lwn.net

Namespaces in operation, part 1: namespaces overview

By Michael KerriskJanuary 4, 2013

10-13 minutes

The Linux 3.8 merge window saw the acceptance of Eric Biederman's sizeable series of user namespace and related patches. Although there remain some details to finish—for example, a number of Linux filesystems are not yet user-namespace aware—the implementation of user namespaces is now functionally complete.

The completion of the user namespaces work is something of a milestone, for a number of reasons. First, this work represents the completion of one of the most complex namespace implementations to date, as evidenced by the fact that it has been around five years since the first steps in the implementation of user namespaces (in Linux 2.6.23). Second, the namespace work is currently at something of a "stable point", with the implementation of most of the existing namespaces being more or less complete. This does not mean that work on namespaces has finished: other namespaces may be added in the future, and there will probably be further extensions to existing namespaces, such as the addition of namespace isolation for the kernel log. Finally, the recent changes in the implementation of user namespaces are something of a game changer in terms of how namespaces can be used: starting with Linux 3.8, unprivileged processes can create user namespaces in which they have full privileges, which in turn allows any other type of namespace to be created inside a user namespace.

Thus, the present moment seems a good point to take an overview of namespaces and a practical look at the namespace API. This is the first of a series of articles that does so: in this article, we provide an overview of the currently available namespaces; in the follow-on articles, we'll show how the namespace APIs can be used in programs.

The namespaces

Currently, Linux implements six different types of namespaces. The purpose of each namespace is to wrap a particular global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. One of the overall goals of namespaces is to support the implementation of containers, a tool for lightweight virtualization (as well as other purposes) that provides a group of processes with the illusion that they are the only processes on the system.

In the discussion below, we present the namespaces in the order that they were implemented (or at least, the order in which the implementations were completed). The `CLONE_NEW*` identifiers listed in parentheses are the names of the constants used to identify namespace types when employing the namespace-related APIs (`clone()`, `unshare()`, and `setns()`) that we will describe in our follow-on articles.

Mount namespaces (`CLONE_NEWNS`, Linux 2.4.19) isolate the set of filesystem mount points seen by a group of processes. Thus, processes in different mount namespaces can have different views of the filesystem hierarchy. With the addition of mount namespaces, the `mount()` and `umount()` system calls ceased operating on a global set of mount points visible to all processes on the system and instead performed operations that affected just the mount namespace associated with the calling process.

One use of mount namespaces is to create environments that are similar to chroot jails. However, by contrast with the use of the `chroot()` system call, mount namespaces are a more secure and flexible tool for this task. Other more sophisticated uses of mount namespaces are also possible. For example, separate mount namespaces can be set up in a master-slave relationship, so that the mount events are automatically propagated from one namespace to another; this allows, for example, an optical disk device that is mounted in one namespace to automatically appear in other namespaces.

Mount namespaces were the first type of namespace to be implemented on Linux, appearing in 2002. This fact accounts for the rather generic "NEWNS" moniker (short for "new namespace"): at that time no one seems to have been thinking that other, different types of namespace might be needed in the future.

UTS namespaces (`CLONE_NEWUTS`, Linux 2.6.19) isolate two system identifiers—`nodename` and `domainname`—returned by the `uname()` system call; the names are set using the `sethostname()` and `setdomainname()` system calls. In the context of containers, the UTS namespaces feature allows each container to have its own hostname and NIS domain name. This can be useful for initialization and configuration scripts that tailor their actions based on these names. The term "UTS" derives from the name of the structure passed to the `uname()` system call: `struct utsname`. The name of that structure in turn derives from "UNIX Time-sharing System".

IPC namespaces (`CLONE_NEWIPC`, Linux 2.6.19) isolate certain interprocess communication (IPC) resources, namely, System V IPC objects and (since Linux 2.6.30) POSIX message queues. The common characteristic of these IPC mechanisms is that IPC objects are identified by mechanisms other than filesystem pathnames. Each IPC namespace has its own set of System V IPC identifiers and its own POSIX message queue filesystem.

PID namespaces (`CLONE_NEWPID`, Linux 2.6.24) isolate the process ID number space. In other words, processes in different PID namespaces can have the same PID. One of the main benefits of PID namespaces is that containers can be migrated between hosts while keeping the same process IDs for the processes inside the container. PID namespaces also allow each container to have its own `init` (PID 1), the "ancestor of all processes" that manages various system initialization tasks and reaps orphaned child processes when they terminate.

From the point of view of a particular PID namespace instance, a process has two PIDs: the PID inside the namespace, and the PID outside the namespace on the host system. PID namespaces can be nested: a process will have one PID for each of the layers of the hierarchy starting from the PID namespace in which it resides through to the root PID namespace. A process can see (e.g., ~~view via `/proc/`*PID* and~~ send signals with `kill()`) only processes contained in its own PID namespace and the namespaces nested below that PID namespace.

Network namespaces (`CLONE_NEWNET`, started in Linux ~~2.4.19~~ 2.6.24 and largely completed by about Linux 2.6.29) provide isolation of the system resources associated with networking. Thus, each network namespace has its own network devices, IP addresses, IP routing tables, `/proc/net` directory, port numbers, and so on.

Network namespaces make containers useful from a networking perspective: each container can have its own (virtual) network device and its own applications that bind to the per-namespace port number space; suitable routing rules in the host system can direct network packets to the network device associated with a specific container. Thus, for example, it is possible to have multiple containerized web servers on the same host system, with each server bound to port 80 in its (per-container) network namespace.

User namespaces (`CLONE_NEWUSER`, started in Linux 2.6.23 and completed in Linux 3.8) isolate the user and group ID number spaces. In other words, a process's user and group IDs can be different inside and outside a user namespace. The most interesting case here is that 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.

Starting in Linux 3.8, unprivileged processes can create user namespaces, which opens up a raft of interesting new possibilities for applications: since an otherwise unprivileged process can hold root privileges inside the user namespace, unprivileged applications now have access to functionality that was formerly limited to root. Eric Biederman has put a lot of effort into making the user namespaces implementation safe and correct. However, the changes wrought by this work are subtle and wide ranging. Thus, it may happen that user namespaces have some as-yet unknown security issues that remain to be found and fixed in the future.

Concluding remarks

It's now around a decade since the implementation of the first Linux namespace. Since that time, the namespace concept has expanded into a more general framework for isolating a range of global resources whose scope was formerly system-wide. As a result, namespaces now provide the basis for a complete lightweight virtualization system, in the form of containers. As the namespace concept has expanded, the associated API has grown—from a single system call (`clone()`) and one or two `/proc` files—to include a number of other system calls and many more files under `/proc`. The details of that API will form the subject of the follow-ups to this article.

Series index

The following list shows later articles in this series, along with their example programs:

- Part 2: the namespaces API
- `demo_uts_namespaces.c`: demonstrate the use of UTS

namespaces

- `ns_exec.c`: join a namespace using `setns()` and execute a command
- `unshare.c`: unshare namespaces and execute a command; similar in concept to `unshare(1)`
- Part 3: PID namespaces
- `pidns_init_sleep.c`: demonstrate PID namespaces
- `multi_pidns.c`: create a series of child processes in nested PID namespaces
- Part 4: more on PID namespaces
- `ns_child_exec.c`: create a child process that executes a shell command in new namespace(s)
- `simple_init.c`: a simple `init(1)`-style program to be used as the `init` program in a PID namespace
- `orphan.c`: demonstrate that a child becomes orphaned and is adopted by the `init` process when its parent exits
- `ns_run.c`: join one or more namespaces using `setns()` and execute a command in those namespaces, possibly inside a child process; similar in concept to `nsenter(1)`
- Part 5: user namespaces
- `demo_userns.c`: simple program to create a user namespace and display process credentials and capabilities
- `userns_child_exec.c`: create a child process that

executes a shell command in new namespace(s); similar to `ns_child_exec.c`, but with additional options for use with user namespaces

- Part 6: more on user namespaces
- `userns_setns_test.c`: test the operation of `setns()` from two different user namespaces.
- Part 7: network namespaces
- Mount namespaces and shared subtrees
- Mount namespaces, mount propagation, and unbindable mounts

Index entries for this article	
Kernel	Containers
Kernel	Namespaces