**Namespaces**

**PID namespace**

## Basic Information

The PID (Process IDentifier) namespace is a feature in the Linux kernel that provides process isolation by enabling a group of processes to have their own set of unique PIDs, separate from the PIDs in other namespaces. This is particularly useful in containerization, where process isolation is essential for security and resource management.

When a new PID namespace is created, the first process in that namespace is assigned PID 1. This process becomes the "init" process of the new namespace and is responsible for managing other processes within the namespace. Each subsequent process created within the namespace will have a unique PID within that namespace, and these PIDs will be independent of PIDs in other namespaces.

From the perspective of a process within a PID namespace, it can only see other processes in the same namespace. It is not aware of processes in other namespaces, and it cannot interact with them using traditional process management tools (e.g., kill, wait, etc.). This provides a level of isolation that helps prevent processes from interfering with one another.

### How it works:

1. When a new process is created (e.g., by using the clone() system call), the process can be assigned to a new or existing PID namespace. **If a new namespace is created, the process becomes the "init" process of that namespace**.
2. The **kernel** maintains a **mapping between the PIDs in the new namespace and the corresponding PIDs** in the parent namespace (i.e., the namespace from which the new namespace was created). This mapping **allows the kernel to translate PIDs when necessary**, such as when sending signals between processes in different namespaces.
3. **Processes within a PID namespace can only see and interact with other processes in the same namespace**. They are not aware of processes in other namespaces, and their PIDs are unique within their namespace.
4. When a **PID namespace is destroyed** (e.g., when the "init" process of the namespace exits), **all processes within that namespace are terminated**. This ensures that all resources associated with the namespace are properly cleaned up.

## Lab:

### Create different Namespaces

#### CLI

Copy

sudo unshare -pf --mount-proc /bin/bash

Error: bash: fork: Cannot allocate memory

By mounting a new instance of the /proc filesystem if you use the param --mount-proc, you ensure that the new mount namespace has an **accurate and isolated view of the process information specific to that namespace**.

#### Docker

Copy

docker run -ti --name ubuntu1 -v /usr:/ubuntu1 ubuntu bash

### Check which namespace are your process in

Copy

ls -l /proc/self/ns/pid

lrwxrwxrwx 1 root root 0 Apr 3 18:45 /proc/self/ns/pid -> 'pid:[4026532412]'

### Find all PID namespaces

Copy

sudo find /proc -maxdepth 3 -type l -name pid -exec readlink {} \; 2>/dev/null | sort -u

Note that the root use from the initial (default) PID namespace can see all the processes, even the ones in new PID names paces, thats why we can see all the PID namespaces.

### Enter inside a PID namespace

Copy

nsenter -t TARGET\_PID --pid /bin/bash

When you enter inside a PID namespace from the default namespace, you will still be able to see all the processes. And the process from that PID ns will be able to see the new bash on the PID ns.

Also, you can only **enter in another process PID namespace if you are root**. And you **cannot** **enter** in other namespace **without a descriptor** pointing to it (like /proc/self/ns/pid)

## References

* <https://stackoverflow.com/questions/44666700/unshare-pid-bin-bash-fork-cannot-allocate-memory>

**Mount namespace**

## Basic Information

A mount namespace is a Linux kernel feature that provides isolation of the file system mount points seen by a group of processes. Each mount namespace has its own set of file system mount points, and **changes to the mount points in one namespace do not affect other namespaces**. This means that processes running in different mount namespaces can have different views of the file system hierarchy.

Mount namespaces are particularly useful in containerization, where each container should have its own file system and configuration, isolated from other containers and the host system.

### How it works:

1. When a new mount namespace is created, it is initialized with a **copy of the mount points from its parent namespace**. This means that, at creation, the new namespace shares the same view of the file system as its parent. However, any subsequent changes to the mount points within the namespace will not affect the parent or other namespaces.
2. When a process modifies a mount point within its namespace, such as mounting or unmounting a file system, the **change is local to that namespace** and does not affect other namespaces. This allows each namespace to have its own independent file system hierarchy.
3. Processes can move between namespaces using the setns() system call, or create new namespaces using the unshare() or clone() system calls with the CLONE\_NEWNS flag. When a process moves to a new namespace or creates one, it will start using the mount points associated with that namespace.
4. **File descriptors and inodes are shared across namespaces**, meaning that if a process in one namespace has an open file descriptor pointing to a file, it can **pass that file descriptor** to a process in another namespace, and **both processes will access the same file**. However, the file's path may not be the same in both namespaces due to differences in mount points.

## Lab:

### Create different Namespaces

#### CLI

Copy

sudo unshare -m [--mount-proc] /bin/bash

By mounting a new instance of the /proc filesystem if you use the param --mount-proc, you ensure that the new mount namespace has an **accurate and isolated view of the process information specific to that namespace**.

Error: bash: fork: Cannot allocate memory

#### Docker

Copy

docker run -ti --name ubuntu1 -v /usr:/ubuntu1 ubuntu bash

### Check which namespace is your process in

Copy

ls -l /proc/self/ns/mnt

lrwxrwxrwx 1 root root 0 Apr 4 20:30 /proc/self/ns/mnt -> 'mnt:[4026531841]'

### Find all Mount namespaces

Copy

sudo find /proc -maxdepth 3 -type l -name mnt -exec readlink {} \; 2>/dev/null | sort -u

# Find the processes with an specific namespace

sudo find /proc -maxdepth 3 -type l -name mnt -exec ls -l {} \; 2>/dev/null | grep <ns-number>

### Enter inside a Mount namespace

Copy

nsenter -m TARGET\_PID --pid /bin/bash

Also, you can only **enter in another process namespace if you are root**. And you **cannot** **enter** in other namespace **without a descriptor** pointing to it (like /proc/self/ns/mnt).

Because new mounts are only accessible within the namespace it's possible that a namespace contains sensitive information that can only be accessible from it.

### Mount something

Copy

# Generate new mount ns

unshare -m /bin/bash

mkdir /tmp/mount\_ns\_example

mount -t tmpfs tmpfs /tmp/mount\_ns\_example

mount | grep tmpfs # "tmpfs on /tmp/mount\_ns\_example"

echo test > /tmp/mount\_ns\_example/test

ls /tmp/mount\_ns\_example/test # Exists

# From the host

mount | grep tmpfs # Cannot see "tmpfs on /tmp/mount\_ns\_example"

ls /tmp/mount\_ns\_example/test # Doesn't exist

## References

* <https://stackoverflow.com/questions/44666700/unshare-pid-bin-bash-fork-cannot-allocate-memory>

**Network namespace**

## Basic Information

A network namespace is a Linux kernel feature that provides isolation of the network stack, allowing **each network namespace to have its own independent network configuration**, interfaces, IP addresses, routing tables, and firewall rules. This isolation is useful in various scenarios, such as containerization, where each container should have its own network configuration, independent of other containers and the host system.

### How it works:

1. When a new network namespace is created, it starts with a **completely isolated network stack**, with **no network interfaces** except for the loopback interface (lo). This means that processes running in the new network namespace cannot communicate with processes in other namespaces or the host system by default.
2. **Virtual network interfaces**, such as veth pairs, can be created and moved between network namespaces. This allows for establishing network connectivity between namespaces or between a namespace and the host system. For example, one end of a veth pair can be placed in a container's network namespace, and the other end can be connected to a **bridge** or another network interface in the host namespace, providing network connectivity to the container.
3. Network interfaces within a namespace can have their **own IP addresses, routing tables, and firewall rules**, independent of other namespaces. This allows processes in different network namespaces to have different network configurations and operate as if they are running on separate networked systems.
4. Processes can move between namespaces using the setns() system call, or create new namespaces using the unshare() or clone() system calls with the CLONE\_NEWNET flag. When a process moves to a new namespace or creates one, it will start using the network configuration and interfaces associated with that namespace.

## Lab:

### Create different Namespaces

#### CLI

Copy

sudo unshare -n [--mount-proc] /bin/bash

# Run ifconfig or ip -a

By mounting a new instance of the /proc filesystem if you use the param --mount-proc, you ensure that the new mount namespace has an **accurate and isolated view of the process information specific to that namespace**.

Error: bash: fork: Cannot allocate memory

1. **Problem Explanation**:
   * The Linux kernel allows a process to create new namespaces using the unshare system call. However, the process that initiates the creation of a new PID namespace (referred to as the "unshare" process) does not enter the new namespace; only its child processes do.
   * Running %unshare -p /bin/bash% starts /bin/bash in the same process as unshare. Consequently, /bin/bash and its child processes are in the original PID namespace.
   * The first child process of /bin/bash in the new namespace becomes PID 1. When this process exits, it triggers the cleanup of the namespace if there are no other processes, as PID 1 has the special role of adopting orphan processes. The Linux kernel will then disable PID allocation in that namespace.
2. **Consequence**:
   * The exit of PID 1 in a new namespace leads to the cleaning of the PIDNS\_HASH\_ADDING flag. This results in the alloc\_pid function failing to allocate a new PID when creating a new process, producing the "Cannot allocate memory" error.
3. **Solution**:
   * The issue can be resolved by using the -f option with unshare. This option makes unshare fork a new process after creating the new PID namespace.
   * Executing %unshare -fp /bin/bash% ensures that the unshare command itself becomes PID 1 in the new namespace. /bin/bash and its child processes are then safely contained within this new namespace, preventing the premature exit of PID 1 and allowing normal PID allocation.

By ensuring that unshare runs with the -f flag, the new PID namespace is correctly maintained, allowing /bin/bash and its sub-processes to operate without encountering the memory allocation error.

#### Docker

Copy

docker run -ti --name ubuntu1 -v /usr:/ubuntu1 ubuntu bash

# Run ifconfig or ip -a

### Check which namespace is your process in

Copy

ls -l /proc/self/ns/net

lrwxrwxrwx 1 root root 0 Apr 4 20:30 /proc/self/ns/net -> 'net:[4026531840]'

### Find all Network namespaces

Copy

sudo find /proc -maxdepth 3 -type l -name net -exec readlink {} \; 2>/dev/null | sort -u | grep "net:"

# Find the processes with an specific namespace

sudo find /proc -maxdepth 3 -type l -name net -exec ls -l {} \; 2>/dev/null | grep <ns-number>

### Enter inside a Network namespace

Copy

nsenter -n TARGET\_PID --pid /bin/bash

Also, you can only **enter in another process namespace if you are root**. And you **cannot** **enter** in other namespace **without a descriptor** pointing to it (like /proc/self/ns/net).

## References

* <https://stackoverflow.com/questions/44666700/unshare-pid-bin-bash-fork-cannot-allocate-memory>

**IPC Namespace**

## Basic Information

An IPC (Inter-Process Communication) namespace is a Linux kernel feature that provides **isolation** of System V IPC objects, such as message queues, shared memory segments, and semaphores. This isolation ensures that processes in **different IPC namespaces cannot directly access or modify each other's IPC objects**, providing an additional layer of security and privacy between process groups.

### How it works:

1. When a new IPC namespace is created, it starts with a **completely isolated set of System V IPC objects**. This means that processes running in the new IPC namespace cannot access or interfere with the IPC objects in other namespaces or the host system by default.
2. IPC objects created within a namespace are visible and **accessible only to processes within that namespace**. Each IPC object is identified by a unique key within its namespace. Although the key may be identical in different namespaces, the objects themselves are isolated and cannot be accessed across namespaces.
3. Processes can move between namespaces using the setns() system call or create new namespaces using the unshare() or clone() system calls with the CLONE\_NEWIPC flag. When a process moves to a new namespace or creates one, it will start using the IPC objects associated with that namespace.

## Lab:

### Create different Namespaces

#### CLI

Copy

sudo unshare -i [--mount-proc] /bin/bash

By mounting a new instance of the /proc filesystem if you use the param --mount-proc, you ensure that the new mount namespace has an **accurate and isolated view of the process information specific to that namespace**.

Error: bash: fork: Cannot allocate memory

When unshare is executed without the -f option, an error is encountered due to the way Linux handles new PID (Process ID) namespaces. The key details and the solution are outlined below:

1. **Problem Explanation**:
   1. The Linux kernel allows a process to create new namespaces using the unshare system call. However, the process that initiates the creation of a new PID namespace (referred to as the "unshare" process) does not enter the new namespace; only its child processes do.
   2. Running %unshare -p /bin/bash% starts /bin/bash in the same process as unshare. Consequently, /bin/bash and its child processes are in the original PID namespace.
   3. The first child process of /bin/bash in the new namespace becomes PID 1. When this process exits, it triggers the cleanup of the namespace if there are no other processes, as PID 1 has the special role of adopting orphan processes. The Linux kernel will then disable PID allocation in that namespace.
2. **Consequence**:
   1. The exit of PID 1 in a new namespace leads to the cleaning of the PIDNS\_HASH\_ADDING flag. This results in the alloc\_pid function failing to allocate a new PID when creating a new process, producing the "Cannot allocate memory" error.
3. **Solution**:
   1. The issue can be resolved by using the -f option with unshare. This option makes unshare fork a new process after creating the new PID namespace.
   2. Executing %unshare -fp /bin/bash% ensures that the unshare command itself becomes PID 1 in the new namespace. /bin/bash and its child processes are then safely contained within this new namespace, preventing the premature exit of PID 1 and allowing normal PID allocation.

By ensuring that unshare runs with the -f flag, the new PID namespace is correctly maintained, allowing /bin/bash and its sub-processes to operate without encountering the memory allocation error.

#### Docker

Copy

docker run -ti --name ubuntu1 -v /usr:/ubuntu1 ubuntu bash

### Check which namespace is your process in

Copy

ls -l /proc/self/ns/ipc

lrwxrwxrwx 1 root root 0 Apr 4 20:37 /proc/self/ns/ipc -> 'ipc:[4026531839]'

### Find all IPC namespaces

Copy

sudo find /proc -maxdepth 3 -type l -name ipc -exec readlink {} \; 2>/dev/null | sort -u

# Find the processes with an specific namespace

sudo find /proc -maxdepth 3 -type l -name ipc -exec ls -l {} \; 2>/dev/null | grep <ns-number>

### Enter inside an IPC namespace

Copy

nsenter -i TARGET\_PID --pid /bin/bash

Also, you can only **enter in another process namespace if you are root**. And you **cannot** **enter** in other namespace **without a descriptor** pointing to it (like /proc/self/ns/net).

### Create IPC object

Copy

# Container

sudo unshare -i /bin/bash

ipcmk -M 100

Shared memory id: 0

ipcs -m

------ Shared Memory Segments --------

key shmid owner perms bytes nattch status

0x2fba9021 0 root 644 100 0

# From the host

ipcs -m # Nothing is seen

## References

* <https://stackoverflow.com/questions/44666700/unshare-pid-bin-bash-fork-cannot-allocate-memory>

**UTS namespace**

## Basic Information

A UTS (UNIX Time-Sharing System) namespace is a Linux kernel feature that provides i**solation of two system identifiers**: the **hostname** and the **NIS** (Network Information Service) domain name. This isolation allows each UTS namespace to have its **own independent hostname and NIS domain name**, which is particularly useful in containerization scenarios where each container should appear as a separate system with its own hostname.

### How it works:

1. When a new UTS namespace is created, it starts with a **copy of the hostname and NIS domain name from its parent namespace**. This means that, at creation, the new namespace s**hares the same identifiers as its parent**. However, any subsequent changes to the hostname or NIS domain name within the namespace will not affect other namespaces.
2. Processes within a UTS namespace **can change the hostname and NIS domain name** using the sethostname() and setdomainname() system calls, respectively. These changes are local to the namespace and do not affect other namespaces or the host system.
3. Processes can move between namespaces using the setns() system call or create new namespaces using the unshare() or clone() system calls with the CLONE\_NEWUTS flag. When a process moves to a new namespace or creates one, it will start using the hostname and NIS domain name associated with that namespace.

## Lab:

### Create different Namespaces

#### CLI

Copy

sudo unshare -u [--mount-proc] /bin/bash

By mounting a new instance of the /proc filesystem if you use the param --mount-proc, you ensure that the new mount namespace has an **accurate and isolated view of the process information specific to that namespace**.

Error: bash: fork: Cannot allocate memory

When unshare is executed without the -f option, an error is encountered due to the way Linux handles new PID (Process ID) namespaces. The key details and the solution are outlined below:

1. **Problem Explanation**:
   1. The Linux kernel allows a process to create new namespaces using the unshare system call. However, the process that initiates the creation of a new PID namespace (referred to as the "unshare" process) does not enter the new namespace; only its child processes do.
   2. Running %unshare -p /bin/bash% starts /bin/bash in the same process as unshare. Consequently, /bin/bash and its child processes are in the original PID namespace.
   3. The first child process of /bin/bash in the new namespace becomes PID 1. When this process exits, it triggers the cleanup of the namespace if there are no other processes, as PID 1 has the special role of adopting orphan processes. The Linux kernel will then disable PID allocation in that namespace.
2. **Consequence**:
   1. The exit of PID 1 in a new namespace leads to the cleaning of the PIDNS\_HASH\_ADDING flag. This results in the alloc\_pid function failing to allocate a new PID when creating a new process, producing the "Cannot allocate memory" error.
3. **Solution**:
   1. The issue can be resolved by using the -f option with unshare. This option makes unshare fork a new process after creating the new PID namespace.
   2. Executing %unshare -fp /bin/bash% ensures that the unshare command itself becomes PID 1 in the new namespace. /bin/bash and its child processes are then safely contained within this new namespace, preventing the premature exit of PID 1 and allowing normal PID allocation.

By ensuring that unshare runs with the -f flag, the new PID namespace is correctly maintained, allowing /bin/bash and its sub-processes to operate without encountering the memory allocation error.

#### Docker

Copy

docker run -ti --name ubuntu1 -v /usr:/ubuntu1 ubuntu bash

### Check which namespace is your process in

Copy

ls -l /proc/self/ns/uts

lrwxrwxrwx 1 root root 0 Apr 4 20:49 /proc/self/ns/uts -> 'uts:[4026531838]'

### Find all UTS namespaces

Copy

sudo find /proc -maxdepth 3 -type l -name uts -exec readlink {} \; 2>/dev/null | sort -u

# Find the processes with an specific namespace

sudo find /proc -maxdepth 3 -type l -name uts -exec ls -l {} \; 2>/dev/null | grep <ns-number>

### Enter inside an UTS namespace

Copy

nsenter -u TARGET\_PID --pid /bin/bash

Also, you can only **enter in another process namespace if you are root**. And you **cannot** **enter** in other namespace **without a descriptor** pointing to it (like /proc/self/ns/uts).

### Change hostname

Copy

unshare -u /bin/bash

hostname newhostname # Hostname won't be changed inside the host UTS ns

## References

* <https://stackoverflow.com/questions/44666700/unshare-pid-bin-bash-fork-cannot-allocate-memory>

**Time Namespace**

## Basic Information

The time namespace in Linux allows for per-namespace offsets to the system monotonic and boot-time clocks. It is commonly used in Linux containers to change the date/time within a container and adjust clocks after restoring from a checkpoint or snapshot.

## Lab:

### Create different Namespaces

#### CLI

Copy

sudo unshare -T [--mount-proc] /bin/bash

By mounting a new instance of the /proc filesystem if you use the param --mount-proc, you ensure that the new mount namespace has an **accurate and isolated view of the process information specific to that namespace**.

Error: bash: fork: Cannot allocate memory

When unshare is executed without the -f option, an error is encountered due to the way Linux handles new PID (Process ID) namespaces. The key details and the solution are outlined below:

1. **Problem Explanation**:
   1. The Linux kernel allows a process to create new namespaces using the unshare system call. However, the process that initiates the creation of a new PID namespace (referred to as the "unshare" process) does not enter the new namespace; only its child processes do.
   2. Running %unshare -p /bin/bash% starts /bin/bash in the same process as unshare. Consequently, /bin/bash and its child processes are in the original PID namespace.
   3. The first child process of /bin/bash in the new namespace becomes PID 1. When this process exits, it triggers the cleanup of the namespace if there are no other processes, as PID 1 has the special role of adopting orphan processes. The Linux kernel will then disable PID allocation in that namespace.
2. **Consequence**:
   1. The exit of PID 1 in a new namespace leads to the cleaning of the PIDNS\_HASH\_ADDING flag. This results in the alloc\_pid function failing to allocate a new PID when creating a new process, producing the "Cannot allocate memory" error.
3. **Solution**:
   1. The issue can be resolved by using the -f option with unshare. This option makes unshare fork a new process after creating the new PID namespace.
   2. Executing %unshare -fp /bin/bash% ensures that the unshare command itself becomes PID 1 in the new namespace. /bin/bash and its child processes are then safely contained within this new namespace, preventing the premature exit of PID 1 and allowing normal PID allocation.

By ensuring that unshare runs with the -f flag, the new PID namespace is correctly maintained, allowing /bin/bash and its sub-processes to operate without encountering the memory allocation error.

#### Docker

Copy

docker run -ti --name ubuntu1 -v /usr:/ubuntu1 ubuntu bash

### Check which namespace is your process in

Copy

ls -l /proc/self/ns/time

lrwxrwxrwx 1 root root 0 Apr 4 21:16 /proc/self/ns/time -> 'time:[4026531834]'

### Find all Time namespaces

Copy

sudo find /proc -maxdepth 3 -type l -name time -exec readlink {} \; 2>/dev/null | sort -u

# Find the processes with an specific namespace

sudo find /proc -maxdepth 3 -type l -name time -exec ls -l {} \; 2>/dev/null | grep <ns-number>

### Enter inside a Time namespace

Copy

nsenter -T TARGET\_PID --pid /bin/bash

Also, you can only **enter in another process namespace if you are root**. And you **cannot** **enter** in other namespace **without a descriptor** pointing to it (like /proc/self/ns/net).

## References

* <https://stackoverflow.com/questions/44666700/unshare-pid-bin-bash-fork-cannot-allocate-memory>
* <https://www.phoronix.com/news/Linux-Time-Namespace-Coming>

**User namespace**

## Basic Information

A user namespace is a Linux kernel feature that **provides isolation of user and group ID mappings**, allowing each user namespace to have its **own set of user and group IDs**. This isolation enables processes running in different user namespaces to **have different privileges and ownership**, even if they share the same user and group IDs numerically.

User namespaces are particularly useful in containerization, where each container should have its own independent set of user and group IDs, allowing for better security and isolation between containers and the host system.

### How it works:

1. When a new user namespace is created, it **starts with an empty set of user and group ID mappings**. This means that any process running in the new user namespace will **initially have no privileges outside of the namespace**.
2. ID mappings can be established between the user and group IDs in the new namespace and those in the parent (or host) namespace. This **allows processes in the new namespace to have privileges and ownership corresponding to user and group IDs in the parent namespace**. However, the ID mappings can be restricted to specific ranges and subsets of IDs, allowing for fine-grained control over the privileges granted to processes in the new namespace.
3. Within a user namespace, **processes can have full root privileges (UID 0) for operations inside the namespace**, while still having limited privileges outside the namespace. This allows **containers to run with root-like capabilities within their own namespace without having full root privileges on the host system**.
4. Processes can move between namespaces using the setns() system call or create new namespaces using the unshare() or clone() system calls with the CLONE\_NEWUSER flag. When a process moves to a new namespace or creates one, it will start using the user and group ID mappings associated with that namespace.

## Lab:

### Create different Namespaces

#### CLI

Copy

sudo unshare -U [--mount-proc] /bin/bash

By mounting a new instance of the /proc filesystem if you use the param --mount-proc, you ensure that the new mount namespace has an **accurate and isolated view of the process information specific to that namespace**.

Error: bash: fork: Cannot allocate memory

When unshare is executed without the -f option, an error is encountered due to the way Linux handles new PID (Process ID) namespaces. The key details and the solution are outlined below:

1. **Problem Explanation**:
   1. The Linux kernel allows a process to create new namespaces using the unshare system call. However, the process that initiates the creation of a new PID namespace (referred to as the "unshare" process) does not enter the new namespace; only its child processes do.
   2. Running %unshare -p /bin/bash% starts /bin/bash in the same process as unshare. Consequently, /bin/bash and its child processes are in the original PID namespace.
   3. The first child process of /bin/bash in the new namespace becomes PID 1. When this process exits, it triggers the cleanup of the namespace if there are no other processes, as PID 1 has the special role of adopting orphan processes. The Linux kernel will then disable PID allocation in that namespace.
2. **Consequence**:
   1. The exit of PID 1 in a new namespace leads to the cleaning of the PIDNS\_HASH\_ADDING flag. This results in the alloc\_pid function failing to allocate a new PID when creating a new process, producing the "Cannot allocate memory" error.
3. **Solution**:
   1. The issue can be resolved by using the -f option with unshare. This option makes unshare fork a new process after creating the new PID namespace.
   2. Executing %unshare -fp /bin/bash% ensures that the unshare command itself becomes PID 1 in the new namespace. /bin/bash and its child processes are then safely contained within this new namespace, preventing the premature exit of PID 1 and allowing normal PID allocation.

By ensuring that unshare runs with the -f flag, the new PID namespace is correctly maintained, allowing /bin/bash and its sub-processes to operate without encountering the memory allocation error.

#### Docker

Copy

docker run -ti --name ubuntu1 -v /usr:/ubuntu1 ubuntu bash

To use user namespace, Docker daemon needs to be started with **--userns-remap=default**(In ubuntu 14.04, this can be done by modifying /etc/default/docker and then executing sudo service docker restart)

### Check which namespace is your process in

Copy

ls -l /proc/self/ns/user

lrwxrwxrwx 1 root root 0 Apr 4 20:57 /proc/self/ns/user -> 'user:[4026531837]'

It's possible to check the user map from the docker container with:

Copy

cat /proc/self/uid\_map

0 0 4294967295 --> Root is root in host

0 231072 65536 --> Root is 231072 userid in host

Or from the host with:

Copy

cat /proc/<pid>/uid\_map

### Find all User namespaces

Copy

sudo find /proc -maxdepth 3 -type l -name user -exec readlink {} \; 2>/dev/null | sort -u

# Find the processes with an specific namespace

sudo find /proc -maxdepth 3 -type l -name user -exec ls -l {} \; 2>/dev/null | grep <ns-number>

### Enter inside a User namespace

Copy

nsenter -U TARGET\_PID --pid /bin/bash

Also, you can only **enter in another process namespace if you are root**. And you **cannot** **enter** in other namespace **without a descriptor** pointing to it (like /proc/self/ns/user).

### Create new User namespace (with mappings)

Copy

unshare -U [--map-user=<uid>|<name>] [--map-group=<gid>|<name>] [--map-root-user] [--map-current-user]

Copy

# Container

sudo unshare -U /bin/bash

nobody@ip-172-31-28-169:/home/ubuntu$ #Check how the user is nobody

# From the host

ps -ef | grep bash # The user inside the host is still root, not nobody

root 27756 27755 0 21:11 pts/10 00:00:00 /bin/bash

### Recovering Capabilities

In the case of user namespaces, **when a new user namespace is created, the process that enters the namespace is granted a full set of capabilities within that namespace**. These capabilities allow the process to perform privileged operations such as **mounting** **filesystems**, creating devices, or changing ownership of files, but **only within the context of its user namespace**.

For example, when you have the CAP\_SYS\_ADMIN capability within a user namespace, you can perform operations that typically require this capability, like mounting filesystems, but only within the context of your user namespace. Any operations you perform with this capability won't affect the host system or other namespaces.

Therefore, even if getting a new process inside a new User namespace **will give you all the capabilities back** (CapEff: 000001ffffffffff), you actually can **only use the ones related to the namespace** (mount for example) but not every one. So, this on its own is not enough to escape from a Docker container.

Copy

# There are the syscalls that are filtered after changing User namespace with:

unshare -UmCpf bash

Probando: 0x067 . . . Error

Probando: 0x070 . . . Error

Probando: 0x074 . . . Error

Probando: 0x09b . . . Error

Probando: 0x0a3 . . . Error

Probando: 0x0a4 . . . Error

Probando: 0x0a7 . . . Error

Probando: 0x0a8 . . . Error

Probando: 0x0aa . . . Error

Probando: 0x0ab . . . Error

Probando: 0x0af . . . Error

Probando: 0x0b0 . . . Error

Probando: 0x0f6 . . . Error

Probando: 0x12c . . . Error

Probando: 0x130 . . . Error

Probando: 0x139 . . . Error

Probando: 0x140 . . . Error

Probando: 0x141 . . . Error

Probando: 0x143 . . . Error

## References

* <https://stackoverflow.com/questions/44666700/unshare-pid-bin-bash-fork-cannot-allocate-memory>

**CGroup Namespace**

## Basic Information

A cgroup namespace is a Linux kernel feature that provides **isolation of cgroup hierarchies for processes running within a namespace**. Cgroups, short for **control groups**, are a kernel feature that allows organizing processes into hierarchical groups to manage and enforce **limits on system resources** like CPU, memory, and I/O.

While cgroup namespaces are not a separate namespace type like the others we discussed earlier (PID, mount, network, etc.), they are related to the concept of namespace isolation. **Cgroup namespaces virtualize the view of the cgroup hierarchy**, so that processes running within a cgroup namespace have a different view of the hierarchy compared to processes running in the host or other namespaces.

### How it works:

1. When a new cgroup namespace is created, **it starts with a view of the cgroup hierarchy based on the cgroup of the creating process**. This means that processes running in the new cgroup namespace will only see a subset of the entire cgroup hierarchy, limited to the cgroup subtree rooted at the creating process's cgroup.
2. Processes within a cgroup namespace will **see their own cgroup as the root of the hierarchy**. This means that, from the perspective of processes inside the namespace, their own cgroup appears as the root, and they cannot see or access cgroups outside of their own subtree.
3. Cgroup namespaces do not directly provide isolation of resources; **they only provide isolation of the cgroup hierarchy view**. **Resource control and isolation are still enforced by the cgroup** subsystems (e.g., cpu, memory, etc.) themselves.

For more information about CGroups check:

CGroups

## Basic Information

**Linux Control Groups**, or **cgroups**, are a feature of the Linux kernel that allows the allocation, limitation, and prioritization of system resources like CPU, memory, and disk I/O among process groups. They offer a mechanism for **managing and isolating the resource usage** of process collections, beneficial for purposes such as resource limitation, workload isolation, and resource prioritization among different process groups.

There are **two versions of cgroups**: version 1 and version 2. Both can be used concurrently on a system. The primary distinction is that **cgroups version 2** introduces a **hierarchical, tree-like structure**, enabling more nuanced and detailed resource distribution among process groups. Additionally, version 2 brings various enhancements, including:

In addition to the new hierarchical organization, cgroups version 2 also introduced **several other changes and improvements**, such as support for **new resource controllers**, better support for legacy applications, and improved performance.

Overall, cgroups **version 2 offers more features and better performance** than version 1, but the latter may still be used in certain scenarios where compatibility with older systems is a concern.

You can list the v1 and v2 cgroups for any process by looking at its cgroup file in /proc/<pid>. You can start by looking at your shell’s cgroups with this command:

Copy

$ cat /proc/self/cgroup

12:rdma:/

11:net\_cls,net\_prio:/

10:perf\_event:/

9:cpuset:/

8:cpu,cpuacct:/user.slice

7:blkio:/user.slice

6:memory:/user.slice 5:pids:/user.slice/user-1000.slice/session-2.scope 4:devices:/user.slice

3:freezer:/

2:hugetlb:/testcgroup

1:name=systemd:/user.slice/user-1000.slice/session-2.scope

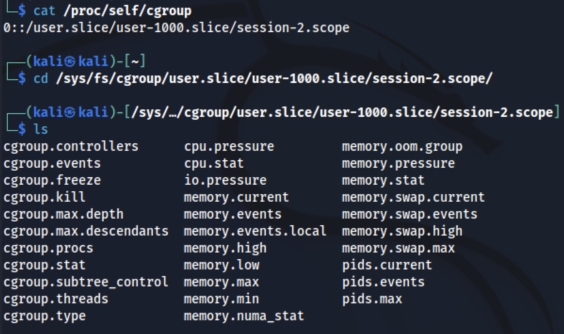
0::/user.slice/user-1000.slice/session-2.scope

The output structure is as follows:

* **Numbers 2–12**: cgroups v1, with each line representing a different cgroup. Controllers for these are specified adjacent to the number.
* **Number 1**: Also cgroups v1, but solely for management purposes (set by, e.g., systemd), and lacks a controller.
* **Number 0**: Represents cgroups v2. No controllers are listed, and this line is exclusive on systems only running cgroups v2.
* The **names are hierarchical**, resembling file paths, indicating the structure and relationship between different cgroups.
* **Names like /user.slice or /system.slice** specify the categorization of cgroups, with user.slice typically for login sessions managed by systemd and system.slice for system services.

### Viewing cgroups

The filesystem is typically utilized for accessing **cgroups**, diverging from the Unix system call interface traditionally used for kernel interactions. To investigate a shell's cgroup configuration, one should examine the **/proc/self/cgroup** file, which reveals the shell's cgroup. Then, by navigating to the **/sys/fs/cgroup** (or **/sys/fs/cgroup/unified**) directory and locating a directory that shares the cgroup's name, one can observe various settings and resource usage information pertinent to the cgroup.



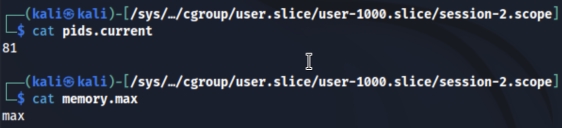
Cgroup Filesystem

The key interface files for cgroups are prefixed with **cgroup**. The **cgroup.procs** file, which can be viewed with standard commands like cat, lists the processes within the cgroup. Another file, **cgroup.threads**, includes thread information.



Cgroup Procs

Cgroups managing shells typically encompass two controllers that regulate memory usage and process count. To interact with a controller, files bearing the controller's prefix should be consulted. For instance, **pids.current** would be referenced to ascertain the count of threads in the cgroup.



Cgroup Memory

The indication of **max** in a value suggests the absence of a specific limit for the cgroup. However, due to the hierarchical nature of cgroups, limits might be imposed by a cgroup at a lower level in the directory hierarchy.

### Manipulating and Creating cgroups

Processes are assigned to cgroups by **writing their Process ID (PID) to the cgroup.procs file**. This requires root privileges. For instance, to add a process:

Copy

echo [pid] > cgroup.procs

Similarly, **modifying cgroup attributes, like setting a PID limit**, is done by writing the desired value to the relevant file. To set a maximum of 3,000 PIDs for a cgroup:

Copy

echo 3000 > pids.max

**Creating new cgroups** involves making a new subdirectory within the cgroup hierarchy, which prompts the kernel to automatically generate necessary interface files. Though cgroups without active processes can be removed with rmdir, be aware of certain constraints:

* **Processes can only be placed in leaf cgroups** (i.e., the most nested ones in a hierarchy).
* **A cgroup cannot possess a controller absent in its parent**.
* **Controllers for child cgroups must be explicitly declared** in the cgroup.subtree\_control file. For example, to enable CPU and PID controllers in a child cgroup:

Copy

echo "+cpu +pids" > cgroup.subtree\_control

The **root cgroup** is an exception to these rules, allowing direct process placement. This can be used to remove processes from systemd management.

**Monitoring CPU usage** within a cgroup is possible through the cpu.stat file, displaying total CPU time consumed, helpful for tracking usage across a service's subprocesses:



CPU usage statistics as shown in the cpu.stat file

## References

* **Book: How Linux Works, 3rd Edition: What Every Superuser Should Know By Brian Ward**

## Lab:

### Create different Namespaces

#### CLI

Copy

sudo unshare -C [--mount-proc] /bin/bash

By mounting a new instance of the /proc filesystem if you use the param --mount-proc, you ensure that the new mount namespace has an **accurate and isolated view of the process information specific to that namespace**.

Error: bash: fork: Cannot allocate memory

When unshare is executed without the -f option, an error is encountered due to the way Linux handles new PID (Process ID) namespaces. The key details and the solution are outlined below:

1. **Problem Explanation**:
   1. The Linux kernel allows a process to create new namespaces using the unshare system call. However, the process that initiates the creation of a new PID namespace (referred to as the "unshare" process) does not enter the new namespace; only its child processes do.
   2. Running %unshare -p /bin/bash% starts /bin/bash in the same process as unshare. Consequently, /bin/bash and its child processes are in the original PID namespace.
   3. The first child process of /bin/bash in the new namespace becomes PID 1. When this process exits, it triggers the cleanup of the namespace if there are no other processes, as PID 1 has the special role of adopting orphan processes. The Linux kernel will then disable PID allocation in that namespace.
2. **Consequence**:
   1. The exit of PID 1 in a new namespace leads to the cleaning of the PIDNS\_HASH\_ADDING flag. This results in the alloc\_pid function failing to allocate a new PID when creating a new process, producing the "Cannot allocate memory" error.
3. **Solution**:
   1. The issue can be resolved by using the -f option with unshare. This option makes unshare fork a new process after creating the new PID namespace.
   2. Executing %unshare -fp /bin/bash% ensures that the unshare command itself becomes PID 1 in the new namespace. /bin/bash and its child processes are then safely contained within this new namespace, preventing the premature exit of PID 1 and allowing normal PID allocation.

By ensuring that unshare runs with the -f flag, the new PID namespace is correctly maintained, allowing /bin/bash and its sub-processes to operate without encountering the memory allocation error.

#### Docker

Copy

docker run -ti --name ubuntu1 -v /usr:/ubuntu1 ubuntu bash

### Check which namespace is your process in

Copy

ls -l /proc/self/ns/cgroup

lrwxrwxrwx 1 root root 0 Apr 4 21:19 /proc/self/ns/cgroup -> 'cgroup:[4026531835]'

### Find all CGroup namespaces

Copy

sudo find /proc -maxdepth 3 -type l -name cgroup -exec readlink {} \; 2>/dev/null | sort -u

# Find the processes with an specific namespace

sudo find /proc -maxdepth 3 -type l -name cgroup -exec ls -l {} \; 2>/dev/null | grep <ns-number>

### Enter inside an CGroup namespace

Copy

nsenter -C TARGET\_PID --pid /bin/bash

Also, you can only **enter in another process namespace if you are root**. And you **cannot** **enter** in other namespace **without a descriptor** pointing to it (like /proc/self/ns/cgroup).

## References

* <https://stackoverflow.com/questions/44666700/unshare-pid-bin-bash-fork-cannot-allocate-memory>