Namespaces in Operation

Written by Michael Kerrisk and Jake Edge for LWN.net

https://lwn.net/Articles/531114/

January 2013 – June 2016

Compiled December 14, 2021

CSE 522S – Advanced Operating Systems

Washington University in St. louis

Distributed under the Creative Commons Attribution-ShareAlike 4.0 license

Namespaces in operation, part 1: namespaces overview

By **Michael Kerrisk**
January 4, 2013

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.

Namespaces in operation, part 2: the namespaces API

By **Michael Kerrisk**
January 8, 2013

A namespace wraps a 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 resource. Namespaces are used for a variety of purposes, with the most notable being the implementation of containers, a technique for lightweight virtualization. This is the second part in a series of articles that looks in some detail at namespaces and the namespaces API. The first article in this series provided an overview of namespaces. This article looks at the namespaces API in some detail and shows the API in action in a number of example programs.

The namespace API consists of three system calls—`clone()`, `unshare()`, and `setns()`—and a number of `/proc` files. In this article, we'll look at all of these system calls and some of the `/proc` files. In order to specify a namespace type on which to operate, the three system calls make use of the `CLONE_NEW*` constants listed in the previous article: `CLONE_NEWIPC`, `CLONE_NEWNS`, `CLONE_NEWNET`, `CLONE_NEWPID`, `CLONE_NEWUSER`, and `CLONE_NEWUTS`.

Creating a child in a new namespace: clone()

One way of creating a namespace is via the use of `clone()`, a system call that creates a new process. For our purposes, `clone()` has the following prototype:

```
int clone(int (*child_func)(void *), void *child_stack, int flags, void *arg);
```

Essentially, `clone()` is a more general version of the traditional UNIX `fork()` system call whose functionality can be controlled via the `flags` argument. In all, there are more than twenty different `CLONE_*` flags that control various aspects of the operation of `clone()`, including whether the parent and child process share resources such as virtual memory, open file descriptors, and signal dispositions. If one of the `CLONE_NEW*` bits is specified in the call, then a new namespace of the corresponding type is created, and the new process is made a member of that namespace; multiple `CLONE_NEW*` bits can be specified in `flags`.

Our example program (`demo_uts_namespace.c`) uses `clone()` with the `CLONE_NEWUTS` flag to create a UTS namespace. As we saw last week, UTS namespaces isolate two system identifiers—the hostname and the NIS domain name—that are set using the `sethostname()` and `setdomainname()` system calls and returned by the `uname()` system call. You can find the full source of the program here. Below, we'll focus on just some of the key pieces of the program (and for brevity, we'll omit the error checking code that is present in the full version of the program).

The example program takes one command-line argument. When run, it creates a child that executes in a new UTS namespace. Inside that namespace, the child changes the hostname to the string given as the program's command-line argument.

The first significant piece of the main program is the `clone()` call that creates the child process:

```
child_pid = clone(childFunc,
                  child_stack + STACK_SIZE,   /* Points to start of
                                                 downwardly growing stack */
                  CLONE_NEWUTS | SIGCHLD, argv[1]);
```

```
printf("PID of child created by clone() is %ld\n", (long) child_pid);
```

The new child will begin execution in the user-defined function `childFunc()`; that function will receive the final `clone()` argument (`argv[1]`) as its argument. Since `CLONE_NEWUTS` is specified as part of the `flags` argument, the child will execute in a newly created UTS namespace.

The main program then sleeps for a moment. This is a (crude) way of giving the child time to change the hostname in its UTS namespace. The program then uses `uname()` to retrieve the host name in the parent's UTS namespace, and displays that hostname:

```
sleep(1);           /* Give child time to change its hostname */

uname(&uts);
printf("uts.nodename in parent: %s\n", uts.nodename);
```

Meanwhile, the `childFunc()` function executed by the child created by `clone()` first changes the hostname to the value supplied in its argument, and then retrieves and displays the modified hostname:

```
sethostname(arg, strlen(arg);

uname(&uts);
printf("uts.nodename in child:  %s\n", uts.nodename);
```

Before terminating, the child sleeps for a while. This has the effect of keeping the child's UTS namespace open, and gives us a chance to conduct some of the experiments that we show later.

Running the program demonstrates that the parent and child processes have independent UTS namespaces:

```
$ su                    # Need privilege to create a UTS namespace
Password:
# uname -n
antero
# ./demo_uts_namespaces bizarro
PID of child created by clone() is 27514
uts.nodename in child:  bizarro
uts.nodename in parent: antero
```

As with most other namespaces (user namespaces are the exception), creating a UTS namespace requires privilege (specifically, `CAP_SYS_ADMIN`). This is necessary to avoid scenarios where set-user-ID applications could be fooled into doing the wrong thing because the system has an unexpected hostname.

Another possibility is that a set-user-ID application might be using the hostname as part of the name of a lock file. If an unprivileged user could run the application in a UTS namespace with an arbitrary hostname, this would open the application to various attacks. Most simply, this would nullify the effect of the lock file, triggering misbehavior in instances of the application that run in different UTS namespaces. Alternatively, a malicious user could run a set-user-ID application in a UTS namespace with a hostname that causes creation of the lock file to overwrite an important file. (Hostname strings can contain arbitrary characters, including slashes.)

The `/proc/`*`PID`*`/ns` files

Each process has a `/proc/`*`PID`*`/ns` directory that contains one file for each type of namespace. Starting in Linux 3.8, each of these files is a special symbolic link that provides a kind of handle for performing certain operations on the associated namespace for the process.

```
$ ls -l /proc/$$/ns         # $$ is replaced by shell's PID
```

```
total 0
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 ipc -> ipc:[4026531839]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 mnt -> mnt:[4026531840]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 net -> net:[4026531956]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 pid -> pid:[4026531836]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 user -> user:[4026531837]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 uts -> uts:[4026531838]
```

One use of these symbolic links is to discover whether two processes are in the same namespace. The kernel does some magic to ensure that if two processes are in the same namespace, then the inode numbers reported for the corresponding symbolic links in `/proc/PID/ns` will be the same. The inode numbers can be obtained using the `stat()` system call (in the `st_ino` field of the returned structure).

However, the kernel also constructs each of the `/proc/PID/ns` symbolic links so that it points to a name consisting of a string that identifies the namespace type, followed by the inode number. We can examine this name using either the `ls -l` or the `readlink` command.

Let's return to the shell session above where we ran the `demo_uts_namespaces` program. Looking at the `/proc/PID/ns` symbolic links for the parent and child process provides an alternative method of checking whether the two processes are in the same or different UTS namespaces:

```
^Z                                  # Stop parent and child
[1]+  Stopped                 ./demo_uts_namespaces bizarro
# jobs -l                           # Show PID of parent process
[1]+ 27513 Stopped                 ./demo_uts_namespaces bizarro
# readlink /proc/27513/ns/uts       # Show parent UTS namespace
uts:[4026531838]
# readlink /proc/27514/ns/uts       # Show child UTS namespace
uts:[4026532338]
```

As can be seen, the content of the `/proc/PID/ns/uts` symbolic links differs, indicating that the two processes are in different UTS namespaces.

The `/proc/PID/ns` symbolic links also serve other purposes. If we open one of these files, then the namespace will continue to exist as long as the file descriptor remains open, even if all processes in the namespace terminate. The same effect can also be obtained by bind mounting one of the symbolic links to another location in the file system:

```
# touch ~/uts                                 # Create mount point
# mount --bind /proc/27514/ns/uts ~/uts
```

Before Linux 3.8, the files in `/proc/PID/ns` were hard links rather than special symbolic links of the form described above. In addition, only the `ipc`, `net`, and `uts` files were present.

Joining an existing namespace: setns()

Keeping a namespace open when it contains no processes is of course only useful if we intend to later add processes to it. That is the task of the `setns()` system call, which allows the calling process to join an existing namespace:

```
int setns(int fd, int nstype);
```

More precisely, `setns()` disassociates the calling process from one instance of a particular namespace type and reassociates the process with another instance of the same namespace type.

The `fd` argument specifies the namespace to join; it is a file descriptor that refers to one of the symbolic links

in a /proc/*PID*/ns directory. That file descriptor can be obtained either by opening one of those symbolic links directly or by opening a file that was bind mounted to one of the links.

The nstype argument allows the caller to check the type of namespace that fd refers to. If this argument is specified as zero, no check is performed. This can be useful if the caller already knows the namespace type, or does not care about the type. The example program that we discuss in a moment (ns_exec.c) falls into the latter category: it is designed to work with any namespace type. Specifying nstype instead as one of the CLONE_NEW* constants causes the kernel to verify that fd is a file descriptor for the corresponding namespace type. This can be useful if, for example, the caller was passed the file descriptor via a UNIX domain socket and needs to verify what type of namespace it refers to.

Using setns() and execve() (or one of the other exec() functions) allows us to construct a simple but useful tool: a program that joins a specified namespace and then executes a command in that namespace.

Our program (ns_exec.c, whose full source can be found here) takes two or more command-line arguments. The first argument is the pathname of a /proc/*PID*/ns/* symbolic link (or a file that is bind mounted to one of those symbolic links). The remaining arguments are the name of a program to be executed inside the namespace that corresponds to that symbolic link and optional command-line arguments to be given to that program. The key steps in the program are the following:

```
fd = open(argv[1], O_RDONLY);   /* Get descriptor for namespace */

setns(fd, 0);                   /* Join that namespace */

execvp(argv[2], &argv[2]);      /* Execute a command in namespace */
```

An interesting program to execute inside a namespace is, of course, a shell. We can use the bind mount for the UTS namespace that we created earlier in conjunction with the ns_exec program to execute a shell in the new UTS namespace created by our invocation of demo_uts_namespaces:

```
# ./ns_exec ~/uts /bin/bash     # ~/uts is bound to /proc/27514/ns/uts
My PID is: 28788
```

We can then verify that the shell is in the same UTS namespace as the child process created by demo_uts_namespaces, both by inspecting the hostname and by comparing the inode numbers of the /proc/*PID*/ns/uts files:

```
# hostname
bizarro
# readlink /proc/27514/ns/uts
uts:[4026532338]
# readlink /proc/$$/ns/uts      # $$ is replaced by shell's PID
uts:[4026532338]
```

In earlier kernel versions, it was not possible to use setns() to join mount, PID, and user namespaces, but, starting with Linux 3.8, setns() now supports joining all namespace types.

Leaving a namespace: unshare()

The final system call in the namespaces API is unshare():

```
int unshare(int flags);
```

The unshare() system call provides functionality similar to clone(), but operates on the calling process: it creates the new namespaces specified by the CLONE_NEW* bits in its flags argument and makes the caller a

member of the namespaces. (As with `clone()`, `unshare()` provides functionality beyond working with namespaces that we'll ignore here.) The 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()`).

Leaving aside the other effects of the `clone()` system call, a call of the form:

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

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

```
if (fork() == 0)
    unshare(CLONE_NEWXXX);      /* Executed in the child process */
```

One use of the `unshare()` system call is in the implementation of the `unshare` command, which allows the user to execute a command in a separate namespace from the shell. The general form of this command is:

```
unshare [options] program [arguments]
```

The `options` are command-line flags that specify the namespaces to unshare before executing `program` with the specified `arguments`.

The key steps in the implementation of the `unshare` command are straightforward:

```
/* Code to initialize 'flags' according to command-line options
   omitted */

unshare(flags);

/* Now execute 'program' with 'arguments'; 'optind' is the index
   of the next command-line argument after options */

execvp(argv[optind], &argv[optind]);
```

A simple implementation of the `unshare` command (`unshare.c`) can be found here.

In the following shell session, we use our `unshare.c` program to execute a shell in a separate mount namespace. As we noted in last week's article, mount namespaces isolate the set of filesystem mount points seen by a group of processes, allowing processes in different mount namespaces to have different views of the filesystem hierarchy.

```
# echo $$                           # Show PID of shell
8490
# cat /proc/8490/mounts | grep mq   # Show one of the mounts in namespace
mqueue /dev/mqueue mqueue rw,seclabel,relatime 0 0
# readlink /proc/8490/ns/mnt        # Show mount namespace ID
mnt:[4026531840]
# ./unshare -m /bin/bash            # Start new shell in separate mount namespace
# readlink /proc/$$/ns/mnt          # Show mount namespace ID
mnt:[4026532325]
```

Comparing the output of the two `readlink` commands shows that the two shells are in separate mount namespaces. Altering the set of mount points in one of the namespaces and checking whether that change is visible in the other namespace provides another way of demonstrating that the two programs are in separate namespaces:

```
# umount /dev/mqueue                # Remove a mount point in this shell
# cat /proc/$$/mounts | grep mq     # Verify that mount point is gone
# cat /proc/8490/mounts | grep mq   # Is it still present in the other namespace?
mqueue /dev/mqueue mqueue rw,seclabel,relatime 0 0
```

As can be seen from the output of the last two commands, the `/dev/mqueue` mount point has disappeared in one mount namespace, but continues to exist in the other.

Concluding remarks

In this article we've looked at the fundamental pieces of the namespace API and how they are employed together. In the follow-on articles, we'll look in more depth at some other namespaces, in particular, the PID and user namespaces; user namespaces open up a range of new possibilities for applications to use kernel interfaces that were formerly restricted to privileged applications.

(2013-01-15: updated the concluding remarks to reflect the fact that there will be more than one following article.)

Namespaces in operation, part 3: PID namespaces

By **Michael Kerrisk**
January 16, 2013

Following on from our two earlier namespaces articles (Part 1: namespaces overview and Part 2: the namespaces API), we now turn to look at PID namespaces. The global resource isolated by PID namespaces is the process ID number space. This means that processes in different PID namespaces can have the same process ID. PID namespaces are used to implement containers that can be migrated between host systems while keeping the same process IDs for the processes inside the container.

As with processes on a traditional Linux (or UNIX) system, the process IDs *within* a PID namespace are unique, and are assigned sequentially starting with PID 1. Likewise, as on a traditional Linux system, PID 1—the `init` process—is special: it is the first process created within the namespace, and it performs certain management tasks within the namespace.

First investigations

A new PID namespace is created by calling `clone()` with the `CLONE_NEWPID` flag. We'll show a simple example program that creates a new PID namespace using `clone()` and use that program to map out a few of the basic concepts of PID namespaces. The complete source of the program (`pidns_init_sleep.c`) can be found here. As with the previous article in this series, in the interests of brevity, we omit the error-checking code that is present in the full versions of the example program when discussing it in the body of the article.

The main program creates a new PID namespace using `clone()`, and displays the PID of the resulting child:

```
child_pid = clone(childFunc,
                child_stack + STACK_SIZE,   /* Points to start of
                                               downwardly growing stack */
                CLONE_NEWPID | SIGCHLD, argv[1]);

printf("PID returned by clone(): %ld\n", (long) child_pid);
```

The new child process starts execution in `childFunc()`, which receives the last argument of the `clone()` call (`argv[1]`) as its argument. The purpose of this argument will become clear later.

The `childFunc()` function displays the process ID and parent process ID of the child created by `clone()` and concludes by executing the standard `sleep` program:

```
printf("childFunc(): PID = %ld\n", (long) getpid());
printf("ChildFunc(): PPID = %ld\n", (long) getppid());
...
execlp("sleep", "sleep", "1000", (char *) NULL);
```

The main virtue of executing the `sleep` program is that it provides us with an easy way of distinguishing the child process from the parent in process listings.

When we run this program, the first lines of output are as follows:

```
$ su          # Need privilege to create a PID namespace
Password:
# ./pidns_init_sleep /proc2
PID returned by clone(): 27656
childFunc(): PID  = 1
childFunc(): PPID = 0
```

```
Mounting procfs at /proc2
```

The first two lines line of output from `pidns_init_sleep` show the PID of the child process from the perspective of two different PID namespaces: the namespace of the caller of `clone()` and the namespace in which the child resides. In other words, the child process has two PIDs: 27656 in the parent namespace, and 1 in the new PID namespace created by the `clone()` call.

The next line of output shows the parent process ID of the child, within the context of the PID namespace in which the child resides (i.e., the value returned by `getppid()`). The parent PID is 0, demonstrating a small quirk in the operation of PID namespaces. As we detail below, PID namespaces form a hierarchy: a process can "see" only those processes contained in its own PID namespace and in the child namespaces nested below that PID namespace. Because the parent of the child created by `clone()` is in a different namespace, the child cannot "see" the parent; therefore, `getppid()` reports the parent PID as being zero.

For an explanation of the last line of output from `pidns_init_sleep`, we need to return to a piece of code that we skipped when discussing the implementation of the `childFunc()` function.

`/proc/`*PID* and PID namespaces

Each process on a Linux system has a `/proc/`*PID* directory that contains pseudo-files describing the process. This scheme translates directly into the PID namespaces model. Within a PID namespace, the `/proc/`*PID* directories show information only about processes within that PID namespace or one of its descendant namespaces.

However, in order to make the `/proc/`*PID* directories that correspond to a PID namespace visible, the proc filesystem ("procfs" for short) needs to be mounted from within that PID namespace. From a shell running inside the PID namespace (perhaps invoked via the `system()` library function), we can do this using a `mount` command of the following form:

```
# mount -t proc proc /mount_point
```

Alternatively, a procfs can be mounted using the `mount()` system call, as is done inside our program's `childFunc()` function:

```
mkdir(mount_point, 0555);       /* Create directory for mount point */
mount("proc", mount_point, "proc", 0, NULL);
printf("Mounting procfs at %s\n", mount_point);
```

The `mount_point` variable is initialized from the string supplied as the command-line argument when invoking `pidns_init_sleep`.

In our example shell session running `pidns_init_sleep` above, we mounted the new procfs at `/proc2`. In real world usage, the procfs would (if it is required) usually be mounted at the usual location, `/proc`, using either of the techniques that we describe in a moment. However, mounting the procfs at `/proc2` during our demonstration provides an easy way to avoid creating problems for the rest of the processes on the system: since those processes are in the same *mount* namespace as our test program, changing the filesystem mounted at `/proc` would confuse the rest of the system by making the `/proc/`*PID* directories for the root PID namespace invisible.

Thus, in our shell session the procfs mounted at `/proc` will show the *PID* subdirectories for the processes visible from the parent PID namespace, while the procfs mounted at `/proc2` will show the *PID* subdirectories for processes that reside in the child PID namespace. In passing, it's worth mentioning that although the processes in the child PID namespace will be able to see the *PID* directories exposed by the `/proc` mount

point, those PIDs will not be meaningful for the processes in the child PID namespace, since system calls made by those processes interpret PIDs in the context of the PID namespace in which they reside.

Having a procfs mounted at the traditional `/proc` mount point is necessary if we want various tools such as `ps` to work correctly inside the child PID namespace, because those tools rely on information found at `/proc`. There are two ways to achieve this without affecting the `/proc` mount point used by parent PID namespace. First, if the child process is created using the `CLONE_NEWNS` flag, then the child will be in a different mount namespace from the rest of the system. In this case, mounting the new procfs at `/proc` would not cause any problems. Alternatively, instead of employing the `CLONE_NEWNS` flag, the child could change its root directory with `chroot()` and mount a procfs at `/proc`.

Let's return to the shell session running `pidns_init_sleep`. We stop the program and use `ps` to examine some details of the parent and child processes within the context of the parent namespace:

```
^Z                          Stop the program, placing in background
[1]+  Stopped                 ./pidns_init_sleep /proc2
# ps -C sleep -C pidns_init_sleep -o "pid ppid stat cmd"
  PID  PPID STAT CMD
27655 27090 T    ./pidns_init_sleep /proc2
27656 27655 S    sleep 600
```

The "PPID" value (27655) in the last line of output above shows that the parent of the process executing `sleep` is the process executing `pidns_init_sleep`.

By using the `readlink` command to display the (differing) contents of the `/proc/`*PID*`/ns/pid` symbolic links (explained in last week's article), we can see that the two processes are in separate PID namespaces:

```
# readlink /proc/27655/ns/pid
pid:[4026531836]
# readlink /proc/27656/ns/pid
pid:[4026532412]
```

At this point, we can also use our newly mounted procfs to obtain information about processes in the new PID namespace, from the perspective of that namespace. To begin with, we can obtain a list of PIDs in the namespace using the following command:

```
# ls -d /proc2/[1-9]*
/proc2/1
```

As can be seen, the PID namespace contains just one process, whose PID (inside the namespace) is 1. We can also use the `/proc/`*PID*`/status` file as a different method of obtaining some of the same information about that process that we already saw earlier in the shell session:

```
# cat /proc2/1/status | egrep '^(Name|PP*id)'
Name:   sleep
Pid:    1
PPid:   0
```

The `PPid` field in the file is 0, matching the fact that `getppid()` reports that the parent process ID for the child is 0.

Nested PID namespaces

As noted earlier, PID namespaces are hierarchically nested in parent-child relationships. Within a PID namespace, it is possible to see all other processes in the same namespace, as well as all processes that are members of descendant namespaces. Here, "see" means being able to make system calls that operate on

specific PIDs (e.g., using `kill()` to send a signal to process). Processes in a child PID namespace cannot see processes that exist (only) in the parent PID namespace (or further removed ancestor namespaces).

A process will have one PID in each of the layers of the PID namespace hierarchy starting from the PID namespace in which it resides through to the root PID namespace. Calls to `getpid()` always report the PID associated with the namespace in which the process resides.

We can use the program shown here (`multi_pidns.c`) to show that a process has different PIDs in each of the namespaces in which it is visible. In the interests of brevity, we will simply explain what the program does, rather than walking though its code.

The program recursively creates a series of child process in nested PID namespaces. The command-line argument specified when invoking the program determines how many children and PID namespaces to create:

```
# ./multi_pidns 5
```

In addition to creating a new child process, each recursive step mounts a procfs filesystem at a uniquely named mount point. At the end of the recursion, the last child executes the `sleep` program. The above command line yields the following output:

```
Mounting procfs at /proc4
Mounting procfs at /proc3
Mounting procfs at /proc2
Mounting procfs at /proc1
Mounting procfs at /proc0
Final child sleeping
```

Looking at the PIDs in each procfs, we see that each successive procfs "level" contains fewer PIDs, reflecting the fact that each PID namespace shows only the processes that are members of that PID namespace or its descendant namespaces:

```
^Z                          Stop the program, placing in background
[1]+  Stopped            ./multi_pidns 5
# ls -d /proc4/[1-9]*       Topmost PID namespace created by program
/proc4/1  /proc4/2  /proc4/3  /proc4/4  /proc4/5
# ls -d /proc3/[1-9]*
/proc3/1  /proc3/2  /proc3/3  /proc3/4
# ls -d /proc2/[1-9]*
/proc2/1  /proc2/2  /proc2/3
# ls -d /proc1/[1-9]*
/proc1/1  /proc1/2
# ls -d /proc0/[1-9]*       Bottommost PID namespace
/proc0/1
```

A suitable `grep` command allows us to see the PID of the process at the tail end of the recursion (i.e., the process executing `sleep` in the most deeply nested namespace) in all of the namespaces where it is visible:

```
# grep -H 'Name:.*sleep' /proc?/[1-9]*/status
/proc0/1/status:Name:       sleep
/proc1/2/status:Name:       sleep
/proc2/3/status:Name:       sleep
/proc3/4/status:Name:       sleep
/proc4/5/status:Name:       sleep
```

In other words, in the most deeply nested PID namespace (`/proc0`), the process executing `sleep` has the PID 1, and in the topmost PID namespace created (`/proc4`), that process has the PID 5.

If you run the test programs shown in this article, it's worth mentioning that they will leave behind mount

points and mount directories. After terminating the programs, shell commands such as the following should suffice to clean things up:

```
# umount /proc?
# rmdir /proc?
```

Concluding remarks

In this article, we've looked in quite some detail at the operation of PID namespaces. In the next article, we'll fill out the description with a discussion of the PID namespace `init` process, as well as a few other details of the PID namespaces API.

Namespaces in operation, part 4: more on PID namespaces

By **Michael Kerrisk**
January 23, 2013

In this article, we continue last week's discussion of PID namespaces (and extend our ongoing series on namespaces). One use of PID namespaces is to implement a package of processes (a container) that behaves like a self-contained Linux system. A key part of a traditional system—and likewise a PID namespace container—is the `init` process. Thus, we'll look at the special role of the `init` process and note one or two areas where it differs from the traditional `init` process. In addition, we'll look at some other details of the namespaces API as it applies to PID namespaces.

The PID namespace `init` process

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 (e.g., perhaps starting other processes that should be a standard part of the namespace) and becomes the parent for processes in the namespace that become orphaned.

In order to explain the operation of PID namespaces, we'll make use of a few purpose-built example programs. The first of these programs, `ns_child_exec.c`, has the following command-line syntax:

```
ns_child_exec [options] command [arguments]
```

The `ns_child_exec` program uses the `clone()` system call to create a child process; the child then executes the given `command` with the optional `arguments`. The main purpose of the `options` is to specify new namespaces that should be created as part of the `clone()` call. For example, the `-p` option causes the child to be created in a new PID namespace, as in the following example:

```
$ su                    # Need privilege to create a PID namespace
Password:
# ./ns_child_exec -p sh -c 'echo $$'
1
```

That command line creates a child in a new PID namespace to execute a shell `echo` command that displays the shell's PID. With a PID of 1, the shell was the `init` process for the PID namespace that (briefly) existed while the shell was running.

Our next example program, `simple_init.c`, is a program that we'll execute as the `init` process of a PID namespace. This program is designed to allow us to demonstrate some features of PID namespaces and the `init` process.

The `simple_init` program performs the two main functions of `init`. One of these functions is "system initialization". Most `init` systems are more complex programs that take a table-driven approach to system initialization. Our (much simpler) `simple_init` program provides a simple shell facility that allows the user to manually execute any shell commands that might be needed to initialize the namespace; this approach also allows us to freely execute shell commands in order to conduct experiments in the namespace. The other function performed by `simple_init` is to reap the status of its terminated children using `waitpid()`.

Thus, for example, we can use the `ns_child_exec` program in conjunction with `simple_init` to fire up an `init` process that runs in a new PID namespace:

```
# ./ns_child_exec -p ./simple_init
init$
```

The `init$` prompt indicates that the `simple_init` program is ready to read and execute a shell command.

We'll now use the two programs we've presented so far in conjunction with another small program, `orphan.c`, to demonstrate that processes that become orphaned inside a PID namespace are adopted by the PID namespace `init` process, rather than the system-wide `init` process.

The `orphan` program performs a `fork()` to create a child process. The parent process then exits while the child continues to run; when the parent exits, the child becomes an orphan. The child executes a loop that continues until it becomes an orphan (i.e., `getppid()` returns 1); once the child becomes an orphan, it terminates. The parent and the child print messages so that we can see when the two processes terminate and when the child becomes an orphan.

In order to see what that our `simple_init` program reaps the orphaned child process, we'll employ that program's `-v` option, which causes it to produce verbose messages about the children that it creates and the terminated children whose status it reaps:

```
# ./ns_child_exec -p ./simple_init -v
        init: my PID is 1
init$ ./orphan
        init: created child 2
Parent (PID=2) created child with PID 3
Parent (PID=2; PPID=1) terminating
        init: SIGCHLD handler: PID 2 terminated
init$                    # simple_init prompt interleaved with output from child
Child  (PID=3) now an orphan (parent PID=1)
Child  (PID=3) terminating
        init: SIGCHLD handler: PID 3 terminated
```

In the above output, the indented messages prefixed with `init:` are printed by the `simple_init` program's verbose mode. All of the other messages (other than the `init$` prompts) are produced by the `orphan` program. From the output, we can see that the child process (PID 3) becomes an orphan when its parent (PID 2) terminates. At that point, the child is adopted by the PID namespace `init` process (PID 1), which reaps the child when it terminates.

Signals and the `init` process

The traditional Linux `init` process is treated specially with respect to signals. The only signals that can be delivered to `init` are those for which the process has established a signal handler; all other signals are ignored. This prevents the `init` process—whose presence is essential for the stable operation of the system—from being accidentally killed, even by the superuser.

PID namespaces implement some analogous behavior for the namespace-specific `init` process. Other processes in the namespace (even privileged processes) can send only those signals for which the `init` process has established a handler. This prevents members of the namespace from inadvertently killing a process that has an essential role in the namespace. Note, however, that (as for the traditional `init` process) the kernel can still generate signals for the PID namespace `init` process in all of the usual circumstances (e.g., hardware exceptions, terminal-generated signals such as `SIGTTOU`, and expiration of a timer).

Signals can also (subject to the usual permission checks) be sent to the PID namespace `init` process by processes in ancestor PID namespaces. Again, only the signals for which the `init` process has established a handler can be sent, with two exceptions: `SIGKILL` and `SIGSTOP`. When a process in an ancestor PID namespace sends these two signals to the `init` process, they are forcibly delivered (and can't be caught). The

SIGSTOP signal stops the init process; SIGKILL terminates it. Since the init process is essential to the functioning of the PID namespace, if the init process is terminated by SIGKILL (or it terminates for any other reason), the kernel terminates all other processes in the namespace by sending them a SIGKILL signal.

Normally, a PID namespace will also be destroyed when its init process terminates. However, there is an unusual corner case: the namespace won't be destroyed as long as a /proc/*PID*/ns/pid file for one of the processes in that namespaces is bind mounted or held open. However, it is not possible to create new processes in the namespace (via setns() plus fork()): the lack of an init process is detected during the fork() call, which fails with an ENOMEM error (the traditional error indicating that a PID cannot be allocated). In other words, the PID namespace continues to exist, but is no longer usable.

Mounting a procfs filesystem (revisited)

In the previous article in this series, the /proc filesystems (procfs) for the PID namespaces were mounted at various locations other than the traditional /proc mount point. This allowed us to use shell commands to look at the contents of the /proc/*PID* directories that corresponded to each of the new PID namespace while at the same time using the ps command to look at the processes visible in the root PID namespace.

However, tools such as ps rely on the contents of the procfs mounted at /proc to obtain the information that they require. Therefore, if we want ps to operate correctly inside a PID namespace, we need to mount a procfs for that namespace. Since the simple_init program permits us to execute shell commands, we can perform this task from the command line, using the mount command:

```
# ./ns_child_exec -p -m ./simple_init
init$ mount -t proc proc /proc
init$ ps a
  PID TTY      STAT   TIME COMMAND
    1 pts/8    S      0:00 ./simple_init
    3 pts/8    R+     0:00 ps a
```

The ps a command lists all processes accessible via /proc. In this case, we see only two processes, reflecting the fact that there are only two processes running in the namespace.

When running the ns_child_exec command above, we employed that program's -m option, which places the child that it creates (i.e., the process running simple_init) inside a separate *mount* namespace. As a consequence, the mount command does not affect the /proc mount seen by processes outside the namespace.

unshare() and setns()

In the second article in this series, we described two system calls that are part of the namespaces API: unshare() and setns(). Since Linux 3.8, these system calls can be employed with PID namespaces, but they have some idiosyncrasies when used with those namespaces.

Specifying the CLONE_NEWPID flag in a call to unshare() creates a new PID namespace, but does *not* place the caller in the new namespace. Rather, 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.

The setns() system call now supports PID namespaces:

```
setns(fd, 0);   /* Second argument can be CLONE_NEWPID to force a
                   check that 'fd' refers to a PID namespace */
```

The fd argument is a file descriptor that identifies a PID namespace that is a descendant of the PID namespace

of the caller; that file descriptor is obtained by opening the `/proc/PID/ns/pid` file for one of the processes in the target namespace. As with `unshare()`, `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.

We can use an enhanced version of the `ns_exec.c` program that we presented in the second article in this series to demonstrate some aspects of using `setns()` with PID namespaces that appear surprising until we understand what is going on. The new program, `ns_run.c`, has the following syntax:

```
ns_run [-f] [-n /proc/PID/ns/FILE]... command [arguments]
```

The program uses `setns()` to join the namespaces specified by the `/proc/PID/ns` files contained within `-n` options. It then goes on to execute the given `command` with optional `arguments`. If the `-f` option is specified, it uses `fork()` to create a child process that is used to execute the command.

Suppose that, in one terminal window, we fire up our `simple_init` program in a new PID namespace in the usual manner, with verbose logging so that we are informed when it reaps child processes:

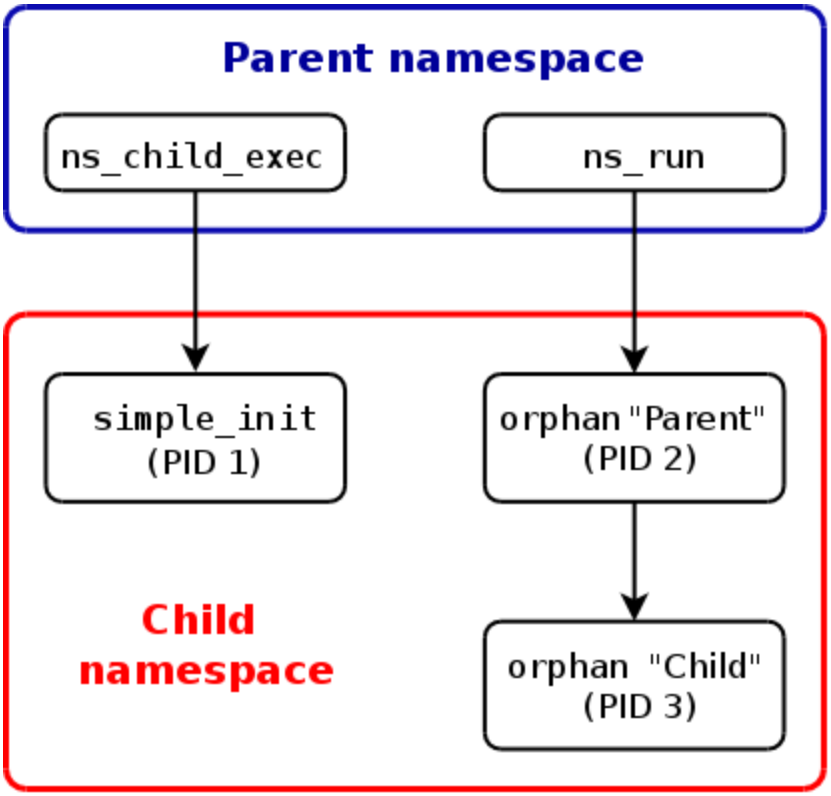
```
# ./ns_child_exec -p ./simple_init -v
        init: my PID is 1
init$
```

Then we switch to a second terminal window where we use the `ns_run` program to execute our `orphan` program. This will have the effect of creating two processes in the PID namespace governed by `simple_init`:

```
# ps -C sleep -C simple_init
  PID TTY          TIME CMD
 9147 pts/8    00:00:00 simple_init
 # ./ns_run -f -n /proc/9147/ns/pid ./orphan
 Parent (PID=2) created child with PID 3
 Parent (PID=2; PPID=0) terminating
 #
 Child  (PID=3) now an orphan (parent PID=1)
 Child  (PID=3) terminating
```

Looking at the output from the "Parent" process (PID 2) created when the `orphan` program is executed, we see that its parent process ID is 0. This reflects the fact that the process that started the `orphan` process (`ns_run`) is in a different namespace—one whose members are invisible to the "Parent" process. As already noted in the previous article, `getppid()` returns 0 in this case.

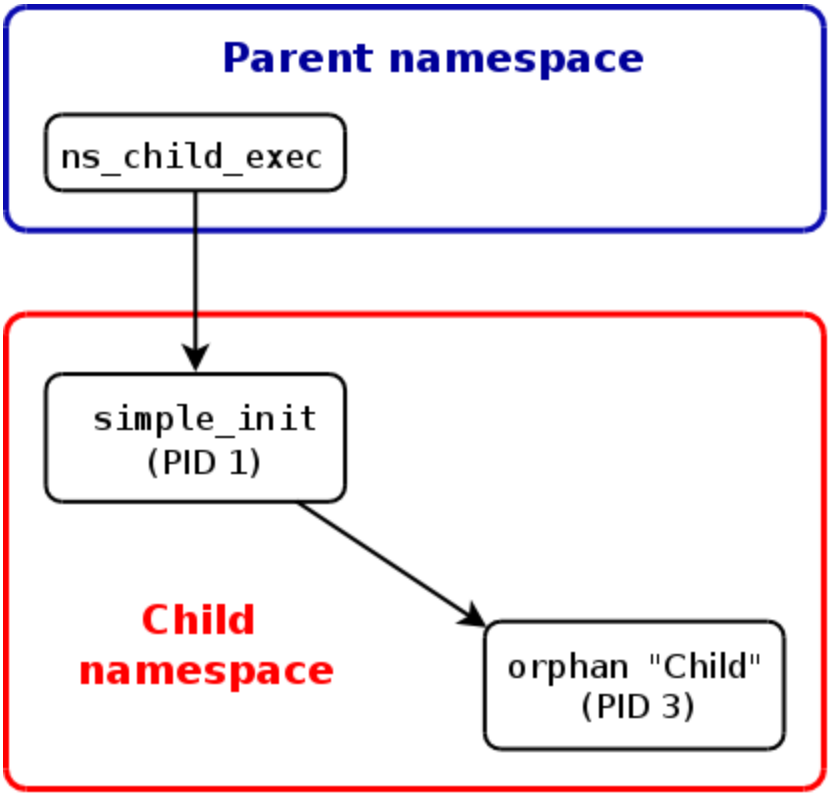
The following diagram shows the relationships of the various processes before the `orphan` "Parent" process terminates. The arrows indicate parent-child relationships between processes.



Returning to the window running the `simple_init` program, we see the following output:

```
init: SIGCHLD handler: PID 3 terminated
```

The "Child" process (PID 3) created by the `orphan` program was reaped by `simple_init`, but the "Parent" process (PID 2) was not. This is because the "Parent" process was reaped by its parent (`ns_run`) in a different namespace. The following diagram shows the processes and their relationships after the `orphan` "Parent" process has terminated and before the "Child" terminates.



It's worth emphasizing that `setns()` and `unshare()` treat PID namespaces specially. For other types of namespaces, these system calls *do* change the namespace of the caller. The reason that these system calls do not change the PID namespace of the calling process is because becoming a member of another PID namespace would cause the process's idea of its own PID to change, since `getpid()` reports the process's PID with respect to the PID namespace in which the process resides. Many user-space programs and libraries rely on the assumption that a process's PID (as reported by `getpid()`) is constant (in fact, the GNU C library `getpid()` wrapper function *caches* the PID); those programs would break if a process's PID changed. To put things another way: a process's PID namespace membership is determined when the process is created, and (unlike other types of namespace membership) cannot be changed thereafter.

Concluding remarks

In this article we've looked at the special role of the PID namespace `init` process, shown how to mount a procfs for a PID namespace so that it can be used by tools such as `ps`, and looked at some of the peculiarities of `unshare()` and `setns()` when employed with PID namespaces. This completes our discussion of PID namespaces; in the next article, we'll turn to look at user namespaces.

Namespaces in operation, part 5: User namespaces

By **Michael Kerrisk**
February 27, 2013

Continuing our ongoing series on namespaces, this article looks more closely at user namespaces, a feature whose implementation was (largely) completed in Linux 3.8. (The remaining work consists of changes for XFS and a number of other filesystems; the latter has already been merged for 3.9.) User namespaces allow per-namespace mappings of user and group IDs. This means that a process's user and group IDs inside a user namespace can be different from its IDs outside of the namespace. Most notably, a process can have a nonzero user ID outside a namespace while at the same time having a user ID of zero inside the namespace; in other words, the process is unprivileged for operations outside the user namespace but has root privileges inside the namespace.

Creating user namespaces

User namespaces are created by specifying the `CLONE_NEWUSER` flag when calling `clone()` or `unshare()`. Starting with Linux 3.8 (and unlike the flags used for creating other types of namespaces), no privilege is required to create a user namespace. In our examples below, all of the user namespaces are created using the unprivileged user ID 1000.

To begin investigating user namespaces, we'll make use of a small program, `demo_userns.c`, that creates a child in a new user namespace. The child simply displays its effective user and group IDs as well as its capabilities. Running this program as an unprivileged user produces the following result:

```
$ id -u          # Display effective user ID of shell process
1000
$ id -g          # Effective group ID of shell
1000
$ ./demo_userns
eUID = 65534;  eGID = 65534;  capabilities: =ep
```

The output from this program shows some interesting details. One of these is the capabilities that were assigned to the child process. The string "`=ep`" (produced by the library function `cap_to_text()`, which converts capability sets to a textual representation) indicates that the child has a full set of permitted and effective capabilities, even though the program was run from an unprivileged account. 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.

The second point of interest is the user and group IDs of the child process. As noted above, a process's user and group IDs inside and outside a user namespace can be different. However, there needs to be a mapping from the user IDs inside a user namespace to a corresponding set of user IDs outside the namespace; the same is true of group IDs. This allows the system to perform the appropriate permission checks when a process in a user namespace performs operations that affect the wider system (e.g., sending a signal to a process outside the namespace or accessing a file).

System calls that return process user and group IDs—for example, `getuid()` and `getgid()`—always return credentials as they appear inside the user namespace in which the calling process resides. If a user ID has no mapping inside the namespace, then system calls that return user IDs return the value defined in the file `/proc/sys/kernel/overflowuid`, which on a standard system defaults to the value 65534. Initially, a user namespace has no user ID mapping, so all user IDs inside the namespace map to this value. Likewise, a new user namespace has no mappings for group IDs, and all unmapped group IDs map to `/proc/sys/kernel`

`/overflowgid` (which has the same default as `overflowuid`).

There is one other important point worth noting that can't be gleaned from the output above. Although the new process has a full set of capabilities in the new user namespace, it has no capabilities in the parent namespace. This is true regardless of the credentials and capabilities of the process that calls `clone()`. In particular, even if root employs `clone(CLONE_NEWUSER)`, the resulting child process will have no capabilities in the parent namespace.

One final point to be made about the creation of user namespaces is that namespaces can be nested; that is, each user namespace (other than the initial user namespace) has a parent user namespace, and can have zero or more child user namespaces. The parent of a user namespace is the user namespace of the process that creates the user namespace via a call to `clone()` or `unshare()` with the `CLONE_NEWUSER` flag. The significance of the parent-child relationship between user namespaces will become clearer in the remainder of this article.

Mapping user and group IDs

Normally, one of the first steps after creating a new user namespace is to define the mappings used for the user and group IDs of the processes that will be created in that namespace. This is done by writing mapping information to the `/proc/`*PID*`/uid_map` and `/proc/`*PID*`/gid_map` files corresponding to one of the processes in the user namespace. (Initially, these two files are empty.) This information consists of one or more lines, each of which contains three values separated by white space:

```
ID-inside-ns   ID-outside-ns   length
```

Together, the `ID-inside-ns` and `length` values define a range of IDs inside the namespace that are to be mapped to an ID range of the same length outside the namespace. The `ID-outside-ns` value specifies the starting point of the outside range. How `ID-outside-ns` is interpreted depends on the whether the process opening the file `/proc/`*PID*`/uid_map` (or `/proc/`*PID*`/gid_map`) is in the same user namespace as the process *PID*:

- If the two processes are in the same namespace, then `ID-outside-ns` is interpreted as a user ID (group ID) in the parent user namespace of the process *PID*. The common case here is that a process is writing to its own mapping file (`/proc/self/uid_map` or `/proc/self/gid_map`).
- If the two processes are in different namespaces, then `ID-outside-ns` is interpreted as a user ID (group ID) in the user namespace of the process opening `/proc/`*PID*`/uid_map` (`/proc/`*PID*`/gid_map`). The writing process is then defining the mapping relative to its own user namespace.

Suppose that we once more invoke our `demo_userns` program, but this time with a single command-line argument (any string). This causes the program to loop, continuously displaying credentials and capabilities every few seconds:

```
$ ./demo_userns x
eUID = 65534;  eGID = 65534;  capabilities: =ep
eUID = 65534;  eGID = 65534;  capabilities: =ep
```

Now we switch to another terminal window—to a shell process running in another namespace (namely, the parent user namespace of the process running `demo_userns`) and create a user ID mapping for the child process in the new user namespace created by `demo_userns`:

```
$ ps -C demo_userns -o 'pid uid comm'      # Determine PID of clone child
  PID   UID COMMAND
 4712  1000 demo_userns                    # This is the parent
 4713  1000 demo_userns                    # Child in a new user namespace
```

```
$ echo '0 1000 1' > /proc/4713/uid_map
```

If we return to the window running `demo_userns`, we now see:

```
eUID = 0;  eGID = 65534;  capabilities: =ep
```

In other words, the user ID 1000 in the parent user namespace (which was formerly mapped to 65534) has been mapped to user ID 0 in the user namespace created by `demo_userns`. From this point, all operations within the new user namespace that deal with this user ID will see the number 0, while corresponding operations in the parent user namespace will see the same process as having user ID 1000.

We can likewise create a mapping for group IDs in the new user namespace. Switching to another terminal window, we create a mapping for the single group ID 1000 in the parent user namespace to the group ID 0 in the new user namespace:

```
$ echo '0 1000 1' > /proc/4713/gid_map
```

Switching back to the window running `demo_userns`, we see that change reflected in the display of the effective group ID:

```
eUID = 0;  eGID = 0;  capabilities: =ep
```

Rules for writing to mapping files

There are a number of rules governing writing to `uid_map` files; analogous rules apply for writing to `gid_map` files. The most important of these rules are as follows.

Defining a mapping is a one-time operation per namespace: we can perform only a single write (that may contain multiple newline-delimited records) to a `uid_map` file of exactly one of the processes in the user namespace. Furthermore, the number of lines that may be written to the file is currently limited to five (an arbitrary limit that may be increased in the future).

The `/proc/`*PID*`/uid_map` file is owned by the user ID that created the namespace, and is writeable only by that user (or a privileged user). In addition, all of the following requirements must be met:

- The writing process must have the `CAP_SETUID` (`CAP_SETGID` for `gid_map`) capability in the user namespace of the process *PID*.
- Regardless of capabilities, the writing process must be in either the user namespace of the process *PID* or inside the (immediate) parent user namespace of the process *PID*.
- One of the following must be true:
    - The data written to `uid_map` (`gid_map`) consists of a single line that maps (only) the writing process's effective user ID (group ID) in the parent user namespace to a user ID (group ID) in the user namespace. This rule allows the initial process in a user namespace (i.e., the child created by `clone()`) to write a mapping for its own user ID (group ID).
    - The process has the `CAP_SETUID` (`CAP_SETGID` for `gid_map`) capability in the parent user namespace. Such a process can define mappings to arbitrary user IDs (group IDs) in the parent user namespace. As we noted earlier, the initial process in a new user namespace has no capabilities in the parent namespace. Thus, only a process in the parent namespace can write a mapping that maps arbitrary IDs in the parent user namespace.

Capabilities, execve(), and user ID 0

In an earlier article in this series, we developed the ns_child_exec program. This program uses clone() to create a child process in new namespaces specified by command-line options and then executes a shell command in the child process.

Suppose that we use this program to execute a shell in a new user namespace and then within that shell we try to define the user ID mapping for the new user namespace. In doing so, we run into a problem:

```
$ ./ns_child_exec -U  bash
$ echo '0 1000 1' > /proc/$$/uid_map       # $$ is the PID of the shell
bash: echo: write error: Operation not permitted
```

This error occurs because the shell has no capabilities inside the new user namespace, as can be seen from the following commands:

```
$ id -u         # Verify that user ID and group ID are not mapped
65534
$ id -g
65534
$ cat /proc/$$/status | egrep 'Cap(Inh|Prm|Eff)'
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
```

The problem occurred at the execve() call that executed the bash shell: when a process with non-zero user IDs performs an execve(), the process's capability sets are cleared. (The capabilities(7) manual page details the treatment of capabilities during an execve().)

To avoid this problem, it is necessary to create a user ID mapping inside the user namespace *before* performing the execve(). This is not possible with the ns_child_exec program; we need a slightly enhanced version of the program that does allow this.

The userns_child_exec.c program performs the same task as the ns_child_exec program, and has the same command-line interface, except that it allows two additional command-line options, -M and -G. These options accept string arguments that are used to define user and group ID maps for the new user namespace. For example, the following command maps both user ID 1000 and group ID 1000 to 0 in the new user namespace:

```
$ ./userns_child_exec -U -M '0 1000 1' -G '0 1000 1' bash
```

This time, updating the mapping files succeeds, and we see that the shell has the expected user ID, group ID, and capabilities:

```
$ id -u
0
$ id -g
0
$ cat /proc/$$/status | egrep 'Cap(Inh|Prm|Eff)'
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
```

There are some subtleties to the implementation of the userns_child_exec program. First, either the parent process (i.e., the caller of clone()) or the new child process could update the user ID and group ID maps of the new user namespace. However, following the rules above, the only kind of mapping that the child process could define would be one that maps just its own effective user ID. If we want to define arbitrary user and group ID mappings in the child, then that must be done by the parent process. Furthermore, the parent process

must have suitable capabilities, namely `CAP_SETUID`, `CAP_SETGID`, and (to ensure that the parent has the permissions needed to open the mapping files) `CAP_DAC_OVERRIDE`.

Furthermore, the parent must ensure that it updates the mapping files before the child calls `execve()` (otherwise we have exactly the problem described above, where the child will lose capabilities during the `execve()`). To do this, the two processes employ a pipe to ensure the required synchronization; comments in the program source code give full details.

Viewing user and group ID mappings

The examples so far showed the use of `/proc/`*PID*`/uid_map` and `/proc/`*PID*`/gid_map` files for defining a mapping. These files can also be used to view the mappings governing a process. As when writing to these files, the second (`ID-outside-ns`) value is interpreted according to which process is opening the file. If the process opening the file is in the same user namespace as the process *PID*, then `ID-outside-ns` is defined with respect to the parent user namespace. If the process opening the file is in a different user namespace, then `ID-outside-ns` is defined with respect to the user namespace of the process opening the file.

We can illustrate this by creating a couple of user namespaces running shells, and examining the `uid_map` files of the processes in the namespaces. We begin by creating a new user namespace with a process running a shell:

```
$ id -u             # Display effective user ID
1000
$ ./userns_child_exec -U -M '0 1000 1' -G '0 1000 1' bash
$ echo $$           # Show shell's PID for later reference
2465
$ cat /proc/2465/uid_map
         0       1000          1
$ id -u             # Mapping gives this process an effective user ID of 0
0
```

Now suppose we switch to another terminal window and create a sibling user namespace that employs different user and group ID mappings:

```
$ ./userns_child_exec -U -M '200 1000 1' -G '200 1000 1' bash
$ cat /proc/self/uid_map
       200       1000          1
$ id -u             # Mapping gives this process an effective user ID of 200
200
$ echo $$           # Show shell's PID for later reference
2535
```

Continuing in the second terminal window, which is running in the second user namespace, we view the user ID mapping of the process in the other user namespace:

```
$ cat /proc/2465/uid_map
         0        200          1
```

The output of this command shows that user ID 0 in the other user namespace maps to user ID 200 in this namespace. Note that the same command produced different output when executed in the other user namespace, because the kernel generates the `ID-outside-ns` value according to the user namespace of the process that is reading from the file.

If we switch back to the first terminal window, and display the user ID mapping file for the process in the second user namespace, we see the converse mapping:

```
$ cat /proc/2535/uid_map
       200          0          1
```

Again, the output here is different from the same command when executed in the second user namespace, because the `ID-outside-ns` value is generated according to the user namespace of the process that is reading from the file. Of course, in the initial namespace, user ID 0 in the first namespace and user ID 200 in the second namespace both map to user ID 1000. We can verify this by executing the following commands in a third shell window inside the initial user namespace:

```
$ cat /proc/2465/uid_map
         0       1000          1
$ cat /proc/2535/uid_map
       200       1000          1
```

Concluding remarks

In this article, we've looked at the basics of user namespaces: creating a user namespace, using user and group ID map files, and the interaction of user namespaces and capabilities.

As we noted in an earlier article, one of the motivations for implementing user namespaces is to give non-root applications access to functionality that was formerly limited to the root user. In traditional UNIX systems, various pieces of functionality have been limited to the root user in order to prevent unprivileged users from manipulating the runtime environment of privileged programs, which could affect the operation of those programs in unexpected or undesirable ways.

A user namespace allows a process (that is unprivileged outside the namespace) to have root privileges while at the same time limiting the scope of that privilege to the namespace, with the result that the process cannot manipulate the runtime environment of privileged programs in the wider system. In order to use these root privileges meaningfully, we need to combine user namespaces with other types of namespaces—that topic will form the subject of the next article in this series.

Namespaces in operation, part 6: more on user namespaces

By **Michael Kerrisk**
March 6, 2013

In this article, we continue last week's discussion of user namespaces. In particular, we look in more detail at the interaction of user namespaces and capabilities as well as the combination of user namespaces with other types of namespaces. For the moment at least, this article will conclude our series on namespaces.

User namespaces and capabilities

Each process is associated with a particular user namespace. A process created by a call to `fork()` or a call to `clone()` without the `CLONE_NEWUSER` flag is placed in the same user namespace as its parent process. A process can change its user-namespace membership using `setns()`, if it has the `CAP_SYS_ADMIN` capability in the target namespace; in that case, it obtains a full set of capabilities upon entering the target namespace.

On the other hand, a `clone(CLONE_NEWUSER)` call creates a new user namespace and places the new child process in that namespace. This call also establishes a parental relationship between the two namespaces: each user namespace (other than the initial namespace) has a parent—the user namespace of the process that created it using `clone(CLONE_NEWUSER)`. A parental relationship between user namespaces is also established when a process calls `unshare(CLONE_NEWUSER)`. The difference is that `unshare()` places the caller in the new user namespace, and the parent of that namespace is the caller's previous user namespace. As we'll see in a moment, the parental relationship between user namespaces is important because it defines the capabilities that a process may have in a child namespace.

Each process also has three associated sets of capabilities: permitted, effective, and inheritable. The capabilities(7) manual page describes these three sets in some detail. In this article, it is mainly the effective capability set that is of interest to us. This set determines a process's ability to perform privileged operations.

User namespaces change the way in which (effective) capabilities are interpreted. First, having a capability inside a particular user namespace allows a process to perform operations only on resources governed by that namespace; we say more on this point below, when we talk about the interaction of user namespaces with other types of namespaces. In addition, whether or not a process has capabilities in a particular user namespace depends on its namespace membership and the parental relationship between user namespaces. The rules are as follows:

1. A process has a capability inside a user namespace if it is a member of the namespace and that capability is present in its effective capability set. A process may obtain capabilities in its effective set in a number of ways. The most common reasons are that it executed a program that conferred capabilities (a set-user-ID program or a program that has associated file capabilities) or it is the child of a call to `clone(CLONE_NEWUSER)`, which automatically obtains a full set of capabilities.

2. If a process has a capability in a user namespace, then it has that capability in all child (and further removed descendant) namespaces as well. Put another way: creating a new user namespace does not isolate the members of that namespace from the effects of privileged processes in a parent namespace.

3. When a user namespace is created, the kernel records the effective user ID of the creating process as being the "owner" of the namespace. A process whose effective user ID matches that of the owner of a user namespace and which is a member of the parent namespace has all capabilities in the namespace.

> By virtue of the previous rule, those capabilities propagate down into all descendant namespaces as well. This means that after creation of a new user namespace, other processes owned by the same user in the parent namespace have all capabilities in the new namespace.

We can demonstrate the third rule with the help of a small program, `userns_setns_test.c`. This program takes one command-line argument: the pathname of a `/proc/PID/ns/user` file that identifies a user namespace. It creates a child in a new user namespace and then both the parent (which remains in the same user namespace as the shell that was used to invoke the program) and the child attempt to join the namespace specified on the command line using `setns()`; as noted above, `setns()` requires that the caller have the `CAP_SYS_ADMIN` capability in the target namespace.

For our demonstration, we use this program in conjunction with the `userns_child_exec.c` program developed in the previous article in this series. First, we use that program to start a shell (we use `ksh`, simply to create a distinctively named process) running in a new user namespace:

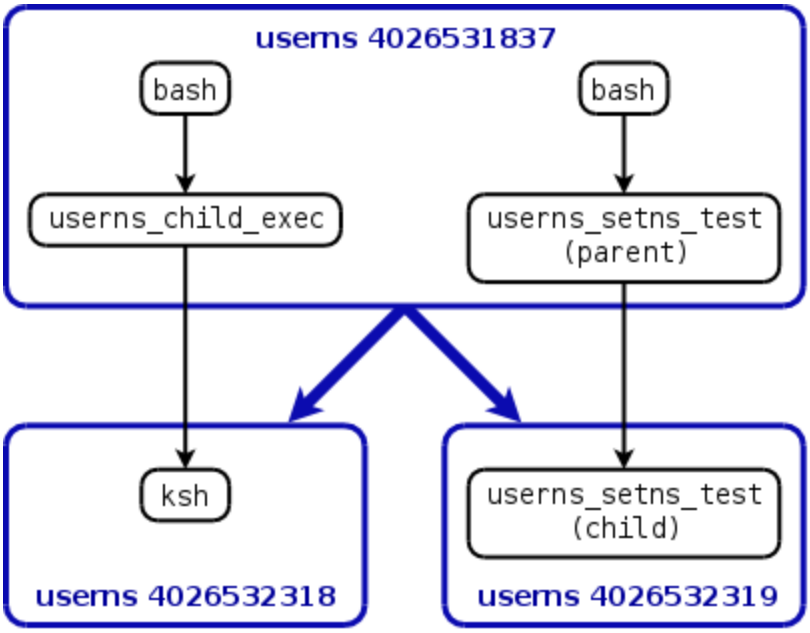
```
$ id -u
1000
$ readlink /proc/$$/ns/user       # Obtain ID for initial namespace
user:[4026531837]
$ ./userns_child_exec -U -M '0 1000 1' -G '0 1000 1' ksh
ksh$ echo $$                      # Obtain PID of shell
528
ksh$ readlink /proc/$$/ns/user    # This shell is in a new namespace
user:[4026532318]
```

Now, we switch to a separate terminal window, to a shell running in the initial namespace, and run our test program:

```
$ readlink /proc/$$/ns/user       # Verify that we are in parent namespace
user:[4026531837]
$ ./userns_setns_test /proc/528/ns/user
parent: readlink("/proc/self/ns/user") ==> user:[4026531837]
parent: setns() succeeded

child:  readlink("/proc/self/ns/user") ==> user:[4026532319]
child:  setns() failed: Operation not permitted
```

The following program shows the parental relationships between the various processes (black arrows) and namespaces (blue arrows) that have been created:



Looking at the output of the `readlink` commands at the start of each shell session, we can see that the parent process created when the `userns_setns_test` program was run is in the initial user namespace (4026531837). (As noted in an earlier article in this series, these numbers are i-node numbers for symbolic links in the `/proc/`*PID*`/ns` directory.) As such, by rule three above, since the parent process had the same effective user ID (1000) as the process that created the new user namespace (4026532318), it had all capabilities in that namespace, including `CAP_SYS_ADMIN`; thus the `setns()` call in the parent succeeds.

On the other hand, the child process created by `userns_setns_test` is in a different namespace (4026532319)—in effect, a sibling namespace of the namespace where the `ksh` process is running. As such, the second of the rules described above does not apply, because that namespace is not an ancestor of namespace 4026532318. Thus, the child process does not have the `CAP_SYS_ADMIN` capability in that namespace and the `setns()` call fails.

Combining user namespaces with other types of namespaces

Creating namespaces other than user namespaces requires the `CAP_SYS_ADMIN` capability. On the other hand, creating a user namespace requires (since Linux 3.8) no capabilities, and the first process in the namespace gains a full set of capabilities (in the new user namespace). This means that that process can now create any other type of namespace using a second call to `clone()`.

However, this two-step process is not necessary. It is also possible to include additional `CLONE_NEW*` flags in the *same* `clone()` (or `unshare()`) call that employs `CLONE_NEWUSER` to create the new user namespace. In this case, the kernel guarantees that the `CLONE_NEWUSER` flag is acted upon first, creating a new user namespace in which the to-be-created child has all capabilities. The kernel then acts on all of the remaining `CLONE_NEW*` flags, creating corresponding new namespaces and making the child a member of all of those namespaces.

Thus, for example, an unprivileged process can make a call of the following form to create a child process that is a member of both a new user namespace and a new UTS namespace:

```
clone(child_func, stackp, CLONE_NEWUSER | CLONE_NEWUTS, arg);
```

We can use our `userns_child_exec` program to perform a `clone()` call equivalent to the above and execute a shell in the child process. The following command specifies the creation of a new UTS namespace (`-u`), and a new user namespace (`-U`) in which both user and group ID 1000 are mapped to 0:

```
$ uname -n           # Display hostname for later reference
antero
$ ./userns_child_exec -u -U -M '0 1000 1' -G '0 1000 1' bash
```

As expected, the shell process has a full set of permitted and effective capabilities:

```
$ id -u              # Show effective user and group ID of shell
0
$ id -g
0
$ cat /proc/$$/status | egrep 'Cap(Inh|Prm|Eff)'
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
```

In the above output, the hexadecimal value 1fffffffff represents a capability set in which all 37 of the currently available Linux capabilities are enabled.

We can now go on to modify the hostname—one of the global resources isolated by UTS namespaces—using the standard `hostname` command; that operation requires the `CAP_SYS_ADMIN` capability. First, we set the hostname to a new value, and then we review that value with the `uname` command:

```
$ hostname bizarro     # Update hostname in this UTS namespace
$ uname -n             # Verify the change
bizarro
```

Switching to another terminal window—one that is running in the initial UTS namespace—we then check the hostname in that UTS namespace:

```
$ uname -n             # Hostname in original UTS namespace is unchanged
antero
```

From the above output, we can see that the change of hostname in the child UTS namespace is not visible in the parent UTS namespace.

Capabilities revisited

Although the kernel grants all capabilities to the initial process in a user namespace, this does not mean that process then has superuser privileges within the wider system. (It may, however, mean that unprivileged users now have access to exploits in kernel code that was formerly accessible only to root, as this mail on a vulnerability in tmpfs mounts notes.) When a new IPC, mount, network, PID, or UTS namespace is created via `clone()` or `unshare()`, the kernel records the user namespace of the creating process against the new namespace. Whenever a process operates on global resources governed by a namespace, permission checks are performed according to the process's capabilities in the user namespace that the kernel associated with the that namespace.

For example, suppose that we create a new user namespace using `clone(CLONE_NEWUSER)`. The resulting child process will have a full set of capabilities in the new user namespace, which means that it will, for example, be able to create other types of namespaces and be able to change its user and group IDs to other IDs that are mapped in the namespace. (In the previous article in this series, we saw that only a privileged process in the *parent* user namespace can create mappings to IDs other than the effective user and group ID of the process that created the namespace, so there is no security loophole here.)

On the other hand, the child process would not be able to mount a filesystem. The child process is still in the initial mount namespace, and in order to mount a filesystem in that namespace, it would need to have capabilities in the user namespace associated with that mount namespace (i.e., it would need capabilities in the

initial user namespace), which it does not have. Analogous statements apply for the global resources isolated by IPC, network, PID, and UTS namespaces.

Furthermore, the child process would not be able to perform privileged operations that require capabilities that are not (currently) governed by namespaces. Thus, for example, the child could not do things such as raising its hard resource limits, setting the system time, setting process priorities, or loading kernel modules~~, or rebooting the system~~. All of those operations require capabilities that sit outside the user namespace hierarchy; in effect, those operations require that the caller have capabilities in the initial user namespace.

By isolating the effect of capabilities to namespaces, user namespaces thus deliver on the promise of safely allowing unprivileged users access to functionality that was formerly limited to the root user. This in turn creates interesting possibilities for new kinds of user-space applications. For example, it now becomes possible for unprivileged users to run Linux containers without root privileges, to construct Chrome-style sandboxes without the use of set-user-ID-root helpers, to implement fakeroot-type applications without employing dynamic-linking tricks, and to implement `chroot()`-based applications for process isolation. Barring kernel bugs, applications that employ user namespaces to access privileged kernel functionality are more secure than traditional applications based on set-user-ID-root: with a user-namespace-based approach, even if an applications is compromised, it does not have any privileges that can be used to do damage in the wider system.

The author would like to thank Eric Biederman for answering many questions that came up as he experimented with namespaces during the course of writing this article series.…

Namespaces in operation, part 7: Network namespaces

Please consider subscribing to LWN

Subscriptions are the lifeblood of LWN.net. If you appreciate this content and would like to see more of it, your subscription will help to ensure that LWN continues to thrive. Please visit this page to join up and keep LWN on the net.

By **Jake Edge**
January 22, 2014

Namespaces in operation

It's been a while since last we looked at Linux namespaces. Our series has been missing a piece that we are finally filling in: network namespaces. As the name would imply, network namespaces partition the use of the network—devices, addresses, ports, routes, firewall rules, etc.—into separate boxes, essentially virtualizing the network within a single running kernel instance. Network namespaces entered the kernel in 2.6.24, almost exactly five years ago; it took something approaching a year before they were ready for prime time. Since then, they seem to have been largely ignored by many developers.

Basic network namespace management

As with the others, network namespaces are created by passing a flag to the `clone()` system call: `CLONE_NEWNET`. From the command line, though, it is convenient to use the `ip` networking configuration tool to set up and work with network namespaces. For example:

```
# ip netns add netns1
```

This command creates a new network namespace called `netns1`. When the `ip` tool creates a network namespace, it will create a bind mount for it under `/var/run/netns`; that allows the namespace to persist even when no processes are running within it and facilitates the manipulation of the namespace itself. Since network namespaces typically require a fair amount of configuration before they are ready for use, this feature will be appreciated by system administrators.

The "`ip netns exec`" command can be used to run network management commands within the namespace:

```
# ip netns exec netns1 ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

This command lists the interfaces visible inside the namespace. A network namespace can be removed with:

```
# ip netns delete netns1
```

This command removes the bind mount referring to the given network namespace. The namespace itself, however, will persist for as long as any processes are running within it.

Network namespace configuration

New network namespaces will have a loopback device but no other network devices. Aside from the loopback device, each network device (physical or virtual interfaces, bridges, etc.) can only be present in a single network namespace. In addition, physical devices (those connected to real hardware) cannot be assigned to namespaces other than the root. Instead, virtual network devices (e.g. virtual ethernet or veth) can be created and assigned to a namespace. These virtual devices allow processes inside the namespace to communicate over the network; it is the configuration, routing, and so on that determine who they can communicate with.

When first created, the `lo` loopback device in the new namespace is down, so even a loopback `ping` will fail:

```
# ip netns exec netns1 ping 127.0.0.1
connect: Network is unreachable
```

Bringing that interface up will allow pinging the loopback address:

```
# ip netns exec netns1 ip link set dev lo up
# ip netns exec netns1 ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.051 ms
...
```

But that still doesn't allow communication between netns1 and the root namespace. To do that, virtual ethernet devices need to be created and configured:

```
# ip link add veth0 type veth peer name veth1
# ip link set veth1 netns netns1
```

The first command sets up a pair of virtual ethernet devices that are connected. Packets sent to `veth0` will be received by `veth1` and vice versa. The second command assigns `veth1` to the netns1 namespace.

```
# ip netns exec netns1 ifconfig veth1 10.1.1.1/24 up
# ifconfig veth0 10.1.1.2/24 up
```

Then, these two commands set IP addresses for the two devices.

```
# ping 10.1.1.1
PING 10.1.1.1 (10.1.1.1) 56(84) bytes of data.
64 bytes from 10.1.1.1: icmp_seq=1 ttl=64 time=0.087 ms
...

# ip netns exec netns1 ping 10.1.1.2
PING 10.1.1.2 (10.1.1.2) 56(84) bytes of data.
64 bytes from 10.1.1.2: icmp_seq=1 ttl=64 time=0.054 ms
...
```

Communication in both directions is now possible as the `ping` commands above show.

As mentioned, though, namespaces do not share routing tables or firewall rules, as running `route` and `iptables -L` in netns1 will attest.

```
# ip netns exec netns1 route
# ip netns exec netns1 iptables -L
```

The first will simply show a route for packets to the 10.1.1 subnet (using `veth1`), while the second shows no iptables configured. All of that means that packets sent from netns1 to the internet at large will get the dreaded "Network is unreachable" message. There are several ways to connect the namespace to the internet if that is desired. A bridge can be created in the root namespace and the veth device from netns1. Alternatively, IP forwarding coupled with network address translation (NAT) could be configured in the root namespace. Either

of those (and there are other configuration possibilities) will allow packets from netns1 to reach the internet and for replies to be received in netns1.

Non-root processes that are assigned to a namespace (via `clone()`, `unshare()`, or `setns()`) only have access to the networking devices and configuration that have been set up in that namespace—root can add new devices and configure them, of course. Using the `ip netns` sub-command, there are two ways to address a network namespace: by its name, like netns1, or by the process ID of a process in that namespace. Since `init` generally lives in the root namespace, one could use a command like:

```
# ip link set vethX netns 1
```

That would put a (presumably newly created) veth device into the root namespace and it would work for a root user from any other namespace. In situations where it is not desirable to allow root to perform such operations from within a network namespace, the PID and mount namespace features can be used to make the other network namespaces unreachable.

Uses for network namespaces

As we have seen, a namespace's networking can range from none at all (or just loopback) to full access to the system's networking capabilities. That leads to a number of different use cases for network namespaces.

By essentially turning off the network inside a namespace, administrators can ensure that processes running there will be unable to make connections outside of the namespace. Even if a process is compromised through some kind of security vulnerability, it will be unable to perform actions like joining a botnet or sending spam.

Even processes that handle network traffic (a web server worker process or web browser rendering process for example) can be placed into a restricted namespace. Once a connection is established by or to the remote endpoint, the file descriptor for that connection could be handled by a child process that is placed in a new network namespace created by a `clone()` call. The child would inherit its parent's file descriptors, thus have access to the connected descriptor. Another possibility would be for the parent to send the connected file descriptor to a process in a restricted network namespace via a Unix socket. In either case, the lack of suitable network devices in the namespace would make it impossible for the child or worker process to make additional network connections.

Namespaces could also be used to test complicated or intricate networking configurations all on a single box. Running sensitive services in more locked-down, firewall-restricted namespace is another. Obviously, container implementations also use network namespaces to give each container its own view of the network, untrammeled by processes outside of the container. And so on.

Namespaces in general provide a way to partition system resources and to isolate groups of processes from each other's resources. Network namespaces are more of the same, but since networking is a sensitive area for security flaws, providing network isolation of various sorts is particularly valuable. Of course, using multiple namespace types together can provide even more isolation for both security and other needs.

Mount namespaces and shared subtrees

Mount namespaces are a powerful and flexible tool for creating per-user and per-container filesystem trees. They are also a surprisingly complex feature; in this continuation of our series on namespaces we unravel some of that complexity. In particular, we will take a close look at the shared subtrees feature, which allows mount and unmount events to be propagated between mount namespaces in an automatic, controlled fashion.

June 8, 2016
This article was contributed by Michael Kerrisk.

Namespaces in operation

Introduction

Mount namespaces were the first namespace type added to Linux, appearing in 2002 in Linux 2.4.19. They isolate the list of mount points seen by the processes in a namespace. Or, to put things another way, each mount namespace has its own list of mount points, meaning that processes in different namespaces see and are able to manipulate different views of the single directory hierarchy.

When the system is first booted, there is a single mount namespace, the so-called "initial namespace". New mount namespaces are created by using the `CLONE_NEWNS` flag with either the `clone()` system call (to create a new child process in the new namespace) or the `unshare()` system call (to move the caller into the new namespace). When a new mount namespace is created, it receives a copy of the mount point list replicated from the namespace of the caller of `clone()` or `unshare()`.

Following the `clone()` or `unshare()` call, mount points can be independently added and removed in each namespace (via `mount()` and `umount()`). Changes to the mount point list are (by default) visible only to processes in the mount namespace where the process resides; the changes are not visible in other mount namespaces.

Mount namespaces serve a variety of purposes. For example, they can be used to provide per-user views of the filesystem. Other uses include mounting a `/proc` filesystem for a new PID namespace without causing side effects for other process and `chroot()`-style isolation of a process to a portion of the single directory hierarchy. In some use cases, mount namespaces are combined with bind mounts.

Shared subtrees

Once the implementation of mount namespaces was completed, user-space programmers encountered a usability problem: mount namespaces provided *too much* isolation between namespaces. Suppose, for example, that a new disk is loaded into an optical disk drive. In the original implementation, the only way to make that disk visible in all mount namespaces was to mount the disk separately in each namespace. In many cases, it would instead be preferable to perform a single mount operation that makes the disk visible in all (or perhaps some subset) of the mount namespaces on the system.

Because of the problem just described, the shared subtrees feature was added in Linux 2.6.15 (in early 2006, around three years after the initial implementation of mount namespaces). The key benefit of shared subtrees is to allow automatic, controlled propagation of mount and unmount events between namespaces. This means, for example, that mounting an optical disk in one mount namespace can trigger a mount of that disk in all other namespaces.

Under the shared subtrees feature, each mount point is marked with a "propagation type", which determines whether mount points created and removed under this mount point are propagated to other mount points. There are four different propagation types:

- `MS_SHARED`: This mount point shares mount and unmount events with other mount points that are members of its "peer group" (which is described in more detail below). When a mount point is added or removed under this mount point, this change will propagate to the peer group, so that the mount or unmount will also take place under each of the peer mount points. Propagation also occurs in the reverse direction, so that mount and unmount events on a peer mount will also propagate to this mount point.

- `MS_PRIVATE`: This is the converse of a shared mount point. The mount point does not propagate events to any peers, and does not receive propagation events from any peers.

- `MS_SLAVE`: This propagation type sits midway between shared and private. A slave mount has a master—a shared peer group whose members propagate mount and unmount events to the slave mount. However, the slave mount does not propagate events to the master peer group.

- `MS_UNBINDABLE`: This mount point is unbindable. Like a private mount point, this mount point does not propagate events to or from peers. In addition, this mount point can't be the source for a bind mount operation.

It's worth expanding on a few points that were glossed over above. The first is that the propagation type is a per-mount-point setting. Within a namespace, some mount points might be marked shared, while others are marked private (or slave or unbindable).

The second point to emphasize is that the propagation type determines the propagation of mount and unmount events *immediately under* the mount point. Thus, if, under a shared mount, X, we create a child mount, Y, that child mount will propagate to other mount points in the peer group. However, the propagation type of X would have no effect for mount points created and removed under Y; whether or not events under Y are propagated would depend on the propagation type that is defined for Y. Analogously, whether an unmount event would be propagated when X itself is unmounted would depend on the propagation type of the parent mount of X.

In passing, it is perhaps worth clarifying that the word "event" is used here as an abstract term, in the sense of "something happened". The notion of event propagation does not imply some sort of message passing between mount points. Rather, it carries the idea that some mount or unmount operation on one mount point triggered a matching operation one or more other mount points.

Finally, it is possible for a mount to be both the slave of a master peer group as well as sharing events with a set of peers of its own—a so-called slave-and-shared mount. In this case, the mount might receive propagation events from the master, and those events would then be propagated to its peers.

Peer groups

A peer group is a set of mount points that propagate mount and unmount events to one another. A peer group acquires new members when a mount point whose propagation type is shared is either replicated during the creation of a new namespace or is used as the source for a bind mount. (For a bind mount, the details are more complex than we describe here; details can be found in the kernel source file `Documentation/filesystems/sharedsubtree.txt`.) In both cases, the new mount point is made a member of the same peer group as the existing mount point. Conversely, a mount point ceases to be a member of a peer group when it is unmounted, either explicitly, or implicitly when a mount namespace is torn down because the last member process terminates or moves to another namespace.

For example, suppose that in a shell running in the initial mount namespace, we make the root mount point private and create two shared mount points:

```
sh1# mount --make-private /
sh1# mount --make-shared /dev/sda3 /X
sh1# mount --make-shared /dev/sda5 /Y
```

As indicated by the "#" in the shell prompts, privilege is required for the various `mount` commands that we employ in the example shell sessions to create mount points and change their propagation types.

Then, on a second terminal, we use the `unshare` command to create a new mount namespace where we run a shell:

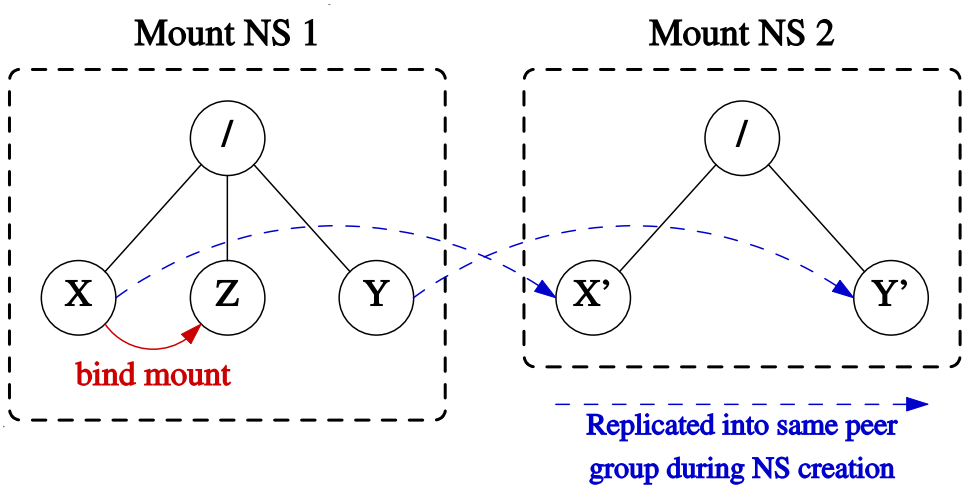
```
sh2# unshare -m --propagation unchanged sh
```

(The `-m` option creates a new mount namespace; the purpose of the `--propagation unchanged` option is explained later.)

Returning to the first terminal, we then create a bind mount from the `/X` mount point:

```
sh1# mkdir /Z
sh1# mount --bind /X /Z
```

Following these steps, we have the situation shown in the diagram below.



In this scenario, there are two peer groups:

- The first peer group contains the mount points X, X' (the duplicate of mount point X that was created when the second namespace was created), and Z (the bind mount created from the source mount point X in the initial namespace).
- The second peer group contains the mount points Y and Y' (the duplicate of mount point Y that was created when the second namespace was created).

Note that the bind mount Z, which was created in the initial namespace *after* the second namespace was created, was not replicated in the second namespace because the parent mount (`/`) was marked private.

Examining propagation types and peer groups via `/proc/PID/mountinfo`

The `/proc/PID/mountinfo` file (documented in the `proc(5)` manual page) displays a range of information

about the mount points for the mount namespace in which the process `PID` resides. All processes that reside in the same mount namespace will see the same view in this file. This file was designed to provide more information about mount points than was possible with the older, non-extensible `/proc/PID/mounts` file. Included in each record in this file is a (possibly empty) set of so-called "optional fields", which display information about the propagation type and peer group (for shared mounts) of each mount.

For a shared mount, the optional fields in the corresponding record in `/proc/PID/mountinfo` will contain a tag of the form `shared:N`. Here, the `shared` tag indicates that the mount is sharing propagation events with a peer group. The peer group is identified by `N`, an integer value that uniquely identifies the peer group. These IDs are numbered starting at 1, and may be recycled when a peer group ceases to exist because all of its members departed the group. All mount points that are members of the same peer group will show a `shared:N` tag with the same `N` in the `/proc/PID/mountinfo` file.

Thus for example, if we list the contents of `/proc/self/mountinfo` in the first of the shells discussed in the example above, we see the following (with a little bit of `sed` filtering to trim some irrelevant information from the output):

```
sh1# cat /proc/self/mountinfo | sed 's/ - .*//'
61 0 8:2 / / rw,relatime
81 61 8:3 / /X rw,relatime shared:1
124 61 8:5 / /Y rw,relatime shared:2
228 61 8:3 / /Z rw,relatime shared:1
```

From this output, we first see that the root mount point is private. This is indicated by the absence of any tags in the optional fields. We also see that the mount points `/X` and `/Z` are shared mount points in the same peer group (with ID 1), which means that mount and unmount events under either of these two mounts will propagate to the other. The mount `/Y` is a shared mount in a different peer group (ID 2), which, by definition, does not propagate events to or from the mounts in peer group 1.

The `/proc/PID/mountinfo` file also enables us to see the parental relationship between mount points. The first field in each record is a unique ID for each mount point. The second field is the ID for the parent mount. From the above output, we can see that the mount points `/X`, `/Y`, and `/Z` are all children of the root mount because their parent IDs are all 61.

Running the same command in the second shell (in the second namespace), we see:

```
sh2# cat /proc/self/mountinfo | sed 's/ - .*//'
147 146 8:2 / / rw,relatime
221 147 8:3 / /X rw,relatime shared:1
224 147 8:5 / /Y rw,relatime shared:2
```

Again, we see that the root mount point is private. Then we see that `/X` is a shared mount in peer group 1, the same peer group as the mounts `/X` and `/Z` in the initial mount namespace. Finally, we see that `/Y` is a shared mount in peer group 2, the same peer group as the mount `/Y` in the initial mount namespace. One final point to note is that the mount points that were replicated in the second namespace have their own unique IDs that differ from the IDs of the corresponding mounts in the initial namespace.

Debating defaults

Because the situation is a little complex, we have so far avoided discussing what the default propagation type is for a new mount point. From the kernel's perspective, the default when a new device mount is created is as follows:

- If the mount point has a parent (i.e., it is a non-root mount point) and the propagation type of the parent

is MS_SHARED, then the propagation type of the new mount is also MS_SHARED.

- Otherwise, the propagation type of the new mount is MS_PRIVATE.

According to these rules, the root mount would be MS_PRIVATE, and all descendant mounts would by default also be MS_PRIVATE. However, MS_SHARED would arguably have been a better default, since it is the more commonly employed propagation type. For that reason, systemd sets the propagation type of all mount points to MS_SHARED. Thus, on most modern Linux distributions, the default propagation type is effectively MS_SHARED. This is not the final word on the subject, however, since the util-linux unshare utility also has something to say. When creating a new mount namespace, unshare assumes that the user wants a fully isolated namespace, and makes all mount points private by performing the equivalent of the following command (which recursively marks all mounts under the root directory as private):

```
mount --make-rprivate /
```

To prevent this, we can use an additional option when creating the new namespace:

```
unshare -m --propagation unchanged <cmd>
```

Concluding remarks

In this article, we introduced the "theory" of mount namespaces and shared subtrees. We now have enough information to demonstrate and understand the semantics of the various propagation types; that will be the subject of a follow-on article.

Mount namespaces, mount propagation, and unbindable mounts

In the previous installment of our article series on namespaces, we looked at the key concepts underlying mount namespaces and shared subtrees, including the notions of mount propagation types and peer groups. In this article, we provide some practical demonstrations of the operation of the various propagation types: `MS_SHARED`, `MS_PRIVATE`, `MS_SLAVE`, and `MS_UNBINDABLE`.

June 15, 2016
This article was contributed by Michael Kerrisk.

Namespaces in operation

`MS_SHARED` and `MS_PRIVATE` example

As we saw in the previous article, the `MS_SHARED` and `MS_PRIVATE` propagation types are roughly opposites. A shared mount point is a member of peer group. Each of the member mount points in a peer group propagates mount and unmount events to the other members of the group. By contrast, a private mount point is not a member of a peer group; it neither propagates events to peers, nor receives events propagated from peers. In the following shell session, we demonstrate the different semantics of these two propagation types.

Suppose that, in the initial mount namespace, we have two existing mount points, `/mntS` and `/mntP`. From a shell in the namespace, we then mark `/mntS` as shared and `/mntP` as private, and view the mounts in `/proc/self/mountinfo`:

```
sh1# mount --make-shared /mntS
sh1# mount --make-private /mntP
sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
77 61 8:17 / /mntS rw,relatime shared:1
83 61 8:15 / /mntP rw,relatime
```

From the output, we see that `/mntS` is a shared mount in peer group 1, and that `/mntP` has no optional tags, indicating that it is a private mount. (As noted in the previous article, most mount and unmount operations require that the user is privileged, as indicated by the '#' prompt.)

On a second terminal, we create a new mount namespace where we run a second shell and inspect the mounts:

```
sh2# unshare -m --propagation unchanged sh
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
222 145 8:17 / /mntS rw,relatime shared:1
225 145 8:15 / /mntP rw,relatime
```

The new mount namespace received a copy of the initial mount namespace's mount points. These new mount points maintain the same propagation types, but have unique mount IDs (first field in the records).

In the second terminal, we then create mounts under each of `/mntS` and `/mntP` and inspect the outcome:

```
sh2# mkdir /mntS/a
sh2# mount /dev/sdb6 /mntS/a
sh2# mkdir /mntP/b
sh2# mount /dev/sdb7 /mntP/b
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
222 145 8:17 / /mntS rw,relatime shared:1
225 145 8:15 / /mntP rw,relatime
178 222 8:22 / /mntS/a rw,relatime shared:2
```

```
230 225 8:23 / /mntP/b rw,relatime
```

From the above, it can be seen that `/mntS/a` was created as shared (inheriting this setting from its parent mount) and `/mntP/b` was created as a private mount.

Returning to the first terminal and inspecting the set-up, we see that the new mount created under the shared mount point `/mntS` propagated to its peer mount (in the initial mount namespace), but the new mount created under the private mount point `/mntP` did not propagate:

```
sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
77 61 8:17 / /mntS rw,relatime shared:1
83 61 8:15 / /mntP rw,relatime
179 77 8:22 / /mntS/a rw,relatime shared:2
```

`MS_SLAVE` example

Making a mount point a slave allows it to receive propagated mount and unmount events from a master peer group, while preventing it from propagating events to that master. This is useful if we want to (say) receive a mount event when an optical disk is mounted in the master peer group (in another mount namespace), but we want to prevent mount and unmount events under the slave mount from having side effects in other namespaces.

We can demonstrate the effect of slaving by first marking two (existing) mount points in the initial mount namespace as shared:

```
sh1# mount --make-shared /mntX
sh1# mount --make-shared /mntY
sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
132 83 8:23 / /mntX rw,relatime shared:1
133 83 8:22 / /mntY rw,relatime shared:2
```

On a second terminal, we create a new mount namespace and inspect the replicated mount points:

```
sh2# unshare -m --propagation unchanged sh
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
168 167 8:23 / /mntX rw,relatime shared:1
169 167 8:22 / /mntY rw,relatime shared:2
```

In the new mount namespace, we then mark one of the mount points as a slave. The effect of changing a shared mount to a slave mount is to make it a slave of the peer group of which it was formerly a member.

```
sh2# mount --make-slave /mntY
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
168 167 8:23 / /mntX rw,relatime shared:1
169 167 8:22 / /mntY rw,relatime master:2
```

In the above output, the `/mntY` mount point is marked with the tag `master:2`. The tag name is perhaps counterintuitive: it indicates that the mount point is a *slave* mount that is receiving propagation events from the master peer group with the ID 2. In the case where a mount is both a slave of another peer group, and shares events with a peer group of its own, then the optional fields in the `/proc/PID/mountinfo` record will show both a `master:M` tag and a `shared:N` tag.

Continuing in the new namespace, we create mounts under each of `/mntX` and `/mntY`:

```
sh2# mkdir /mntX/a
sh2# mount /dev/sda3 /mntX/a
sh2# mkdir /mntY/b
```

```
sh2# mount /dev/sda5 /mntY/b
```

When we inspect the state of the mount points in the new mount namespace, we see that `/mntX/a` was created as a new shared mount (inheriting the "shared" setting from its parent mount) and `/mntY/b` was created as a private mount (i.e., no tags shown in the optional fields):

```
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
168 167 8:23 / /mntX rw,relatime shared:1
169 167 8:22 / /mntY rw,relatime master:2
173 168 8:3 / /mntX/a rw,relatime shared:3
175 169 8:5 / /mntY/b rw,relatime
```

Returning to the first terminal, we see that the mount `/mntX/a` propagated to the `/mntX` peer in the initial namespace, but the mount `/mntY/b` did not propagate:

```
sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
132 83 8:23 / /mntX rw,relatime shared:1
133 83 8:22 / /mntY rw,relatime shared:2
174 132 8:3 / /mntX/a rw,relatime shared:3
```

Next, we create a new mount point under `/mntY` in the initial mount namespace:

```
sh1# mkdir /mntY/c
sh1# mount /dev/sda1 /mntY/c
sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
132 83 8:23 / /mntX rw,relatime shared:1
133 83 8:22 / /mntY rw,relatime shared:2
174 132 8:3 / /mntX/a rw,relatime shared:3
178 133 8:1 / /mntY/c rw,relatime shared:4
```

When we examine the mount points in the second mount namespace, we see that in this case the new mount has been propagated to the slave mount point, and that the new mount is itself a slave mount (to peer group 4):

```
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
168 167 8:23 / /mntX rw,relatime shared:1
169 167 8:22 / /mntY rw,relatime master:2
173 168 8:3 / /mntX/a rw,relatime shared:3
175 169 8:5 / /mntY/b rw,relatime
179 169 8:1 / /mntY/c rw,relatime master:4
```

An aside: bind mounts

In a moment, we'll look at the use of the `MS_UNBINDABLE` propagation type. However, beforehand, it's useful to briefly describe the concept of a bind mount, a feature that first appeared in Linux 2.4.

A bind mount can be used to make a file or directory subtree visible at another location in the single directory hierarchy. In some ways, a bind mount is like a hard link, but it differs in some important respects:

- It is not possible to create a hard link to a directory, but it is possible to bind mount to a directory.
- Hard links can be made only to files on the same filesystem, while a bind mount can cross filesystem boundaries (and even reach out of a `chroot()` jail).
- Hard links entail a modification to the filesystem. By contrast, a bind mount is a record in the mount list of a mount namespace—in other words, a property of the live system.

A bind mount can be created programmatically using the `mount()` `MS_BIND` flag or on the command line using

`mount --bind`. In the following example, we first create a directory containing a file and then bind mount that directory at a new location:

```
# mkdir dir1                  # Create source directory
# touch dir1/x                # Populate the directory
# mkdir dir2                  # Create target for bind mount
# mount --bind dir1 dir2      # Create bind mount
# ls dir2                     # Bind mount has same content
x
```

Then we create a file under the new mount point and observe that the new file is visible under the original directory as well, indicating that the bind mount refers to the same directory object:

```
# touch dir2/y
# ls dir1
x  y
```

By default, when creating a bind mount of a directory, only that directory is mounted at the new location; if there are any mounts under that directory tree, they are not replicated under the mount target. It is also possible to perform a *recursive bind mount*, by calling `mount()` with the flags `MS_BIND` and `MS_REC`, or from the command line using the `mount --rbind` option. In this case, each mount under the source tree is replicated at the corresponding location in the target tree.

`MS_UNBINDABLE` example

The shared, private, and slave propagation types are about managing propagation of mount events between peer mounts (which are typically in different namespaces). Unbindable mounts exist to solve a different problem, one that preceded the existence of mount namespaces. That problem is the so-called "mount point explosion" that occurs when repeatedly performing recursive bind mounts of a higher-level subtree at a lower-level mount point. We'll now walk through a shell session that demonstrates the problem, and then see how unbindable mounts provide a solution.

To begin with, suppose we have a system with the two mount points, as follows:

```
# mount | awk '{print $1, $2, $3}'
/dev/sda1 on /
/dev/sdb6 on /mntX
```

Now suppose that we want to recursively bind mount the root directory under several users' home directories. We'll do this for the first user and inspect the mount points. However, we first create a new namespace in which we recursively mark all mount points as slaves, to prevent the steps that we perform from having any side effects in other mount namespaces:

```
# unshare -m sh
# mount --make-rslave /
# mount --rbind / /home/cecilia
# mount | awk '{print $1, $2, $3}'
/dev/sda1 on /
/dev/sdb6 on /mntX
/dev/sda1 on /home/cecilia
/dev/sdb6 on /home/cecilia/mntX
```

When we repeat the recursive bind operation for the second user, we start to see the explosion problem:

```
# mount --rbind / /home/henry
# mount | awk '{print $1, $2, $3}'
/dev/sda1 on /
```

```
/dev/sdb6 on /mntX
/dev/sda1 on /home/cecilia
/dev/sdb6 on /home/cecilia/mntX
/dev/sda1 on /home/henry
/dev/sdb6 on /home/henry/mntX
/dev/sda1 on /home/henry/home/cecilia
/dev/sdb6 on /home/henry/home/cecilia/mntX
```

Under `/home/henry`, we have not only recursively added the `/mntX` mount, but also the recursive mount of that directory under `/home/cecilia` that was created in the previous step. Upon repeating the step for a third user and simply counting the resulting mounts, it becomes obvious that the explosion is exponential in nature:

```
# mount --rbind / /home/otto
# mount | awk '{print $1, $2, $3}' | wc -l
16
```

We can avoid this mount explosion problem by making each of the new mounts unbindable. The effect of doing this is that recursive bind mounts of the root directory will not replicate the unbindable mounts. Returning to the original scenario, we make an unbindable mount for the first user and examine the mount via `/proc/self/mountinfo`:

```
# mount --rbind --make-unbindable / /home/cecilia
# cat /proc/self/mountinfo | grep /home/cecilia | sed 's/ - .*//'
108 83 8:2 / /home/cecilia rw,relatime unbindable
...
```

An unbindable mount is shown with the tag `unbindable` in the optional fields of the `/proc/self/mountinfo` record.

Now we create unbindable recursive bind mounts for the other two users:

```
# mount --rbind --make-unbindable / /home/henry
# mount --rbind --make-unbindable / /home/otto
```

Upon examining the list of mount points, we see that there has been no explosion of mount points, because the unbindable mounts were not replicated under each user's directory:

```
# mount | awk '{print $1, $2, $3}'
/dev/sda1 on /
/dev/sdb6 on /mntX
/dev/sda1 on /home/cecilia
/dev/sdb6 on /home/cecilia/mntX
/dev/sda1 on /home/henry
/dev/sdb6 on /home/henry/mntX
/dev/sda1 on /home/otto
/dev/sdb6 on /home/otto/mntX
```

Concluding remarks

Mount namespaces, in conjunction with the shared subtrees feature, are a powerful and flexible tool for creating per-user and per-container filesystem trees. They are also a surprisingly complex feature, and we have tried to unravel some of that complexity in this article. However, there are actually several more topics that we haven't considered. For example, there are detailed rules that describe the propagation type that results when performing bind mounts and move (`mount --move`) operations, as well as rules that describe the result when changing the propagation type of a mount. Many of those details can be found in the kernel source file `Documentation/filesystems/sharedsubtree.txt`.