Rocky Enterprise Linux 9.2 Manual Pages on command 'pid_namespaces.7'

$ man pid_namespaces.7

PID_NAMESPACES(7)   Linux Programmer's Manual   PID_NAMESPACES(7)

NAME

    pid_namespaces - overview of Linux PID namespaces

DESCRIPTION

    For an overview of namespaces, see namespaces(7).

    PID namespaces isolate the process ID number space, meaning that processes in different

    PID namespaces can have the same PID. PID namespaces allow containers to provide func‐

    tionality such as suspending/resuming the set of processes in the container and migrating

    the container to a new host while the processes inside the container maintain the same

    PIDs.

    PIDs in a new PID namespace start at 1, somewhat like a standalone system, and calls to

    fork(2), vfork(2), or clone(2) will produce processes with PIDs that are unique within the

    namespace.

    Use of PID namespaces requires a kernel that is configured with the CONFIG_PID_NS option.

  The namespace init process

    The first process created in a new namespace (i.e., the process created using clone(2)

    with the CLONE_NEWPID flag, or the first child created by a process after a call to un‐

    share(2) using the CLONE_NEWPID flag) has the PID 1, and is the "init" process for the

    namespace (see init(1)). This process becomes the parent of any child processes that are

    orphaned because a process that resides in this PID namespace terminated (see below for

    further details).

    If the "init" process of a PID namespace terminates, the kernel terminates all of the pro‐

    cesses in the namespace via a SIGKILL signal. This behavior reflects the fact that the

"init" process is essential for the correct operation of a PID namespace. In this case, a subsequent fork(2) into this PID namespace fail with the error ENOMEM; it is not possible to create a new process in a PID namespace whose "init" process has terminated. Such sce? narios can occur when, for example, a process uses an open file descriptor for a /proc/[pid]/ns/pid file corresponding to a process that was in a namespace to setns(2) into that namespace after the "init" process has terminated. Another possible scenario can occur after a call to unshare(2): if the first child subsequently created by a fork(2) terminates, then subsequent calls to fork(2) fail with ENOMEM.

Only signals for which the "init" process has established a signal handler can be sent to the "init" process by other members of the PID namespace. This restriction applies even to privileged processes, and prevents other members of the PID namespace from accidentally killing the "init" process.

Likewise, a process in an ancestor namespace can?subject to the usual permission checks described in kill(2)?send signals to the "init" process of a child PID namespace only if the "init" process has established a handler for that signal. (Within the handler, the siginfo_t si_pid field described in sigaction(2) will be zero.) SIGKILL or SIGSTOP are treated exceptionally: these signals are forcibly delivered when sent from an ancestor PID namespace.   Neither of these signals can be caught by the "init" process, and so will re? sult in the usual actions associated with those signals (respectively, terminating and stopping the process).

Starting with Linux 3.4, the reboot(2) system call causes a signal to be sent to the name? space "init" process. See reboot(2) for more details.

Nesting PID namespaces

PID namespaces can be nested: each PID namespace has a parent, except for the initial ("root") PID namespace. The parent of a PID namespace is the PID namespace of the process that created the namespace using clone(2) or unshare(2). PID namespaces thus form a tree, with all namespaces ultimately tracing their ancestry to the root namespace. Since Linux 3.7, the kernel limits the maximum nesting depth for PID namespaces to 32.

A process is visible to other processes in its PID namespace, and to the processes in each direct ancestor PID namespace going back to the root PID namespace. In this context, "visible" means that one process can be the target of operations by another process using system calls that specify a process ID. Conversely, the processes in a child PID name? space can't see processes in the parent and further removed ancestor namespaces. More

succinctly: a process can see (e.g., send signals with kill(2), set nice values with set?
priority(2), etc.) only processes contained in its own PID namespace and in descendants of
that namespace.

A process has one process ID in each of the layers of the PID namespace hierarchy in which
is visible, and walking back though each direct ancestor namespace through to the root PID
namespace.   System calls that operate on process IDs always operate using the process ID
that is visible in the PID namespace of the caller. A call to getpid(2) always returns
the PID associated with the namespace in which the process was created.

Some processes in a PID namespace may have parents that are outside of the namespace. For
example, the parent of the initial process in the namespace (i.e., the init(1) process
with PID 1) is necessarily in another namespace. Likewise, the direct children of a
process that uses setns(2) to cause its children to join a PID namespace are in a differ?
ent PID namespace from the caller of setns(2). Calls to getppid(2) for such processes re?
turn 0.

While processes may freely descend into child PID namespaces (e.g., using setns(2) with a
PID namespace file descriptor), they may not move in the other direction. That is to say,
processes may not enter any ancestor namespaces (parent, grandparent, etc.). Changing PID
namespaces is a one-way operation.

The NS_GET_PARENT ioctl(2) operation can be used to discover the parental relationship be?
tween PID namespaces; see ioctl_ns(2).

setns(2) and unshare(2) semantics

Calls to setns(2) that specify a PID namespace file descriptor and calls to unshare(2)
with the CLONE_NEWPID flag cause children subsequently created by the caller to be placed
in a different PID namespace from the caller. (Since Linux 4.12, that PID namespace is
shown via the /proc/[pid]/ns/pid_for_children file, as described in namespaces(7).) These
calls do not, however, change the PID namespace of the calling process, because doing so
would change the caller's idea of its own PID (as reported by getpid()), which would break
many applications and libraries.

To put things another way: a process's PID namespace membership is determined when the
process is created and cannot be changed thereafter. Among other things, this means that
the parental relationship between processes mirrors the parental relationship between PID
namespaces: the parent of a process is either in the same namespace or resides in the im?
mediate parent PID namespace.

A process may call unshare(2) with the CLONE_NEWPID flag only once. After it has per?
formed this operation, its /proc/PID/ns/pid_for_children symbolic link will be empty until
the first child is created in the namespace.

Adoption of orphaned children

When a child process becomes orphaned, it is reparented to the "init" process in the PID
namespace of its parent (unless one of the nearer ancestors of the parent employed the
prctl(2) PR_SET_CHILD_SUBREAPER command to mark itself as the reaper of orphaned descen?
dant processes).   Note that because of the setns(2) and unshare(2) semantics described
above, this may be the "init" process in the PID namespace that is the parent of the
child's PID namespace, rather than the "init" process in the child's own PID namespace.

Compatibility of CLONE_NEWPID with other CLONE_* flags

In current versions of Linux, CLONE_NEWPID can't be combined with CLONE_THREAD. Threads
are required to be in the same PID namespace such that the threads in a process can send
signals to each other. Similarly, it must be possible to see all of the threads of a pro?
cesses in the proc(5) filesystem. Additionally, if two threads were in different PID
namespaces, the process ID of the process sending a signal could not be meaningfully en?
coded when a signal is sent (see the description of the siginfo_t type in sigaction(2)).
Since this is computed when a signal is enqueued, a signal queue shared by processes in
multiple PID namespaces would defeat that.

In earlier versions of Linux, CLONE_NEWPID was additionally disallowed (failing with the
error EINVAL) in combination with CLONE_SIGHAND (before Linux 4.3) as well as CLONE_VM
(before Linux 3.12). The changes that lifted these restrictions have also been ported to
earlier stable kernels.

/proc and PID namespaces

A /proc filesystem shows (in the /proc/[pid] directories) only processes visible in the
PID namespace of the process that performed the mount, even if the /proc filesystem is
viewed from processes in other namespaces.

After creating a new PID namespace, it is useful for the child to change its root direc?
tory and mount a new procfs instance at /proc so that tools such as ps(1) work correctly.
If a new mount namespace is simultaneously created by including CLONE_NEWNS in the flags
argument of clone(2) or unshare(2), then it isn't necessary to change the root directory:
a new procfs instance can be mounted directly over /proc.

From a shell, the command to mount /proc is:

```
$ mount -t proc proc /proc
```

Calling readlink(2) on the path /proc/self yields the process ID of the caller in the PID namespace of the procfs mount (i.e., the PID namespace of the process that mounted the procfs). This can be useful for introspection purposes, when a process wants to discover its PID in other namespaces.

/proc files

/proc/sys/kernel/ns_last_pid (since Linux 3.3)

This file (which is virtualized per PID namespace) displays the last PID that was allocated in this PID namespace. When the next PID is allocated, the kernel will search for the lowest unallocated PID that is greater than this value, and when this file is subsequently read it will show that PID.

This file is writable by a process that has the CAP_SYS_ADMIN or (since Linux 5.9) CAP_CHECKPOINT_RESTORE capability inside the user namespace that owns the PID name?space. This makes it possible to determine the PID that is allocated to the next process that is created inside this PID namespace.

Miscellaneous

When a process ID is passed over a UNIX domain socket to a process in a different PID namespace (see the description of SCM_CREDENTIALS in unix(7)), it is translated into the corresponding PID value in the receiving process's PID namespace.

CONFORMING TO

Namespaces are a Linux-specific feature.

EXAMPLES

See user_namespaces(7).

SEE ALSO

clone(2), reboot(2), setns(2), unshare(2), proc(5), capabilities(7), credentials(7), mount_namespaces(7), namespaces(7), user_namespaces(7), switch_root(8)

COLOPHON

This page is part of release 5.10 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.

Linux   2020-11-01   PID_NAMESPACES(7)