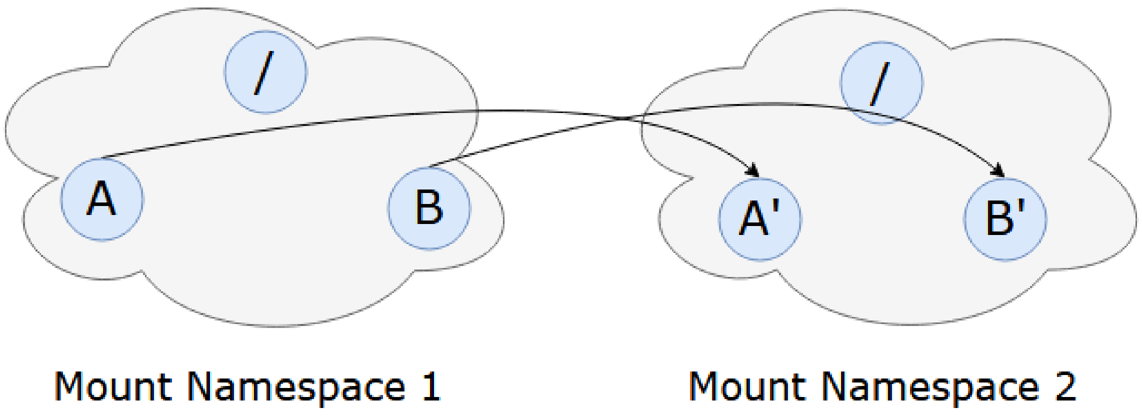
We may create as many devices as we want. Assume we created loopback devices loop1, loop2, loop3 etc. Then we may start to test mount namespaces.

```
# on shell 1
$ mount --make-private /
$ mount --make-shared /dev/loop1 /A
$ mount --make-shared /dev/loop2 /B
```

As stated before, mount command may require root privileges. After shell1 commands executed, shell2 runs following command to separate its running mount environment.

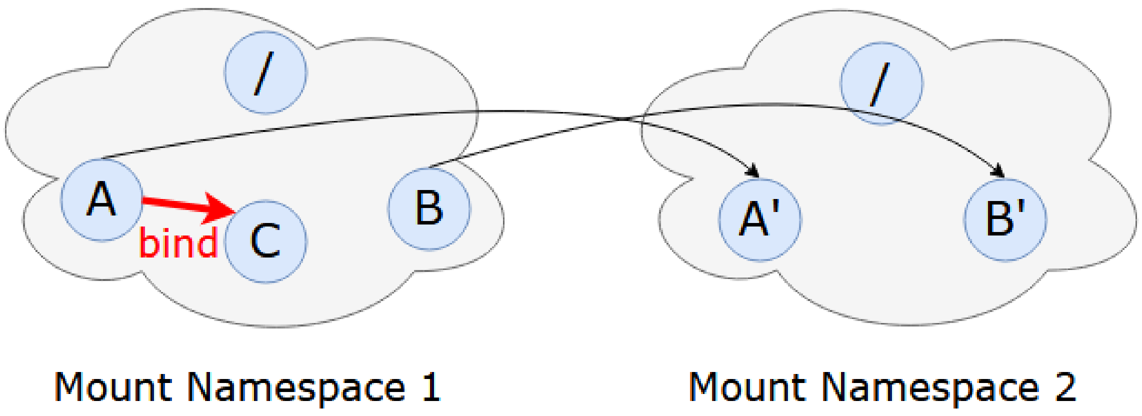
```
# on shell 2
$ unshare -m --propagation unchanged sh
```

The -m parameter provided to create a completely new mount namespace and propagation parameter is related with "shared subtrees feature". Then our mount namespaces can be visualized as shown in following graphic.



After this setup, shell1 and shell2 have different mount namespaces. Lets make a change on mounting state on the one of these shells.

```
# on shell 1
$ mkdir /C
$ mount –bind /A /C
# create a bind mount to C from A
```



This example creates two separate peer groups. Since we have marked parent root "/" as a private for the first shell, changes on shell1 after "*unshare*" were not propagated to second namespace. So, excluding A' and B'; any change on parent root done by the sheel1 can not be seen by sheel2 and even root privileged any user.

4.2 Slavery Test: MS_SLAVE example

Creating a mounting point as a slave allows process to receive any propagated mounting events. It will only get events which are propagated by master peer group but it won't be able to send events to its' master (like ancient master-slave relationship). It is practical to use when developer want to receive a mount event which is propagated by initial or parent mount namespace (for example when USB plugged) while not propagating its own event.

Assume we have two pre-exist mount points in our structure.

```
# on shell 1
$ mount --make-shared /A
$ mount --make-shared /B
```

And assume, second process creates a new mount namespace by using *unshare* executable (or unshare function on C code).

```
# on shell 2
$ unshare -m –-propagation unchanged /bin/bash
```

Then, second process decides to select mount point **A** to make it slave. At that point, notice that we are slaving the already shared mount point which makes it a slave of the peer group.

```
#on shell2
$ mount --make-slave /A
# commands below will be explained later
$ mkdir /A/ImTheMaster; mount/dev/pool3 /A/ImTheMaster
# on shell1
$ mkdir /B/ImThePoorSlave; mount /dev/pool4 /B/ ImThePoorSlave
```

Finally, we may inspect the content of *mountinfo.*

```
481 536 6:13 / /A rw, relatime master:1
482 536 6:12 / /B rw, relatime shared:2
```

At that point by creating and listing directories or files we can detect that the mount points in the second namespace will be propagated to first one, but not vice-versa.

4.3 Extended Test: Private Filesystem Mount Points per Process

We may extend the simple example. What if we apply same methodology for each process on the system? What will happen and what will be the difference?

```
$ unshare -m /bin/bash
$ private_dir=`mktemp -d --tmpdir=/privateTmp`
$ mount -n -o size=8m -t tmpfs tmpfs $private_dir
```

Each time a process runs the commands above (over shell or C code etc.) it will have own private mount point and there is no way to reach that section for other processes. In other words, other processes has no visibility over that section. To illustrate, first process my create an empty file on its private folder;

```
$ cd $private_dir  # assume private_dir = tmp.ceng536odtu
$ touch ThisFileIsPrivateForMe.txt
```

Then, other process may query try to list or reach that private area;

```
$ ls -lFa /privateTmp/tmp.ceng536odtu
# total 8
# drwxr-xr-x 2 root root 4096 Jan 22 20:49 ./
# drwxrwxrwt 8 root root 4096 Jan 22 20:49 ../
```

It means secondary (or all process except which creates private area) are not able to see content of that mounted space.

In conclusion, we may use that methodology to create private temporal file directory for each process. Notice that, this private area can not visible to other processes even root-privileged ones. This security measurement protects victim processes from naïve malicious attackers who tries to get access to files which is not owned by him/her.

4.4 Use Case: Sandboxing – Google nsjail / Capture the Flag

The isolation is the key feature for all Linux Namespace types. There is an open-source project which is developed and distributed by Google, namely nsjail which based on the Linux Namespaces. Google uses extended version this project to host Capture the Flag styled security challenges or competitions (Google, 2018).

Main idea behind the project is separating the planes which owned or controlled by competitors. So, any competitor can not be able to see others' codes or works. Also, this project some work about includes the separation of mount namespaces using *chroot* and *mount*. Many cloud systems like Amazon or Google Cloud Platform are benefit from the isolation nature of the Linux Namespaces.

To be more specific, Mount Namespace is used on that project to provide separate working environment to competitors in means of "files" or "directories".

4.5 Use Case: Shared Hosting Security – CageFS / Separate User Spaces

Many hosting companies provides secure spaces to their high-numbered users on same server. This would be a huge expense and time-consumption if they had to install separate operating systems, virtual machines or programs for each user. So that, developers worked around that and found a solution based on the isolation which is provided by Linux Namespaces again.

CageFS (feature of the CloudLinux software) creates individual and private namespace for each user which blocks malicious user (or normal user, in fluke) to see other's files and creating high level of isolation (CloudLinux, n.d.). In other (shiny) words, each user works on its own 'cage'.

In conclusion; on the system which is running on CloudLinux – CageFS structure, only safe binaries will be available for users and they won't be able to see server configuration files. Also users won't be able to detect presence of other users on the server and can't search or find other users' files.

5. References

CloudLinux. (n.d.). *CageFS - Mount Points*. Retrieved from https://docs.cloudlinux.com/index.html?mount_points.html

Debian. (2017, 07 13). *mount(2) — manpages-dev — Debian testing — Debian Manpages*. Retrieved from https://manpages.debian.org/testing/manpages-dev/mount.2.en.html

Google. (2018, 01 08). *NsJail*. Retrieved from https://github.com/google/nsjail

HelpManual. (n.d.). *mount*. Retrieved from https://helpmanual.io/help/mount/

HelpManual. (n.d.). *unshare*. Retrieved from https://helpmanual.io/help/unshare/

KernelNewbises. (2006). *Linux_2_6_15 - Linux Kernel Newbies*. Retrieved from https://kernelnewbies.org/Linux_2_6_15

Kerrisk, M. (2016, June 8). Retrieved from LWN.net: https://lwn.net/Articles/689856/

Kerrisk, M. (n.d.). *namespaces*. Retrieved from Linux Programmer's Manual: http://man7.org/linux/man-pages/man7/namespaces.7.html