Resource Virtualization with Containers

Tanu Malik

School of Computing, DePaul University

Visiting Faculty, CSE, IIT, Delhi

Namespace management

- Creating a new namespace
  - clone(function, stack, CLONE_NEW*, args)
    - creates a new process and a new namespace; the new process is attached to the new namespace.
- Joining an existing namespace
  - setns(fd, nstype)
    - calling process to join an existing namespace specified by the file descriptor and nstype.
- Redefining namespace
  - unshare(flags)
    - Disassociating parts of process execution contexts
- Discovering namespace relationships
  - ioctl(fd, request)
    - discovery of namespace relationships

Revision

- Clone with CLONE_FILES unset is same as fork. In this case fds are copied and duplicated
- Clone with CLONE_FILES set leads to sharing of fds. Sharing means no new fds are created.
- CLONE_FILES is not to be confused with CLONE_NEWNS which relates to mount points and root directory
- CLONE_NEWNS cannot be used with CLONE_FS.

Flags for namespaces

CLONE_NEWUTS	2.6.19	CAP_SYS_ADMIN
CLONE_NEWPID	2.6.24	CAP_SYS_ADMIN
CLONE_NEWUSER	3.8	No capability is required
CLONE_NEWNS	2.4.19	CAP_SYS_ADMIN
CLONE_NEWIPC	2.6.19	CAP_SYS_ADMIN
CLONE_NEWNET	2.6.29	CAP_SYS_ADMIN

Example 1

- Demo_uts_namespace.c
  - Show two different namespaces
  - Show namespace does not exist
- Force existence
  - # **touch ~/uts** # Create mount point
  - # **mount --bind /proc/<CHILD_PID>/ns/uts ~/uts**
- setns.c
  - **./setns ~/uts /bin/bash**
  - # hostname
  - # ls –l **/proc/$$/ns/**

Unsharing execution contexts: unshare()

int unshare(int flags);

- Functionality similar to clone() but operates on the caller instead of the callee
- It creates the new namespaces specified by the CLONE_NEW* bits in its flags argument and makes the caller a member of the namespaces.
- Main purpose of unshare() is to isolate namespace (and other) side effects without having to create a new process or thread (as is done by clone()).

Example of unshare

- `clone(..., CLONE_NEWXXX, ....);`

is roughly equivalent, in namespace terms, to the sequence:

```
if (fork() == 0)
    unshare(CLONE_NEWXXX);
```

- **Unshare is also available as a command**
    - unshare [options] program [arguments]
    - options are command-line flags that specify the namespaces to unshare before executing program with the specified arguments.

Example 2

- unshare.c
- # **./unshare -m /bin/bash**
  - # Start new shell in separate mount namespace
- # **mount –t tmpfs tmpfs /mnt**
- **# mount | grep mnt**
  - Show one of the mounts in namespace
  - Show absence of it in other namespace

PID

- PID is a system resource. It helps to identify a process uniquely even if there are two processes that share the same human-readable name.
- PIDs are tracked in a special file system called *procfs*.
- /proc is where most Unix-like systems store information regarding processes on a running system.

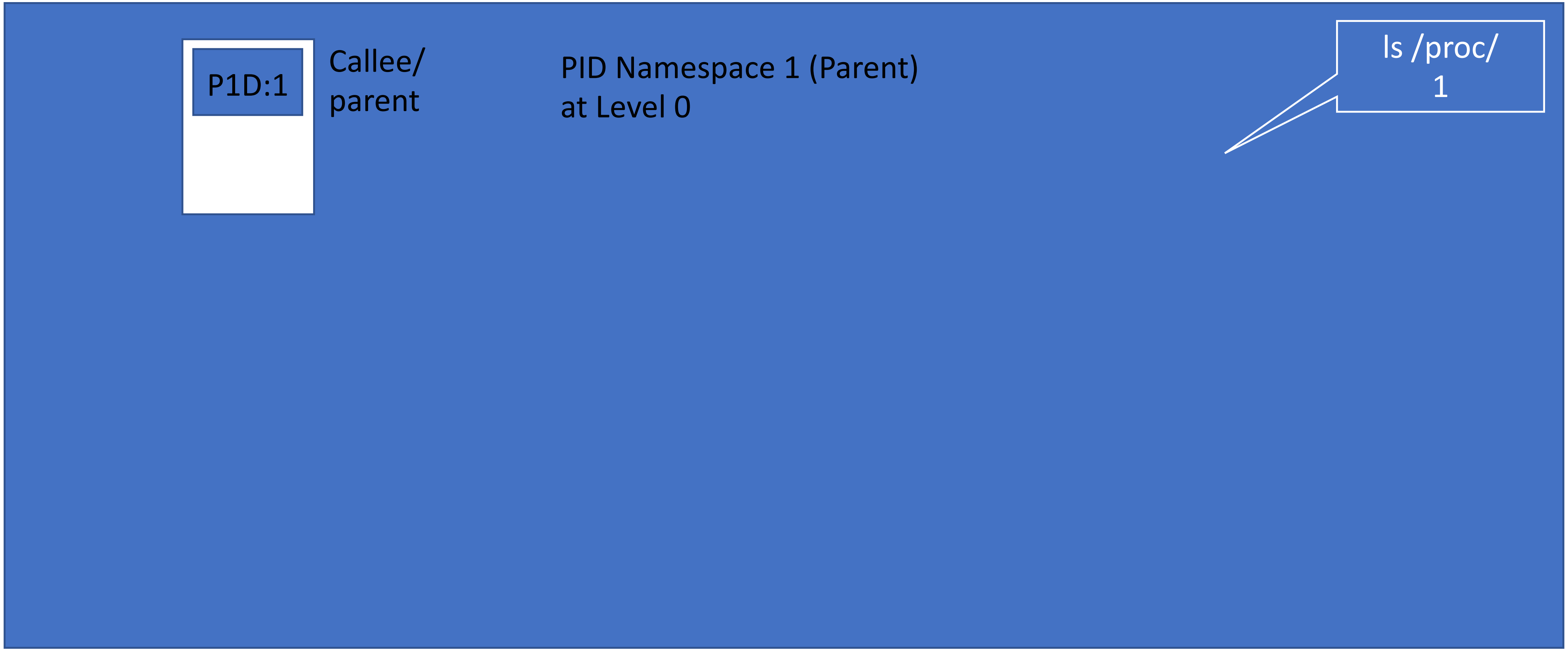
PID Namespace

- PID namespaces isolate the process ID, meaning that processes in different PID namespaces can have the same PID.
- Each PID namespace has its own numbering staring from 1
- PID namespaces can be nested

Creating PID Namespace

```
child_pid = clone(childFunc, child_stack +
STACK_SIZE, CLONE_NEWPID | SIGCHLD, argv[1]);
printf("PID returned by clone(): %ld\n", (long)
child_pid);
```

PID Namespace



P1D:1

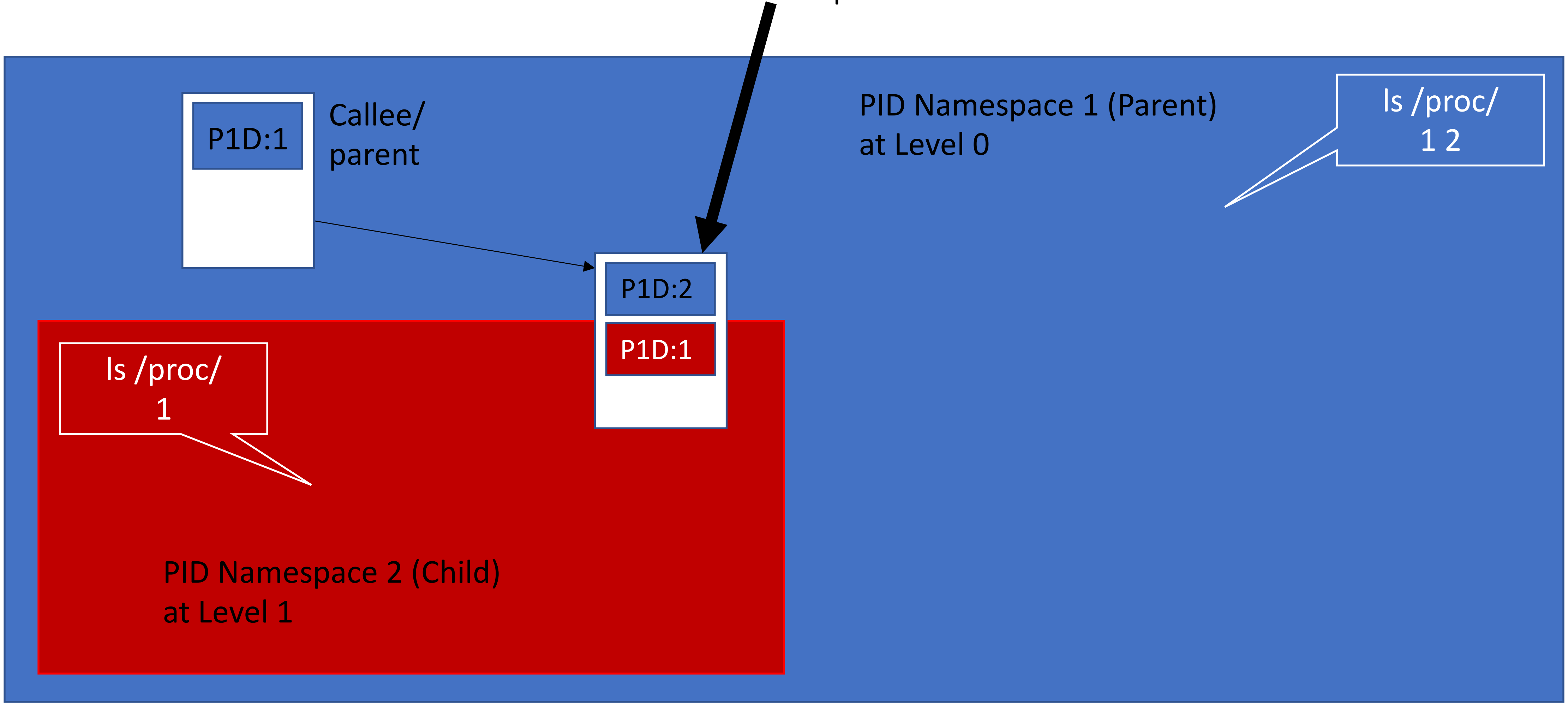
Callee/
parent

PID Namespace 1 (Parent)
at Level 0

ls /proc/
1

PID Namespace

Cloned process
in the new namespace
Has no knowledge of parent
In this namespace



P1D:1

Callee/
parent

PID Namespace 1 (Parent)
at Level 0

ls /proc/
1 2

P1D:2

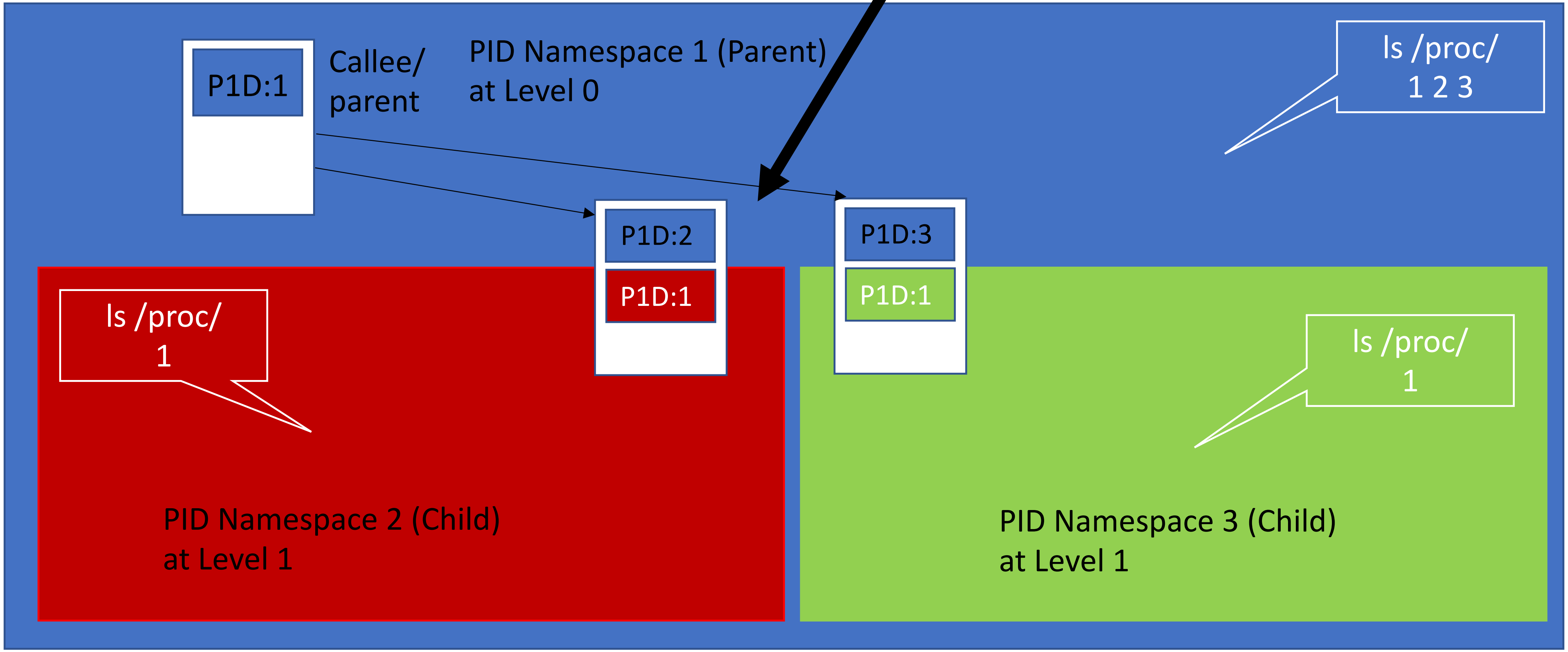
P1D:1

ls /proc/
1

PID Namespace 2 (Child)
at Level 1

PID Namespace

Cloned processes
in the new namespaces
have no knowledge of parent in their
respective namespaces and no knowledge
of each other



P1D:1

Callee/
parent

PID Namespace 1 (Parent)
at Level 0

ls /proc/
1 2 3

P1D:2

P1D:1

P1D:3

P1D:1

ls /proc/
1

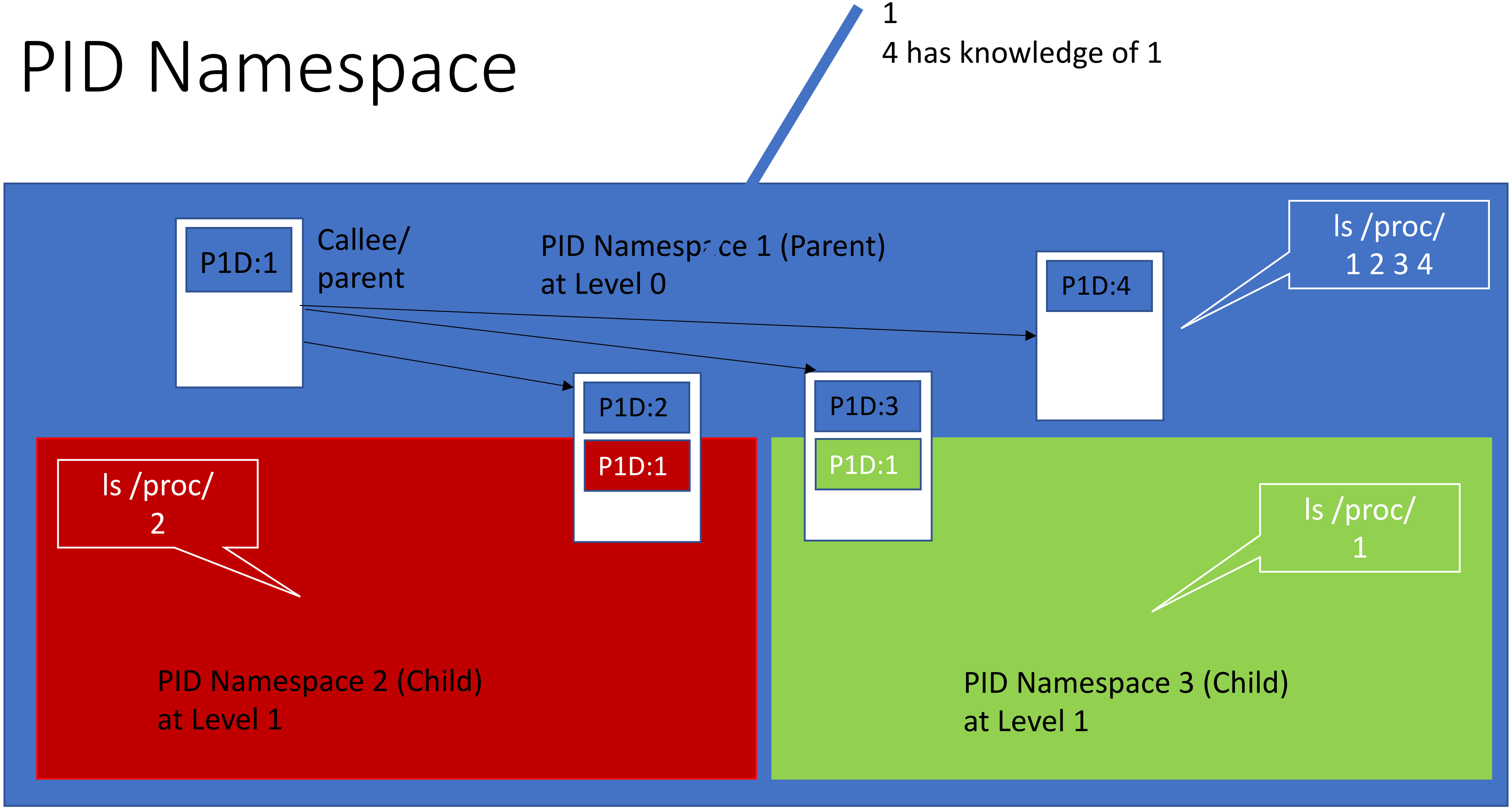
ls /proc/
1

PID Namespace 2 (Child)
at Level 1

PID Namespace 3 (Child)
at Level 1

PID Namespace

2 & 3 have no knowledge of
1
4 has knowledge of 1



P1D:1

Callee/
parent

PID Namespace 1 (Parent)
at Level 0

P1D:4

ls /proc/
1 2 3 4

P1D:2

P1D:1

P1D:3

P1D:1

ls /proc/
2

ls /proc/
1

PID Namespace 2 (Child)
at Level 1

PID Namespace 3 (Child)
at Level 1

Parent child relationship

- Multiple “nested” namespaces.
- Each namespace can have an entirely isolated set of processes.
- Processes belonging to one namespace cannot inspect or kill - in fact cannot even know of the existence of - processes in other sibling or parent namespaces.
    - getppid() returns 0 (null).

Each process multiple PIDs

```
struct upid {
int nr; // the PID value
struct pid_namespace *ns; // namespace where this PID is relevant
// ...
};

struct pid {
// ...
int level; // number of upids
struct upid numbers[0]; // array of upids
};
```

- A single process has multiple PIDs associated with it, one for each namespace.
- In the Linux source code, we can see that a struct named pid, which used to keep track of just a single PID, now tracks multiple PIDs through the use of a struct named upid.
- *getpid()* always reports the PID associated with the namespace in which the calling process of getpid() resides

Initializing a PID namespace

- The first process created inside a PID namespace gets a process ID of 1 within the namespace.
- This process has a similar role to the *init* process on traditional Linux systems.
- In particular, the init process can perform initializations required for the PID namespace as whole
    - starting other processes that should be a standard part of the namespace
    - Terminating other processes if init terminates
    - Reaps orphaned child processes when they terminate.
    - Restrictions apply on sending signals to the *init* process within the namespace i.e. signals can be sent from outside the namespace.

Initializing a PID namespace with shell

- **demo_pid_namespace [options] command [arguments]**
- **./demo_pid_namespace -p sh -c 'echo $$'**

More complex initialization

- Execute a simple shell facility that allows the user to manually execute any shell commands that might be needed to initialize the namespace
- **./demo_pid_namespace -p ./simple_init**
- **./demo_pid_namespace –p -m ./simple_init**
- **mount -t proc proc /proc**
- **ps a**
- Does it really behave like *init*?
- **./demo_pid_namespace -p ./simple_init –v**
- **./orphan**
- Shows the child is adopted by the PID namespace init process (PID 1), which reaps the child when it terminates.

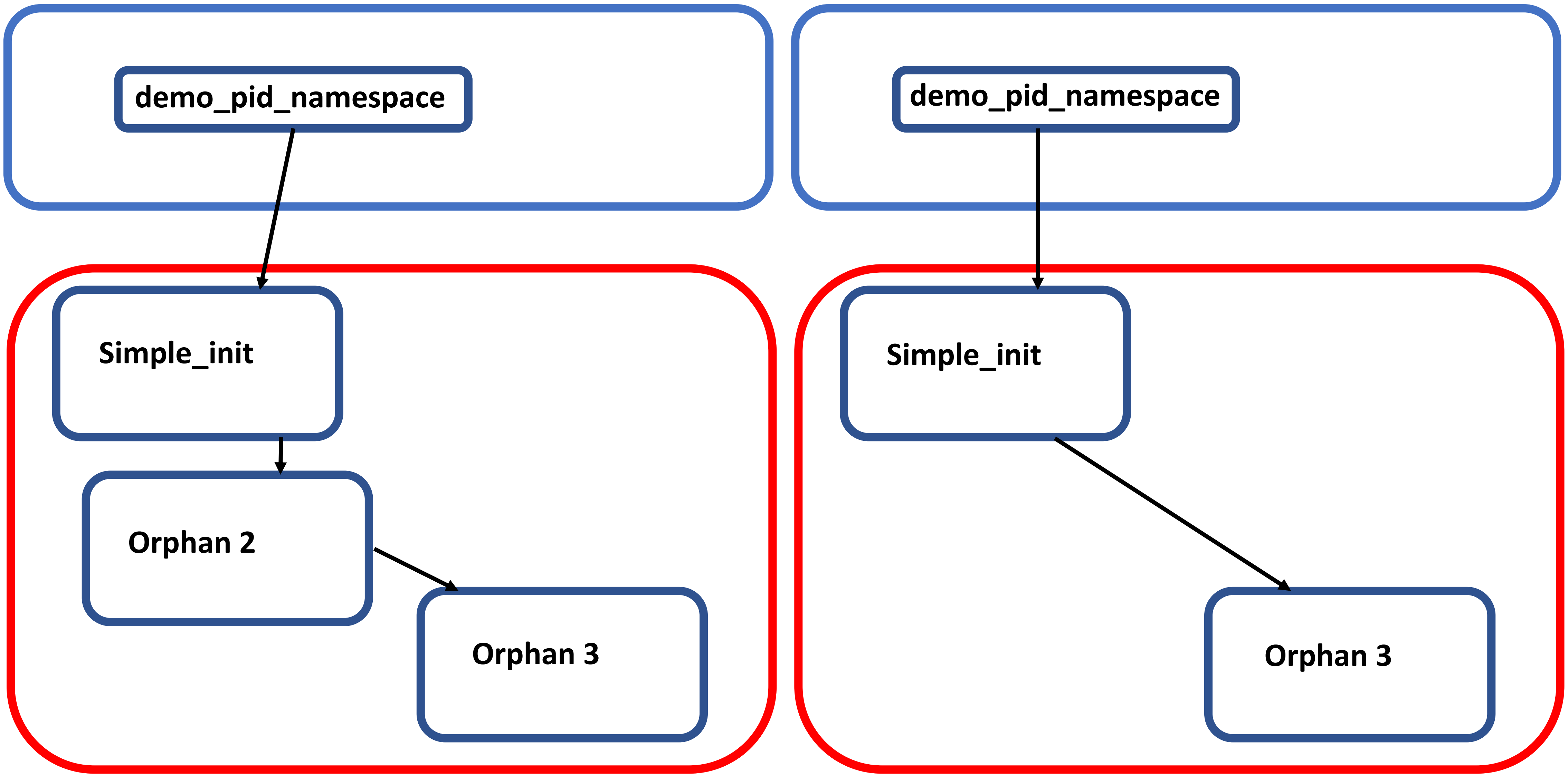


Diagram credit: LWN Namespace Series

Signals and *init*

- What signals can be delivered to traditional *init*? Why?

Signals and *init*

- What signals can be delivered to traditional *init*? Why?

- PID namespaces implement same behavior for the namespace-specific *init* process.
  - Parent namespace can still generate signals for the PID namespace *init* process in all of the usual circumstances
  - Who is the parent of all other processes then? The kernel which kills all the other processes.

PID namespace and setns() and unshare()

- unshare() and setns() typically put the caller and not the callee into the new namespace.
- However, **not true** if fd or flag is a PID namespace.
- Why?

Unshare() and setns() with PID namespace

- unshare() creates a new PID namespace but does *not* place the caller in the new namespace. Instead any children created by the caller will be placed in the new namespace; the first such child will become the *init* process for the namespace.

- setns() does *not* move the caller to the PID namespace; instead, children that are subsequently created by the caller will be placed in the namespace.

Example

- Show PID namespace fails
- Sudo ./unshare –p /bin/bash
- Start a PID namespace
- **./demo_pid_namespace -p ./simple_init**

- **fork orphan which will join the PID namespace of simple_init**
- ./setns_for_pid **-f -n /proc/<PID>/ns/pid ./orphan**

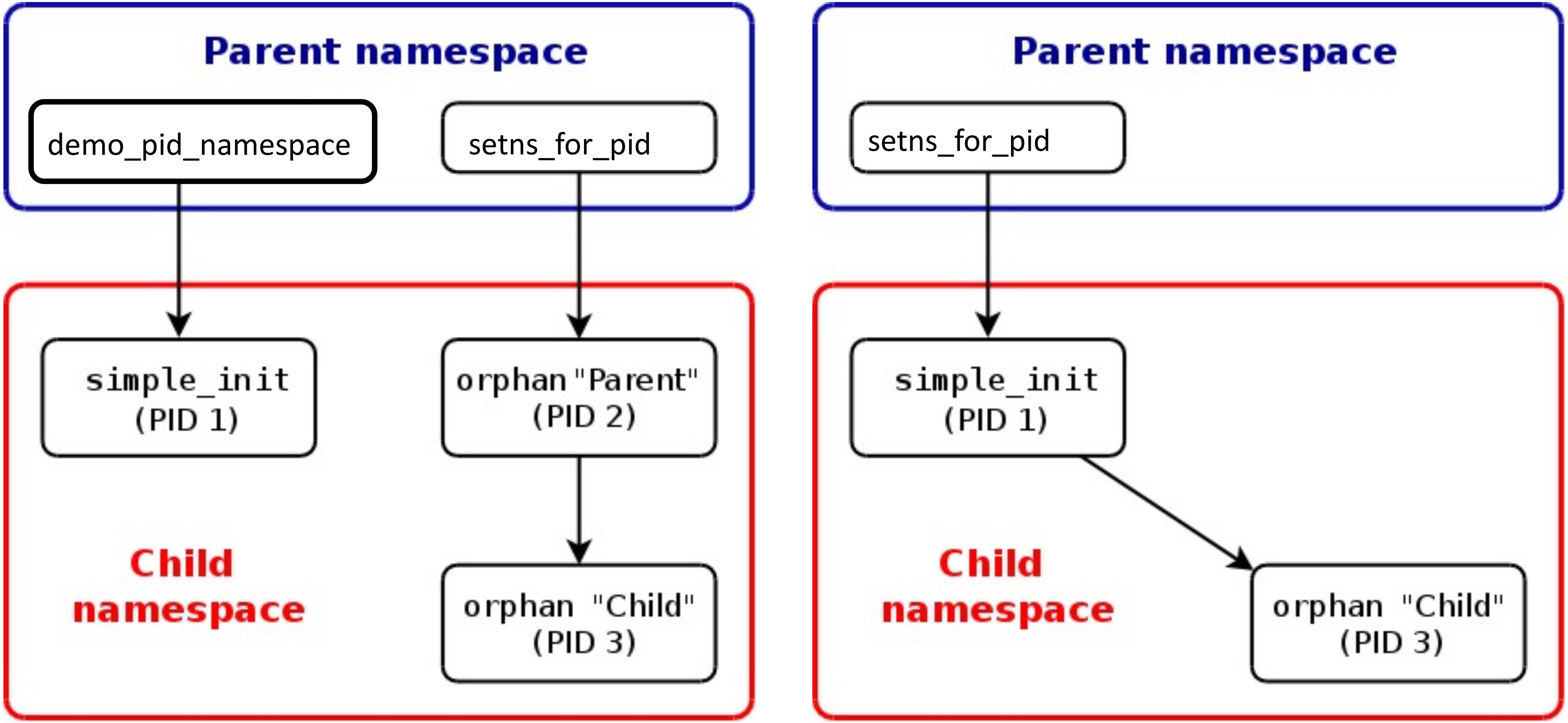
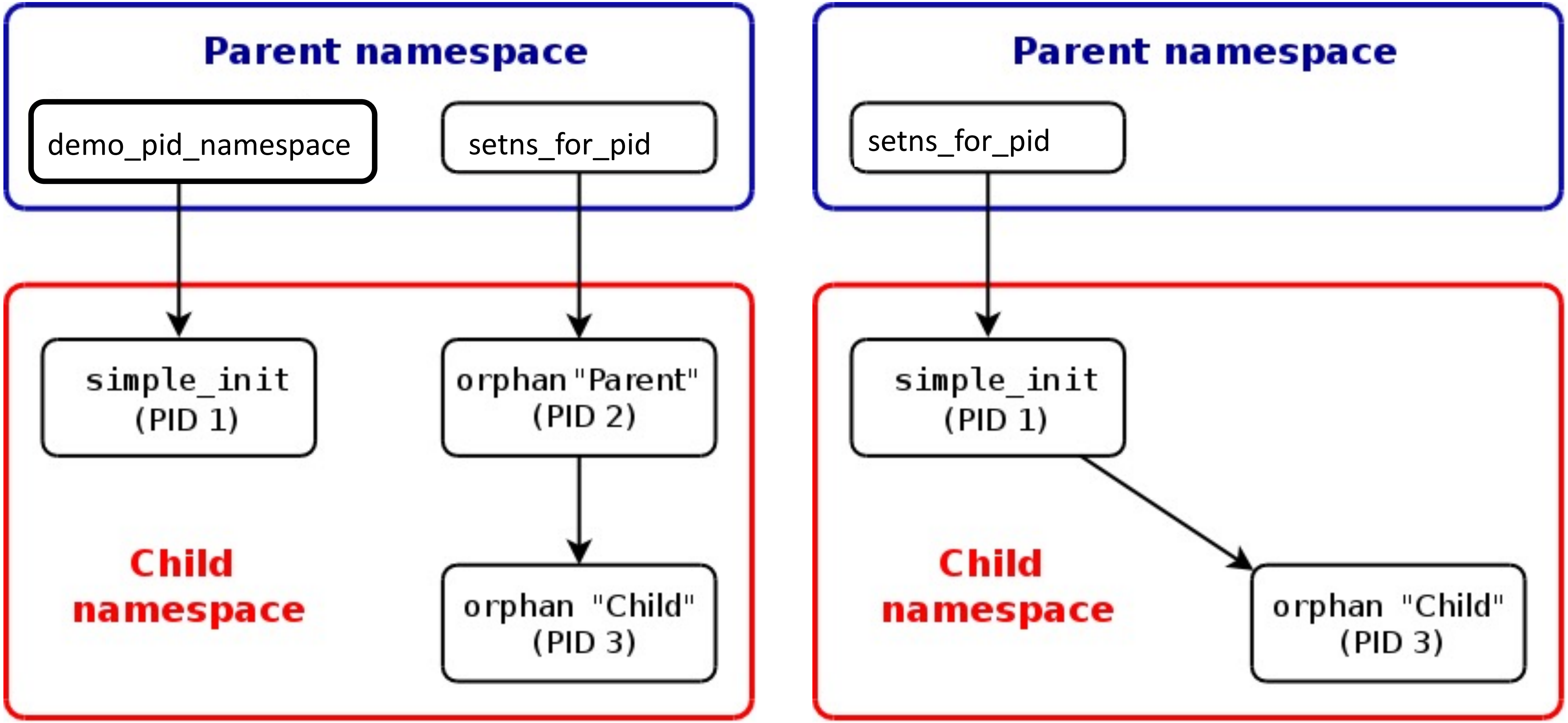


Diagram credit: LWN Namespace Series



Parent namespace

demo_pid_namespace

setns_for_pid

simple_init
(PID 1)

orphan "Parent"
(PID 2)

Child namespace

orphan "Child"
(PID 3)

Parent namespace

setns_for_pid

simple_init
(PID 1)

Child namespace

orphan "Child"
(PID 3)

Diagram credit: LWN Namespace Series

User Namespace

- Allow per-namespace mappings of user and group IDs.
  - A process's user and group IDs can be different inside and outside a user namespace.
- A process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of 0 inside the namespace.
  - This means that the process has full root privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.

Creating User Namespace

```
child_pid = clone(childFunc, child_stack +
STACK_SIZE, CLONE_NEWUSER | SIGCHLD, argv[1]);
```

- Unshare(`CLONE_NEWUSER`)

- No privilege is required to create a user namespace.

Example

- demo_userns.c: creates a child in a new user namespace.
  - Note install libcap-dev (sudo apt-get install libcap-dev –y)
- Child display its effective user and group IDs as well as its capabilities.
- The child has a full set of permitted and effective capabilities, even though the program was run from an unprivileged account.
  - the new process has a full set of capabilities in the new user namespace, it has no capabilities in the parent namespace.
- When a user namespace is created, the first process in the namespace is granted a full set of capabilities in the namespace.
  - This allows that process to perform any initializations that are necessary in the namespace before other processes are created in the namespace.

User and groupIds in User Namespace

- user and group IDs of the child process **may** be different.
- Default values chosen from `/proc/sys/kernel/overflowuid` and

`/proc/sys/kernel/overflowgid`

- Initially user and group IDs have no mapping

- Even if root employs `clone(CLONE_NEWUSER)`, the resulting child process will have no capabilities in the parent namespace

Who sets the mapping?

- Parent process sets the mapping of child process by writing two files available via /proc
    - /proc/*PID*/uid_map and /proc/*PID*/gid_map

UID and GID Mappings

- Records written to/read from /proc/*PID*/uid_map and /proc/*PID*/gid_map have this form:
  - ID-inside-ns ID-outside-ns length
- *ID-inside-ns* and *length* define range of IDs inside user NS that are to be mapped
- *ID-outside-ns* defines start of corresponding mapped range in “outside” user NS
- E.g., following says that IDs 0...9 inside user NS map to IDs 1000...1009 in outside user NS
  - 0 1000 10

User namespaces

- Allow per-namespace mappings of UIDs and GIDs
  - process's UIDs and GIDs inside NS may be different from IDs outside NS
- Interesting use case: process may have nonzero UID outside NS, and UID of 0 inside NS
  - Process has root privileges *for operations inside user NS*

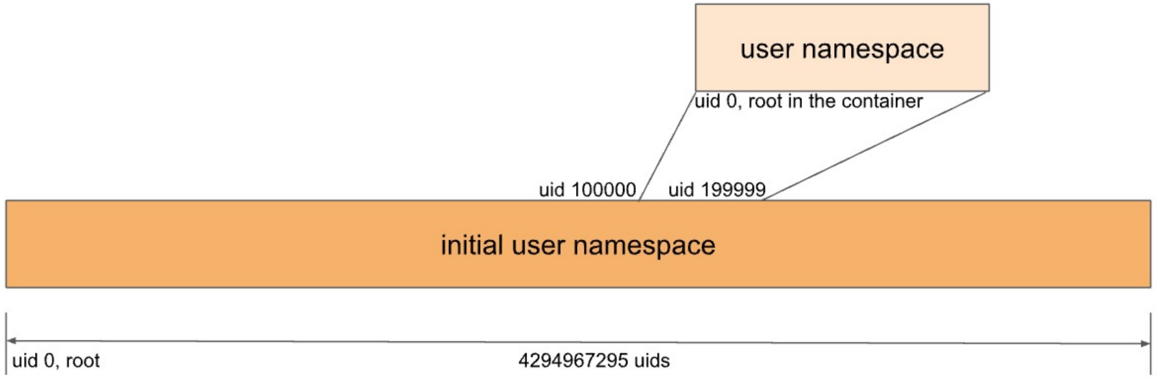


Diagram credit: Michael Kerrisk

User namespaces can be nested

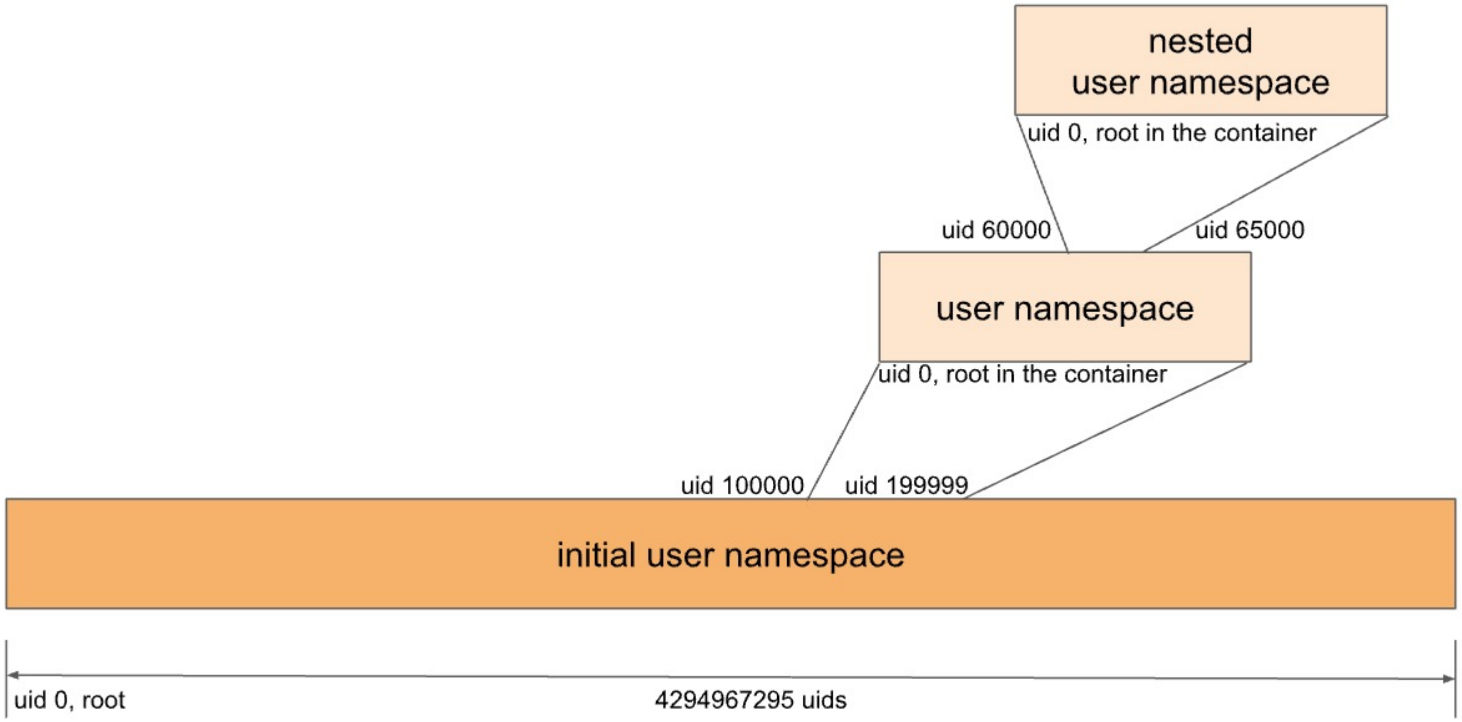


Diagram credit: Michael Kerrisk

Example

- **./demo_userns x**

- Determine PID of cloned child
- **ps -C demo_userns -o 'pid uid comm’**

- **echo '0 1000 1' > /proc/4713/uid_map**

- user ID 1000 in the parent user namespace (earlier mapped to 65534) has been mapped to user ID 0 in the user namespace created by demo_userns.

Who sets the mapping?

- Parent process sets the mapping of child process by writing two files available via /proc
    - /proc/*PID*/uid_map and /proc/*PID*/gid_map

- The child process must use the mapping before mounting
- No privilege is required to create a user namespace.
    - Program was run from unprivileged user account

"Root privileges inside a user NS”

- What does “root privileges in a user NS” mean?

- There are a number of NS types
- Each NS type governs some global resource(s); e.g.:
  - UTS: hostname, NIS domain name
  - Mount: set of mount point
  - Network: IP routing tables, port numbers, /proc/net, ...

- There is an ownership relationship between user NSs and non-user NSs such that **each non-user NS is “owned” by a particular user NS**
  - When creating a new nonuser NS, kernel marks that NS as owned by the **user NS of process creating the new NS**
- If a process operates on resources governed by nonuser NS:
  - Permission checks are done according to **process’s capabiliites in user NS that owns the nonuser NS that governs the resources**

User namespaces “govern” other namespace types

- X is created with Unshare –Ur –u <prog>
- X is in new user NS, with root mappings and has all capabilities
- X is in a new UTS NS, which is owned by new user NS
- X is in initial instance of all other NS types (e.g network NS)

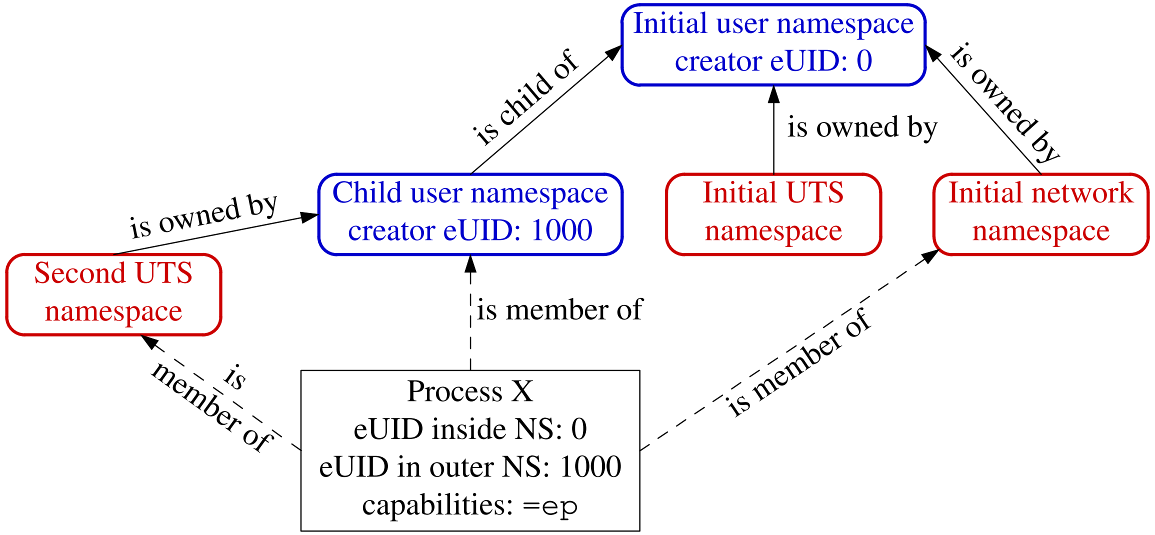


Diagram credit: Michael Kerrisk

Changing hostname

- Suppose X tries to change hostname (CAP_SYS_ADMIN)
- X is in second UTS NS
- Permissions checked according to X’s capabilities in user NS that owns that UTS NS => succeeds (X has capabilities in that user NS)

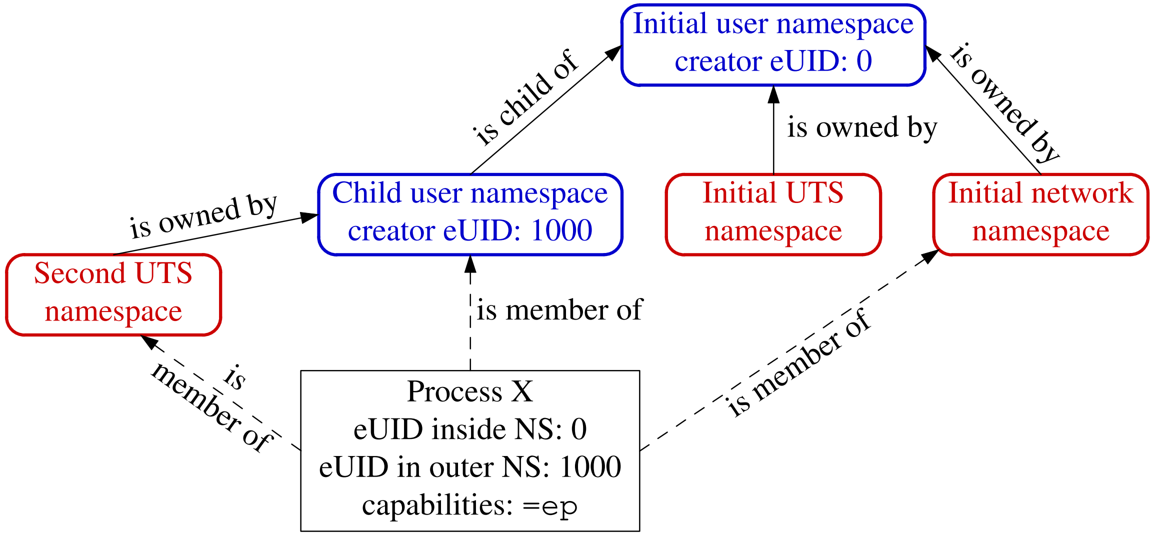


Diagram credit: Michael Kerrisk

Changing hostname

- Suppose X tries to bind to reserved socket port (CAP_NET_BIND_SERVICE)
- X is in initial NET NS
- Permissions checked according to X's

capabilities in user NS that owns

that network NS => fails

(X has no capabilities in initial user NS)

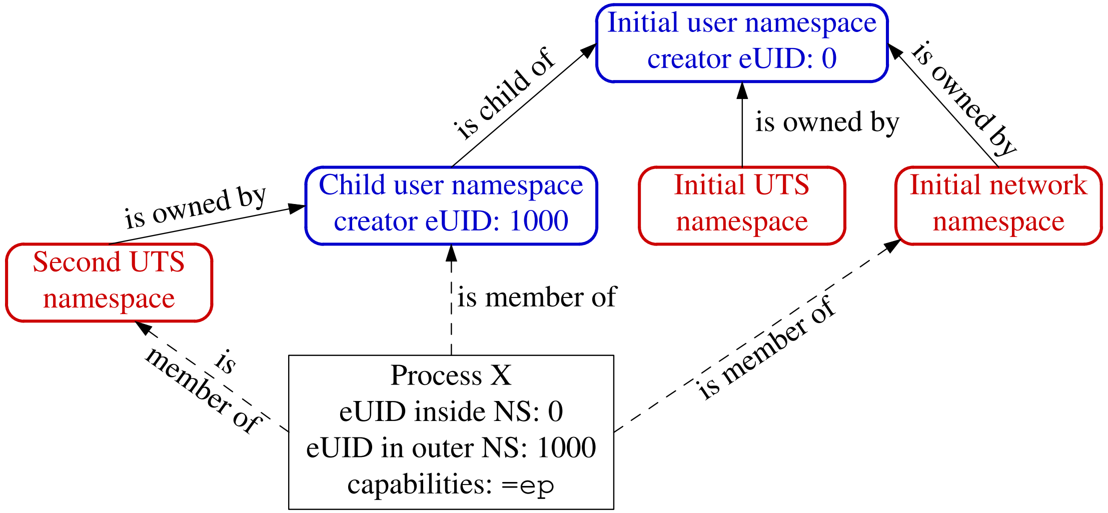


Diagram credit: Michael Kerrisk