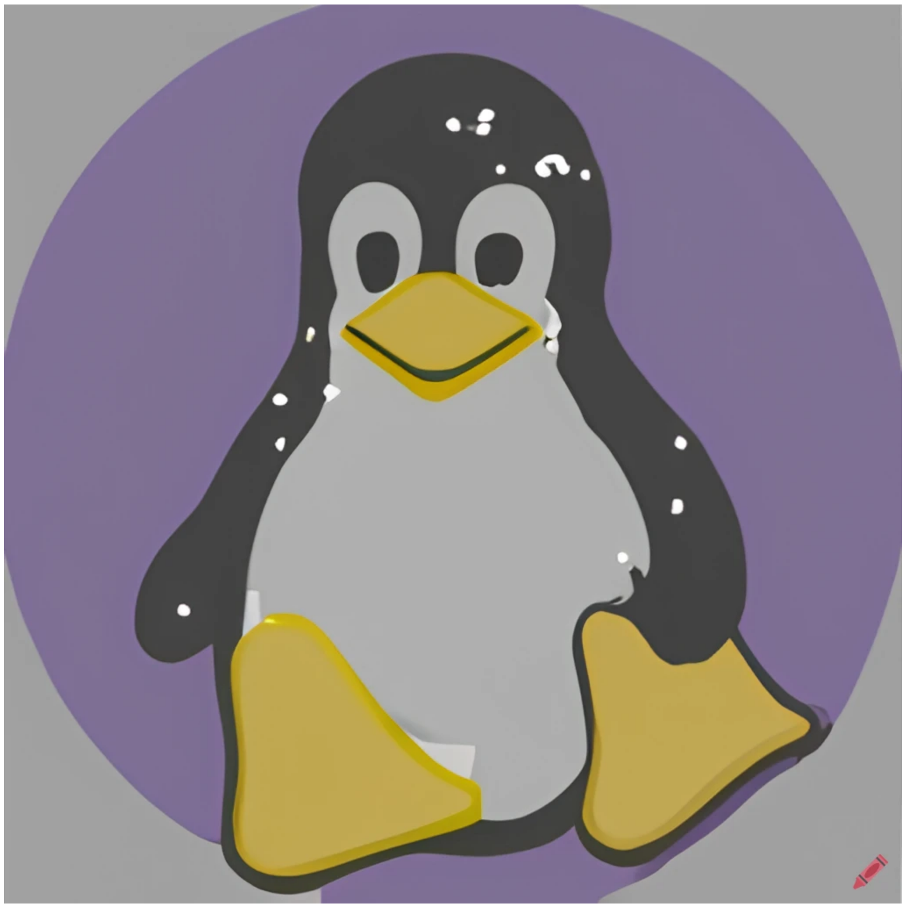
Linux Namespaces

March-2023

Version 1.0 (beta) - not a final version

****missing user & cgroup namespace****

By Shlomi Boutnaru



Created using [Craiyon, AI Image Generator]

Introduction to Namespaces **3**

PID namespace **4**

UTS namespace **5**

Time namespace **6**

Network namespace **7**

IPC namespace **8**

Mount Namespace **9**

Introduction to Namespaces

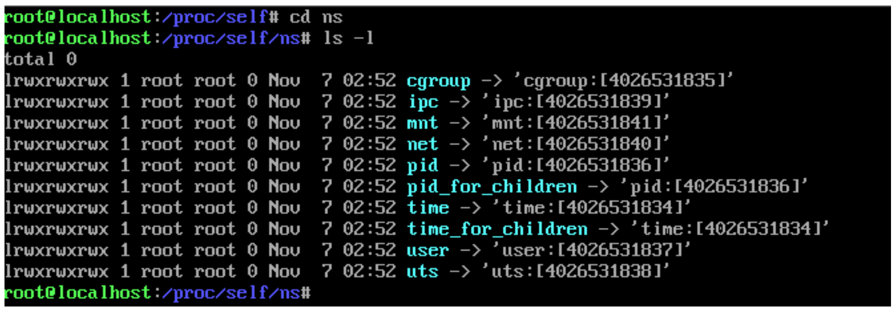
The goal of namespaces is to create an illusion that resources seen by a process are the only resources the system has. Basically we can think about namespace as an isolation technology for processes which are sharing the same kernel (versus processes running with different kernels such as two VMs on the same hypervisor). Namespaces were first introduced as part of kernel 2.4.19. By the way, namespaces are one of the basic building blocks of containers.

Thus, if we have two different processes (that are part of different namespaces) a change made by one of them on a resource (that is part of one of the namespaces) is not visible to the second process (unless there is a bug/security vulnerability). More information about namespace can be found using “man 7 namespaces”[1].

Moreover, in order to use namespaces we can leverage different syscalls: clone, setns, unshare and ioctl_ns. When using “clone” (“man 2 clone”) to create new processes we can pass different flags that will create new namespaces that we want the new process to be part of. With “setns” (“man 2 setns”) we can add a process to an existing namespace. By using “unshre” (“man 2 unshare” we can also move a process to a new namespace (it is different from clone because it is not before the creation of the process). Finally, with “ioctl_ns” (“man 2 ioctl_ns”) to perform operations such as discovery regarding namespaces.

It is important to know that, today there are 8 different types of namespaces: User, PID, UTS (hostname), Time, IPC, Network, Cgroup and Mount. I am going to elaborate on each and one of them in a separate writeup. Also, there is a suggestion to add a syslog[2] namespace which was not merged to the kernel until now.

We can use “/proc” (“man 5 proc”) to query information about namespaces of the running process. The information about namespaces is stored under “/proc/[pid]/ns” - as shown in the screenshot below (taken using copy.sh). Every link shown in the screenshot corresponds to a namespace. The two links ending with “for_children” represent the information about the namespaces of the child processes created by the process.



```
root@localhost:/proc/self# cd ns
root@localhost:/proc/self/ns# ls -l
total 0
lrwxrwxrwx 1 root root 0 Nov  7 02:52 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 Nov  7 02:52 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 root root 0 Nov  7 02:52 mnt -> 'mnt:[4026531841]'
lrwxrwxrwx 1 root root 0 Nov  7 02:52 net -> 'net:[4026531840]'
lrwxrwxrwx 1 root root 0 Nov  7 02:52 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 Nov  7 02:52 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 Nov  7 02:52 time -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Nov  7 02:52 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Nov  7 02:52 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 Nov  7 02:52 uts -> 'uts:[4026531838]'
root@localhost:/proc/self/ns#
```

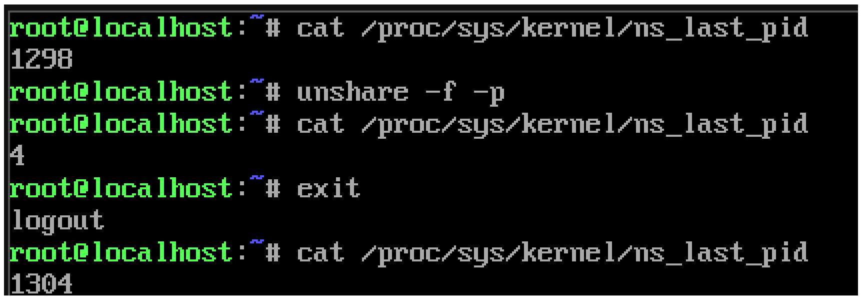
[1] https://man7.org/linux/man-pages/man7/namespaces.7.html
[2] https://lwn.net/Articles/562389

PID namespace

The goal of PID namespaces is to isolate the “Process ID number space”. Thus, different processes in distinct PID namespaces can have the same PID. When a new PID namespace is started the first process gets PID 1 (so we don’t have a new swapper[3]). In order to use PID namespaces we have to ensure that our kernel was compiled with “CONFIG_PID_NS” enabled[4].

Moreover, PID namespace can also be nested , since kernel 3.7 the maximum nesting depth is 32. A process is visible to every other process in the same PID namespace or any direct ancestor PID namespace. The opposite way does not work, a process in a child PID namespace can’t see a process in a parent PID namespace[5].

Also, “/proc” will show only processes which are visible in the PID namespace of the process that executed the “mount” operation for “/proc”[6]. If we want to see the number of the last pid that was allocated in our PID namespace we can use “/proc/sys/kernel/ns_last_pid”[7] - as you can see in the screenshot below.



```
root@localhost:~# cat /proc/sys/kernel/ns_last_pid
1298
root@localhost:~# unshare -f -p
root@localhost:~# cat /proc/sys/kernel/ns_last_pid
4
root@localhost:~# exit
logout
root@localhost:~# cat /proc/sys/kernel/ns_last_pid
1304
```

Lastly, a nice fact to know is that when we pass a pid over a unix domain socket to a process which belongs to another PID namespace, it is resolved to the correct value in the receiving process’ PID namespace[8].

[3] https://medium.com/@boutnaru/the-linux-process-journey-pid-0-swapper-7868d1131316
[4] https://man7.org/linux/man-pages/man7/pid_namespaces.7.html
[5] https://www.schutzwerk.com/en/blog/linux-container-namespaces03-pid-net/
[6] https://lwn.net/Articles/531419/
[7] https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#ns-last-pid
[8] https://man7.org/linux/man-pages/man7/pid_namespaces.7.html

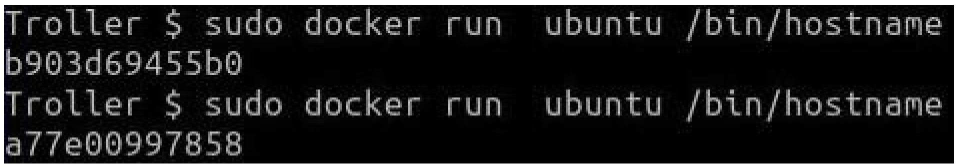
UTS namespace

In the first part of the series we have talked generally about what namespaces are and what we can do with them. Now we are going to deep dive to the different namespaces starting with UTS (Unix Time Sharing).

By using UTS namespaces we can separate/isolate/segregate the hostname and the NIS (Network Information Service) domain of a Linux system. Just to clarify, NIS is a directory service (it has some similarities to Microsoft's Active Directory) that was created by Sun and later was discontinued by Oracle . Thus, UTS today is focused mainly on separating hostname.

In order to get the information described above we can use the following syscalls: uname ("man 2 uname"), gethostname ("man 2 gethostname") and getdomainname ("man 2 getdomainname") - we also have a parallel set syscall for each one of them. For supporting UTS namespaces the kernel should be compiled with "CONFIG_UTS_NS".

Container engines (such as docker) use different namespaces to isolate between different containers, even if they were created from the same image - as shown in the screenshot below.



```
Troller $ sudo docker run  ubuntu /bin/hostname
b903d69455b0
Troller $ sudo docker run  ubuntu /bin/hostname
a77e00997858
```

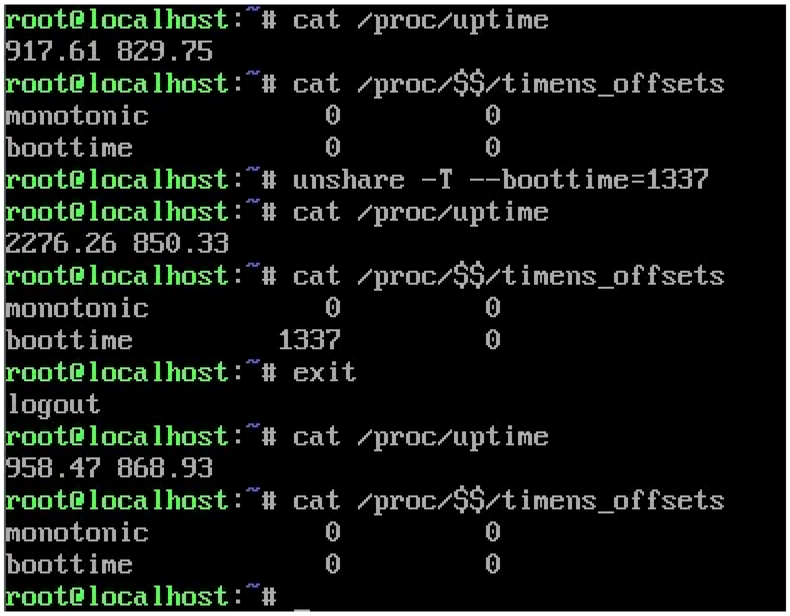
Time namespace

By using time namespaces we can separate/isolate/segregate and thus virtualize the values of two system clocks[9]. We are talking about “CLOCK_MONOTONIC” and “CLOCK_BOOTTIME”.

“CLOCK_MONOTONIC” goal is to represent an absolute elapsed wall-clock time since some arbitrary, fixed point in the past. It is important to understand that it isn't affected by changes in the system time-of-day clock. As opposed to “CLOCK_REALTIME” which can change based on configurations/NTP (Network Time Protocol) data[10]. Moreover, “CLOCK_MONOTONIC” does not measure time spent during suspend. If we want a monotonic clock that is running during suspend we need to use “CLOCK_BOOTTIME” (https://linux.die.net/man/2/clock_gettime).

Thus, all the processes in the same time namespace share the same values of the clocks explained above. Due to that, it affect the results of the following syscalls: timer_settime (“man 2 timer_settime”, clock_gettime (“man 2 clock_gettime”), clock_nanosleep (“man 2 clock_nanosleep”), timerfd_settime (“man 2 timerfd_settime”). Also, the content returned from “/proc/uptime” is affected.

In order to create a new time namespace we have to use the unshare syscall (“man 2 unshare”) and pass the “CLONE_NEWTIME” flag. You can see an example of that using the “unshare” cli tool (“man 1 unshare”) in the screenshot below. In order to see the difference between the initial time namespace and the process’ time namespace we can use “/proc/[PID]/timens_offsets” also shown in the screenshot below.



```
root@localhost:~# cat /proc/uptime
917.61 829.75
root@localhost:~# cat /proc/$$/timens_offsets
monotonic           0         0
boottime            0         0
root@localhost:~# unshare -T --boottime=1337
root@localhost:~# cat /proc/uptime
2276.26 850.33
root@localhost:~# cat /proc/$$/timens_offsets
monotonic           0         0
boottime         1337         0
root@localhost:~# exit
logout
root@localhost:~# cat /proc/uptime
958.47 868.93
root@localhost:~# cat /proc/$$/timens_offsets
monotonic           0         0
boottime            0         0
root@localhost:~# _
```

[9] https://man7.org/linux/man-pages/man7/time_namespaces.7.html
[10] https://stackoverflow.com/questions/3523442/difference-between-clock-realtime-and-clock-monotonic

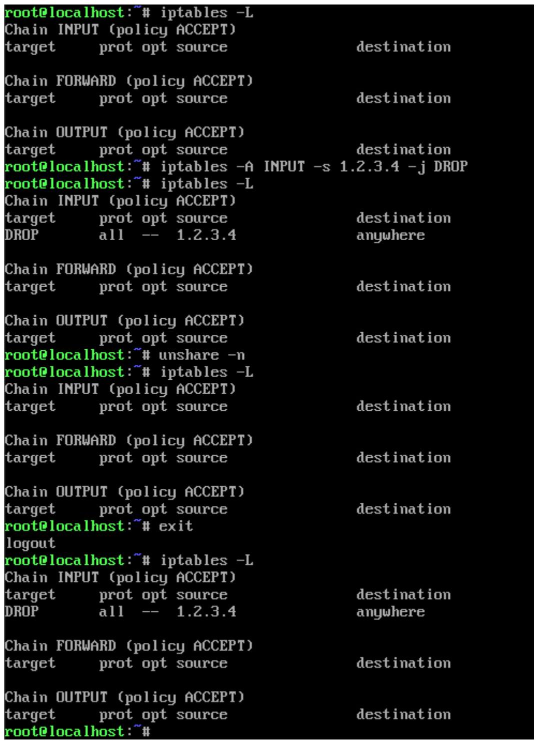
Network namespace

First, in order for the kernel to support network namespaces we need to compile the kernel with "CONFIG_NET_NS" enabled. Overall, network namespaces can separate/isolate/segregate between the different system resources which are associated with networking under Linux. Among those resources are: firewall rules, routing tables (IP), IPv4 and IPv6 protocol stacks, sockets, different directories related to the networking subsystem (like: "/proc/[PID]/net", "/proc/sys/net", "/sys/class/net" and more), etc[11].

By the way, unix domain sockets are also isolated using network namespaces ("man 7 unix"). It is important to understand that a physical network device can exist in one network namespace at a time (singleton). In case the last process in a network namespace returns/exits, Linux frees the namespace which moves the physical network device to the initial network namespace.

Moreover, in case we want to create a bridge to a network device which is part of a different namespace we can use a virtual network device. It can create tunnels between network namespaces[12].

Lastly, you can see an example of creating a network namespace in the screenshot below. As you can see an iptables rule is created but it is not relevant to the newly created network namespace.



```
root@localhost:~# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@localhost:~# iptables -A INPUT -s 1.2.3.4 -j DROP
root@localhost:~# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  1.2.3.4              anywhere

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@localhost:~# unshare -n
root@localhost:~# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@localhost:~# exit
logout
root@localhost:~# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  1.2.3.4              anywhere

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@localhost:~#
```

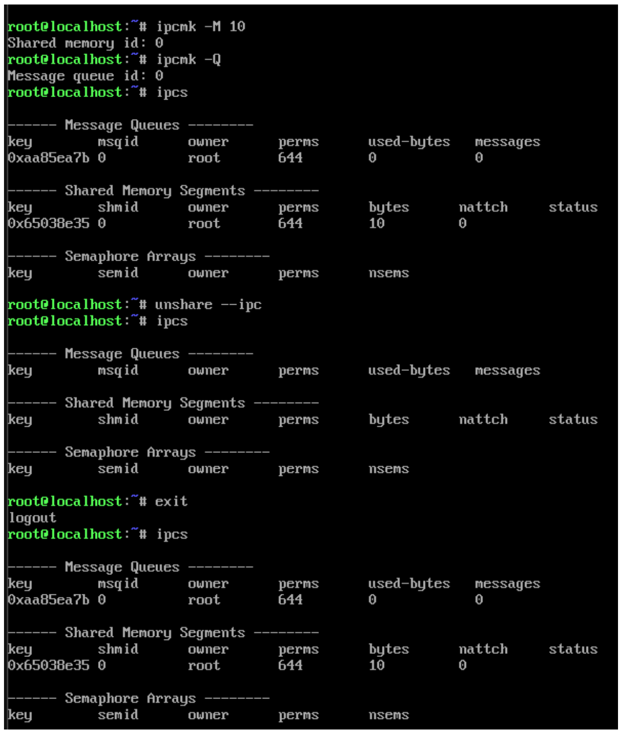
[11] https://man7.org/linux/man-pages/man7/network_namespaces.7.html
[12] https://man7.org/linux/man-pages/man4/veth.4.html

IPC namespace

The goal of the IPC namespace is to isolate between different IPC resources like message queues, semaphores and shared memory. We are talking both on System V IPC objects[13] and POSIX message queues[14]. In order to use "IPC namespaces" the kernel should be compiled with CONFIG_IPC_NS enabled[15].

We can use "/proc" in order to retrieve information about the different IPC objects in an "IPC namespace". Regarding POSIX message queues we have "/proc/sys/fs/mqueue". In case of System V IPC objects we have "/proc/sysvipc" and specific file in "/proc/sys/kernel" (msgmax, msgmnb, msgmni, sem, shmall, shmmax, shmmni, and shm_rmid_forced). For more information I suggest reading proc's man page[16].

Lastly, all IPC objects created in an "IPC namespace" are visible only to all processes/tasks that are members of the same namespace - as shown in the screenshot below. In the demonstration the namespace was created using "unshare"[17], the IPC resource was created using "ipcmk"[18] and the show information about System V IPC resources using "ipcs"[19].



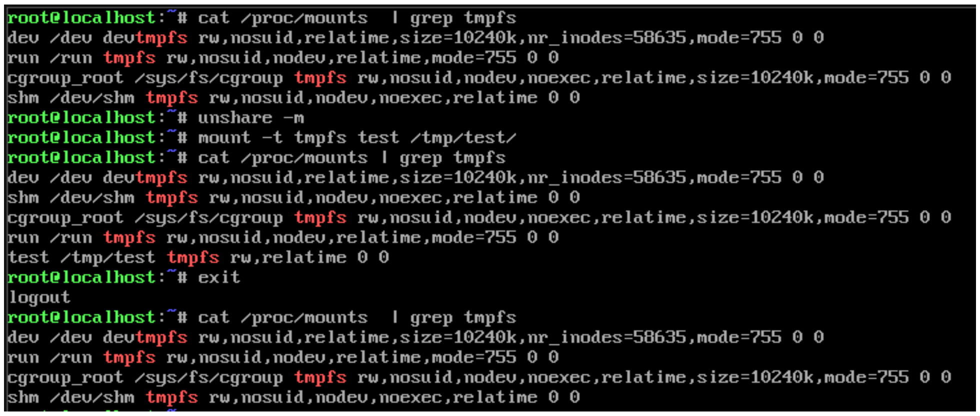
```
root@localhost:~# ipcmk -M 10
Shared memory id: 0
root@localhost:~# ipcmk -Q
Message queue id: 0
root@localhost:~# ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0xaa85ea7b 0          root       644        0            0

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x65038e35 0          root       644        10         0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems

root@localhost:~# unshare --ipc
root@localhost:~# ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems

root@localhost:~# exit
logout
root@localhost:~# ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0xaa85ea7b 0          root       644        0            0

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x65038e35 0          root       644        10         0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

[13] https://man7.org/linux/man-pages/man7/sysvipc.7.html
[14] https://man7.org/linux/man-pages/man7/mq_overview.7.html
[15] https://man7.org/linux/man-pages/man7/ipc_namespaces.7.html
[16] https://man7.org/linux/man-pages/man5/proc.5.html
[17] https://man7.org/linux/man-pages/man1/unshare.1.html
[18] https://man7.org/linux/man-pages/man1/ipcmk.1.html
[19] https://man7.org/linux/man-pages/man1/ipcs.1.html

Mount Namespace

The goal of mount namespaces is to provide isolation regarding the list of mounts as seen by the process/tasks in each namespace. By doing so different processes/tasks that belong to different mount namespaces will see distinct directories hierarchies[20].

Using "/proc" we can inspect the mounting points visible for specific processes using "/proc/[PID]/mounts, "/proc/[PID]/mountinfo" and "/proc/[PID]/mountstats"[21]. You can see the different outputs when reading the data in different mount namespace - as seen in the screenshot below.



```
root@localhost:~# cat /proc/mounts  | grep tmpfs
dev /dev devtmpfs rw,nosuid,relatime,size=10240k,nr_inodes=58635,mode=755 0 0
run /run tmpfs rw,nosuid,nodev,relatime,mode=755 0 0
cgroup_root /sys/fs/cgroup tmpfs rw,nosuid,nodev,noexec,relatime,size=10240k,mode=755 0 0
shm /dev/shm tmpfs rw,nosuid,nodev,noexec,relatime 0 0
root@localhost:~# unshare -m
root@localhost:~# mount -t tmpfs test /tmp/test/
root@localhost:~# cat /proc/mounts | grep tmpfs
dev /dev devtmpfs rw,nosuid,relatime,size=10240k,nr_inodes=58635,mode=755 0 0
shm /dev/shm tmpfs rw,nosuid,nodev,noexec,relatime 0 0
cgroup_root /sys/fs/cgroup tmpfs rw,nosuid,nodev,noexec,relatime,size=10240k,mode=755 0 0
run /run tmpfs rw,nosuid,nodev,relatime,mode=755 0 0
test /tmp/test tmpfs rw,relatime 0 0
root@localhost:~# exit
logout
root@localhost:~# cat /proc/mounts  | grep tmpfs
dev /dev devtmpfs rw,nosuid,relatime,size=10240k,nr_inodes=58635,mode=755 0 0
run /run tmpfs rw,nosuid,nodev,relatime,mode=755 0 0
cgroup_root /sys/fs/cgroup tmpfs rw,nosuid,nodev,noexec,relatime,size=10240k,mode=755 0 0
shm /dev/shm tmpfs rw,nosuid,nodev,noexec,relatime 0 0
```

After the implementation of mount namespaces the isolation they have created a problem. Think about a case when we add some device that we want to be visible on every namespace, for that we need to execute a "mount" command on each namespace. To overcome this the shared subtree feature was introduced in kernel 2.6.15[22].

By using shared subtrees we can propagate mount/unmount events between distinct mount namespaces[23]. It is designed to work between mounts that are members of the same peer group. Thus, "peer group" is defined as a group of vfsmounts that propagate events between each other[24]. Also, we can control the propagation using the mount system call by passing one of the following "mountflags": MS_SHARED, MS_PRIVATE, MS_SLAVE, or MS_UNBINDABLE[25]. For a detailed description of each flag I encourage you to read the following link https://hechao.li/2020/06/09/Mini-Container-Series-Part-1-Filesystem-Isolation/. For more information about shared subtrees I suggest reading the kernel documentation[26].

[20] https://man7.org/linux/man-pages/man7/mount_namespaces.7.html
[21] https://man7.org/linux/man-pages/man5/proc.5.html
[22] https://lwn.net/Articles/689856/
[23]https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/storage_administration_guide/sect-using_the_mount_command-mounting-bind
[24] https://www.redhat.com/sysadmin/mount-namespaces
[25] https://man7.org/linux/man-pages/man2/mount.2.html
[26] https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt