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

$ man user_namespaces.7

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

NAME

user_namespaces - overview of Linux user namespaces

DESCRIPTION

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

User namespaces isolate security-related identifiers and attributes, in particular, user

IDs and group IDs (see credentials(7)), the root directory, keys (see keyrings(7)), and

capabilities (see capabilities(7)). A process's user and group IDs can be different in?

side and outside a user namespace. In particular, a process can have a normal unprivi?

leged user ID outside a user namespace while at the same time having a user ID of 0 inside

the namespace; in other words, the process has full privileges for operations inside the

user namespace, but is unprivileged for operations outside the namespace.

Nested namespaces, namespace membership

User namespaces can be nested; that is, each user namespace?except the initial ("root")

namespace?has a parent user namespace, and can have zero or more child user namespaces.

The parent user namespace is the user namespace of the process that creates the user name?

space via a call to unshare(2) or clone(2) with the CLONE_NEWUSER flag.

The kernel imposes (since version 3.11) a limit of 32 nested levels of user namespaces.

Calls to unshare(2) or clone(2) that would cause this limit to be exceeded fail with the

error EUSERS.

Each process is a member of exactly one user namespace. A process created via fork(2) or

clone(2) without the CLONE_NEWUSER flag is a member of the same user namespace as its par?

ent. A single-threaded process can join another user namespace with setns(2) if it has the CAP_SYS_ADMIN in that namespace; upon doing so, it gains a full set of capabilities in that namespace.

A call to clone(2) or unshare(2) with the CLONE_NEWUSER flag makes the new child process (for clone(2)) or the caller (for unshare(2)) a member of the new user namespace created by the call.

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

Capabilities

The child process created by clone(2) with the CLONE_NEWUSER flag starts out with a com? plete set of capabilities in the new user namespace. Likewise, a process that creates a new user namespace using unshare(2) or joins an existing user namespace using setns(2) gains a full set of capabilities in that namespace. On the other hand, that process has no capabilities in the parent (in the case of clone(2)) or previous (in the case of un? share(2) and setns(2)) user namespace, even if the new namespace is created or joined by the root user (i.e., a process with user ID 0 in the root namespace).

Note that a call to execve(2) will cause a process's capabilities to be recalculated in the usual way (see capabilities(7)). Consequently, unless the process has a user ID of 0 within the namespace, or the executable file has a nonempty inheritable capabilities mask, the process will lose all capabilities. See the discussion of user and group ID mappings, below.

A call to clone(2) or unshare(2) using the CLONE_NEWUSER flag or a call to setns(2) that moves the caller into another user namespace sets the "securebits" flags (see capabili? ties(7)) to their default values (all flags disabled) in the child (for clone(2)) or caller (for unshare(2) or setns(2)). Note that because the caller no longer has capabili? ties in its original user namespace after a call to setns(2), it is not possible for a process to reset its "securebits" flags while retaining its user namespace membership by using a pair of setns(2) calls to move to another user namespace and then return to its original user namespace.

The rules for determining whether or not a process has a capability in a particular user namespace are as follows:

1. A process has a capability inside a user namespace if it is a member of that namespace and it has the capability in its effective capability set. A process can gain capabil? ities in its effective capability set in various ways. For example, it may execute a

set-user-ID program or an executable with associated file capabilities. In addition, a process may gain capabilities via the effect of clone(2), unshare(2), or setns(2), as already described.

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.

3. When a user namespace is created, the kernel records the effective user ID of the cre? ating process as being the "owner" of the namespace. A process that resides in the parent of the user namespace and whose effective user ID matches the owner of the name? space has all capabilities in the namespace. By virtue of the previous rule, this means that the process has all capabilities in all further removed descendant user namespaces as well. The NS_GET_OWNER_UID ioctl(2) operation can be used to discover the user ID of the owner of the namespace; see ioctl_ns(2).

Effect of capabilities within a user namespace

Having a capability inside a user namespace permits a process to perform operations (that require privilege) only on resources governed by that namespace. In other words, having a capability in a user namespace permits a process to perform privileged operations on re? sources that are governed by (nonuser) namespaces owned by (associated with) the user namespace (see the next subsection).

On the other hand, there are many privileged operations that affect resources that are not associated with any namespace type, for example, changing the system (i.e., calendar) time (governed by CAP_SYS_TIME), loading a kernel module (governed by CAP_SYS_MODULE), and cre? ating a device (governed by CAP_MKNOD). Only a process with privileges in the initial user namespace can perform such operations.

Holding CAP_SYS_ADMIN within the user namespace that owns a process's mount namespace al? lows that process to create bind mounts and mount the following types of filesystems:

* /proc (since Linux 3.8)
* /sys (since Linux 3.8)
* devpts (since Linux 3.9)
* tmpfs(5) (since Linux 3.9)
* ramfs (since Linux 3.9)
* mqueue (since Linux 3.9)
* bpf (since Linux 4.4)

Holding CAP_SYS_ADMIN within the user namespace that owns a process's cgroup namespace al?

lows (since Linux 4.6) that process to the mount the cgroup version 2 filesystem and cgroup version 1 named hierarchies (i.e., cgroup filesystems mounted with the "none,name=" option).

Holding CAP_SYS_ADMIN within the user namespace that owns a process's PID namespace allows (since Linux 3.8) that process to mount /proc filesystems.

Note however, that mounting block-based filesystems can be done only by a process that holds CAP_SYS_ADMIN in the initial user namespace.

Interaction of user namespaces and other types of namespaces

Starting in Linux 3.8, unprivileged processes can create user namespaces, and the other types of namespaces can be created with just the CAP_SYS_ADMIN capability in the caller's user namespace.

When a nonuser namespace is created, it is owned by the user namespace in which the creat? ing process was a member at the time of the creation of the namespace. Privileged opera? tions on resources governed by the nonuser namespace require that the process has the nec? essary capabilities in the user namespace that owns the nonuser namespace.

If CLONE_NEWUSER is specified along with other CLONE_NEW* flags in a single clone(2) or unshare(2) call, the user namespace is guaranteed to be created first, giving the child (clone(2)) or caller (unshare(2)) privileges over the remaining namespaces created by the call. Thus, it is possible for an unprivileged caller to specify this combination of flags.

When a new namespace (other than a user namespace) is created via clone(2) or unshare(2), the kernel records the user namespace of the creating process as the owner of the new namespace. (This association can't be changed.) When a process in the new namespace sub? sequently performs privileged operations that operate on global resources isolated by the namespace, the permission checks are performed according to the process's capabilities in the user namespace that the kernel associated with the new namespace. For example, sup? pose that a process attempts to change the hostname (sethostname(2)), a resource governed by the UTS namespace. In this case, the kernel will determine which user namespace owns the process's UTS namespace, and check whether the process has the required capability (CAP_SYS_ADMIN) in that user namespace.

The NS_GET_USERNS ioctl(2) operation can be used to discover the user namespace that owns a nonuser namespace; see ioctl_ns(2).

User and group ID mappings: uid_map and gid_map

When a user namespace is created, it starts out without a mapping of user IDs (group IDs) to the parent user namespace. The /proc/[pid]/uid_map and /proc/[pid]/gid_map files (available since Linux 3.5) expose the mappings for user and group IDs inside the user namespace for the process pid. These files can be read to view the mappings in a user namespace and written to (once) to define the mappings.

The description in the following paragraphs explains the details for uid_map; gid_map is exactly the same, but each instance of "user ID" is replaced by "group ID".

The uid_map file exposes the mapping of user IDs from the user namespace of the process pid to the user namespace of the process that opened uid_map (but see a qualification to this point below). In other words, processes that are in different user namespaces will potentially see different values when reading from a particular uid_map file, depending on the user ID mappings for the user namespaces of the reading processes.

Each line in the uid_map file specifies a 1-to-1 mapping of a range of contiguous user IDs between two user namespaces. (When a user namespace is first created, this file is empty.) The specification in each line takes the form of three numbers delimited by white space. The first two numbers specify the starting user ID in each of the two user name? spaces. The third number specifies the length of the mapped range. In detail, the fields are interpreted as follows:

(1) The start of the range of user IDs in the user namespace of the process pid.

(2) The start of the range of user IDs to which the user IDs specified by field one map.

How field two is interpreted depends on whether the process that opened uid_map and the process pid are in the same user namespace, as follows:

a) If the two processes are in different user namespaces: field two is the start of a range of user IDs in the user namespace of the process that opened uid_map.

b) If the two processes are in the same user namespace: field two is the start of the range of user IDs in the parent user namespace of the process pid. This case en? ables the opener of uid_map (the common case here is opening /proc/self/uid_map) to see the mapping of user IDs into the user namespace of the process that created this user namespace.

(3) The length of the range of user IDs that is mapped between the two user namespaces.

System calls that return user IDs (group IDs)?for example, getuid(2), getgid(2), and the credential fields in the structure returned by stat(2)?return the user ID (group ID) mapped into the caller's user namespace.

When a process accesses a file, its user and group IDs are mapped into the initial user namespace for the purpose of permission checking and assigning IDs when creating a file. When a process retrieves file user and group IDs via stat(2), the IDs are mapped in the opposite direction, to produce values relative to the process user and group ID mappings.

The initial user namespace has no parent namespace, but, for consistency, the kernel pro? vides dummy user and group ID mapping files for this namespace. Looking at the uid_map file (gid_map is the same) from a shell in the initial namespace shows:

```
$ cat /proc/$$/uid_map
         0          0 4294967295
```

This mapping tells us that the range starting at user ID 0 in this namespace maps to a range starting at 0 in the (nonexistent) parent namespace, and the length of the range is the largest 32-bit unsigned integer. This leaves 4294967295 (the 32-bit signed -1 value) unmapped. This is deliberate: (uid_t) -1 is used in several interfaces (e.g., se? treuid(2)) as a way to specify "no user ID". Leaving (uid_t) -1 unmapped and unusable guarantees that there will be no confusion when using these interfaces.

Defining user and group ID mappings: writing to uid_map and gid_map

After the creation of a new user namespace, the uid_map file of one of the processes in the namespace may be written to once to define the mapping of user IDs in the new user namespace. An attempt to write more than once to a uid_map file in a user namespace fails with the error EPERM. Similar rules apply for gid_map files.

The lines written to uid_map (gid_map) must conform to the following rules:

* The three fields must be valid numbers, and the last field must be greater than 0.
* Lines are terminated by newline characters.
* There is a limit on the number of lines in the file. In Linux 4.14 and earlier, this limit was (arbitrarily) set at 5 lines. Since Linux 4.15, the limit is 340 lines. In addition, the number of bytes written to the file must be less than the system page size, and the write must be performed at the start of the file (i.e., lseek(2) and pwrite(2) can't be used to write to nonzero offsets in the file).
* The range of user IDs (group IDs) specified in each line cannot overlap with the ranges in any other lines. In the initial implementation (Linux 3.8), this requirement was satisfied by a simplistic implementation that imposed the further requirement that the values in both field 1 and field 2 of successive lines must be in ascending numerical order, which prevented some otherwise valid maps from being created. Linux 3.9 and

later fix this limitation, allowing any valid set of nonoverlapping maps.

* At least one line must be written to the file.

Writes that violate the above rules fail with the error EINVAL.

In order for a process to write to the /proc/[pid]/uid_map (/proc/[pid]/gid_map) file, all of the following requirements must be met:

1. The writing process must have the CAP_SETUID (CAP_SETGID) capability in the user name?space of the process pid.

2. The writing process must either be in the user namespace of the process pid or be in the parent user namespace of the process pid.

3. The mapped user IDs (group IDs) must in turn have a mapping in the parent user name?space.

4. One of the following two cases applies:

   * Either the writing process has the CAP_SETUID (CAP_SETGID) capability in the parent user namespace.

     + No further restrictions apply: the process can make mappings to arbitrary user IDs (group IDs) in the parent user namespace.

   * Or otherwise all of the following restrictions apply:

     + The data written to uid_map (gid_map) must consist of a single line that maps the writing process's effective user ID (group ID) in the parent user namespace to a user ID (group ID) in the user namespace.

     + The writing process must have the same effective user ID as the process that cre?ated the user namespace.

     + In the case of gid_map, use of the setgroups(2) system call must first be denied by writing "deny" to the /proc/[pid]/setgroups file (see below) before writing to gid_map.

Writes that violate the above rules fail with the error EPERM.

Interaction with system calls that change process UIDs or GIDs

In a user namespace where the uid_map file has not been written, the system calls that change user IDs will fail. Similarly, if the gid_map file has not been written, the sys?tem calls that change group IDs will fail. After the uid_map and gid_map files have been written, only the mapped values may be used in system calls that change user and group IDs.

For user IDs, the relevant system calls include setuid(2), setfsuid(2), setreuid(2), and setresuid(2). For group IDs, the relevant system calls include setgid(2), setfsgid(2), setregid(2), setresgid(2), and setgroups(2).

Writing "deny" to the /proc/[pid]/setgroups file before writing to /proc/[pid]/gid_map will permanently disable setgroups(2) in a user namespace and allow writing to /proc/[pid]/gid_map without having the CAP_SETGID capability in the parent user namespace.

The /proc/[pid]/setgroups file

The /proc/[pid]/setgroups file displays the string "allow" if processes in the user name? space that contains the process pid are permitted to employ the setgroups(2) system call; it displays "deny" if setgroups(2) is not permitted in that user namespace. Note that re? gardless of the value in the /proc/[pid]/setgroups file (and regardless of the process's capabilities), calls to setgroups(2) are also not permitted if /proc/[pid]/gid_map has not yet been set.

A privileged process (one with the CAP_SYS_ADMIN capability in the namespace) may write either of the strings "allow" or "deny" to this file before writing a group ID mapping for this user namespace to the file /proc/[pid]/gid_map. Writing the string "deny" prevents any process in the user namespace from employing setgroups(2).

The essence of the restrictions described in the preceding paragraph is that it is permit? ted to write to /proc/[pid]/setgroups only so long as calling setgroups(2) is disallowed because /proc/[pid]/gid_map has not been set. This ensures that a process cannot transi? tion from a state where setgroups(2) is allowed to a state where setgroups(2) is denied; a process can transition only from setgroups(2) being disallowed to setgroups(2) being al? lowed.

The default value of this file in the initial user namespace is "allow".

Once /proc/[pid]/gid_map has been written to (which has the effect of enabling set? groups(2) in the user namespace), it is no longer possible to disallow setgroups(2) by writing "deny" to /proc/[pid]/setgroups (the write fails with the error EPERM).

A child user namespace inherits the /proc/[pid]/setgroups setting from its parent.

If the setgroups file has the value "deny", then the setgroups(2) system call can't subse? quently be reenabled (by writing "allow" to the file) in this user namespace. (Attempts to do so fail with the error EPERM.) This restriction also propagates down to all child user namespaces of this user namespace.

The /proc/[pid]/setgroups file was added in Linux 3.19, but was backported to many earlier stable kernel series, because it addresses a security issue. The issue concerned files with permissions such as "rwx---rwx". Such files give fewer permissions to "group" than they do to "other". This means that dropping groups using setgroups(2) might allow a process file access that it did not formerly have. Before the existence of user name? spaces this was not a concern, since only a privileged process (one with the CAP_SETGID capability) could call setgroups(2). However, with the introduction of user namespaces, it became possible for an unprivileged process to create a new namespace in which the user had all privileges. This then allowed formerly unprivileged users to drop groups and thus gain file access that they did not previously have. The /proc/[pid]/setgroups file was added to address this security issue, by denying any pathway for an unprivileged process to drop groups with setgroups(2).

Unmapped user and group IDs

There are various places where an unmapped user ID (group ID) may be exposed to user space. For example, the first process in a new user namespace may call getuid(2) before a user ID mapping has been defined for the namespace. In most such cases, an unmapped user ID is converted to the overflow user ID (group ID); the default value for the overflow user ID (group ID) is 65534. See the descriptions of /proc/sys/kernel/overflowuid and /proc/sys/kernel/overflowgid in proc(5).

The cases where unmapped IDs are mapped in this fashion include system calls that return user IDs (getuid(2), getgid(2), and similar), credentials passed over a UNIX domain socket, credentials returned by stat(2), waitid(2), and the System V IPC "ctl" IPC_STAT operations, credentials exposed by /proc/[pid]/status and the files in /proc/sysvipc/*, credentials returned via the si_uid field in the siginfo_t received with a signal (see sigaction(2)), credentials written to the process accounting file (see acct(5)), and cre? dentials returned with POSIX message queue notifications (see mq_notify(3)).

There is one notable case where unmapped user and group IDs are not converted to the cor? responding overflow ID value. When viewing a uid_map or gid_map file in which there is no mapping for the second field, that field is displayed as 4294967295 (-1 as an unsigned in? teger).

Accessing files

In order to determine permissions when an unprivileged process accesses a file, the process credentials (UID, GID) and the file credentials are in effect mapped back to what they would be in the initial user namespace and then compared to determine the permissions that the process has on the file. The same is also of other objects that employ the cre?

dentials plus permissions mask accessibility model, such as System V IPC objects

Operation of file-related capabilities

Certain capabilities allow a process to bypass various kernel-enforced restrictions when performing operations on files owned by other users or groups. These capabilities are: CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, CAP_FOWNER, and CAP_FSETID.

Within a user namespace, these capabilities allow a process to bypass the rules if the process has the relevant capability over the file, meaning that:

* the process has the relevant effective capability in its user namespace; and
* the file's user ID and group ID both have valid mappings in the user namespace.

The CAP_FOWNER capability is treated somewhat exceptionally: it allows a process to bypass the corresponding rules so long as at least the file's user ID has a mapping in the user namespace (i.e., the file's group ID does not need to have a valid mapping).

Set-user-ID and set-group-ID programs

When a process inside a user namespace executes a set-user-ID (set-group-ID) program, the process's effective user (group) ID inside the namespace is changed to whatever value is mapped for the user (group) ID of the file. However, if either the user or the group ID of the file has no mapping inside the namespace, the set-user-ID (set-group-ID) bit is silently ignored: the new program is executed, but the process's effective user (group) ID is left unchanged. (This mirrors the semantics of executing a set-user-ID or set-group-ID program that resides on a filesystem that was mounted with the MS_NOSUID flag, as de? scribed in mount(2).)

Miscellaneous

When a process's user and group IDs are passed over a UNIX domain socket to a process in a different user namespace (see the description of SCM_CREDENTIALS in unix(7)), they are translated into the corresponding values as per the receiving process's user and group ID mappings.

CONFORMING TO

Namespaces are a Linux-specific feature.

NOTES

Over the years, there have been a lot of features that have been added to the Linux kernel that have been made available only to privileged users because of their potential to con? fuse set-user-ID-root applications. In general, it becomes safe to allow the root user in a user namespace to use those features because it is impossible, while in a user name?

space, to gain more privilege than the root user of a user namespace has.

Availability

Use of user namespaces requires a kernel that is configured with the CONFIG_USER_NS op?tion. User namespaces require support in a range of subsystems across the kernel. When an unsupported subsystem is configured into the kernel, it is not possible to configure user namespaces support.

As at Linux 3.8, most relevant subsystems supported user namespaces, but a number of filesystems did not have the infrastructure needed to map user and group IDs between user namespaces. Linux 3.9 added the required infrastructure support for many of the remaining unsupported filesystems (Plan 9 (9P), Andrew File System (AFS), Ceph, CIFS, CODA, NFS, and OCFS2). Linux 3.12 added support for the last of the unsupported major filesystems, XFS.

EXAMPLES

The program below is designed to allow experimenting with user namespaces, as well as other types of namespaces. It creates namespaces as specified by command-line options and then executes a command inside those namespaces. The comments and usage() function inside the program provide a full explanation of the program. The following shell session demon?strates its use.

First, we look at the run-time environment:

```
$ uname -rs     # Need Linux 3.8 or later
Linux 3.8.0
$ id -u         # Running as unprivileged user
1000
$ id -g
1000
```

Now start a new shell in new user (-U), mount (-m), and PID (-p) namespaces, with user ID (-M) and group ID (-G) 1000 mapped to 0 inside the user namespace:

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

The shell has PID 1, because it is the first process in the new PID namespace:

```
bash$ echo $$
1
```

Mounting a new /proc filesystem and listing all of the processes visible in the new PID namespace shows that the shell can't see any processes outside the PID namespace:

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

```
bash$ ps ax
  PID TTY      STAT   TIME COMMAND
    1 pts/3    S      0:00 bash
   22 pts/3    R+     0:00 ps ax
```

Inside the user namespace, the shell has user and group ID 0, and a full set of permitted and effective capabilities:

```
bash$ cat /proc/$$/status | egrep '^[UG]id'
Uid: 0    0    0    0
Gid: 0    0    0    0
bash$ cat /proc/$$/status | egrep '^Cap(Prm|Inh|Eff)'
CapInh:   0000000000000000
CapPrm:   0000001fffffffff
CapEff:   0000001fffffffff
```

Program source

```
/* userns_child_exec.c
   Licensed under GNU General Public License v2 or later
   Create a child process that executes a shell command in new
   namespace(s); allow UID and GID mappings to be specified when
   creating a user namespace.
*/
#define _GNU_SOURCE
#include <sched.h>
#include <unistd.h>
#include <stdint.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <errno.h>
/* A simple error-handling function: print an error message based
```

```
   on the value in 'errno' and terminate the calling process */
#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)
struct child_args {
    char **argv;        /* Command to be executed by child, with args */
    int    pipe_fd[2];  /* Pipe used to synchronize parent and child */
};
static int verbose;
static void
usage(char *pname)
{
    fprintf(stderr, "Usage: %s [options] cmd [arg...]\n\n", pname);
    fprintf(stderr, "Create a child process that executes a shell "
            "command in a new user namespace,\n"
            "and possibly also other new namespace(s).\n\n");
    fprintf(stderr, "Options can be:\n\n");
#define fpe(str) fprintf(stderr, "    %s", str);
    fpe("-i          New IPC namespace\n");
    fpe("-m          New mount namespace\n");
    fpe("-n          New network namespace\n");
    fpe("-p          New PID namespace\n");
    fpe("-u          New UTS namespace\n");
    fpe("-U          New user namespace\n");
    fpe("-M uid_map  Specify UID map for user namespace\n");
    fpe("-G gid_map  Specify GID map for user namespace\n");
    fpe("-z          Map user's UID and GID to 0 in user namespace\n");
    fpe("            (equivalent to: -M '0 <uid> 1' -G '0 <gid> 1')\n");
    fpe("-v          Display verbose messages\n");
    fpe("\n");
    fpe("If -z, -M, or -G is specified, -U is required.\n");
    fpe("It is not permitted to specify both -z and either -M or -G.\n");
    fpe("\n");
    fpe("Map strings for -M and -G consist of records of the form:\n");
```

```
    fpe("\n");
    fpe("    ID-inside-ns   ID-outside-ns   len\n");
    fpe("\n");
    fpe("A map string can contain multiple records, separated"
        " by commas;\n");
    fpe("the commas are replaced by newlines before writing"
        " to map files.\n");
    exit(EXIT_FAILURE);
}
/* Update the mapping file 'map_file', with the value provided in
   'mapping', a string that defines a UID or GID mapping. A UID or
   GID mapping consists of one or more newline-delimited records
   of the form:
       ID_inside-ns    ID-outside-ns   length
   Requiring the user to supply a string that contains newlines is
   of course inconvenient for command-line use. Thus, we permit the
   use of commas to delimit records in this string, and replace them
   with newlines before writing the string to the file. */
static void
update_map(char *mapping, char *map_file)
{
    int fd;
    size_t map_len;     /* Length of 'mapping' */
    /* Replace commas in mapping string with newlines */
    map_len = strlen(mapping);
    for (int j = 0; j < map_len; j++)
        if (mapping[j] == ',')
            mapping[j] = '\n';
    fd = open(map_file, O_RDWR);
    if (fd == -1) {
        fprintf(stderr, "ERROR: open %s: %s\n", map_file,
                strerror(errno));
        exit(EXIT_FAILURE);
```

```
    }
    if (write(fd, mapping, map_len) != map_len) {
        fprintf(stderr, "ERROR: write %s: %s\n", map_file,
                strerror(errno));
        exit(EXIT_FAILURE);
    }
    close(fd);
}
/* Linux 3.19 made a change in the handling of setgroups(2) and the
   'gid_map' file to address a security issue. The issue allowed
   *unprivileged* users to employ user namespaces in order to drop
   The upshot of the 3.19 changes is that in order to update the
   'gid_maps' file, use of the setgroups() system call in this
   user namespace must first be disabled by writing "deny" to one of
   the /proc/PID/setgroups files for this namespace.  That is the
   purpose of the following function. */
static void
proc_setgroups_write(pid_t child_pid, char *str)
{
    char setgroups_path[PATH_MAX];
    int fd;
    snprintf(setgroups_path, PATH_MAX, "/proc/%jd/setgroups",
            (intmax_t) child_pid);
    fd = open(setgroups_path, O_RDWR);
    if (fd == -1) {
        /* We may be on a system that doesn't support
           /proc/PID/setgroups. In that case, the file won't exist,
           and the system won't impose the restrictions that Linux 3.19
           added. That's fine: we don't need to do anything in order
           to permit 'gid_map' to be updated.
           However, if the error from open() was something other than
           the ENOENT error that is expected for that case,  let the
           user know. */
```

```
        if (errno != ENOENT)
            fprintf(stderr, "ERROR: open %s: %s\n", setgroups_path,
                strerror(errno));
        return;
    }
    if (write(fd, str, strlen(str)) == -1)
        fprintf(stderr, "ERROR: write %s: %s\n", setgroups_path,
            strerror(errno));
    close(fd);
}
static int              /* Start function for cloned child */
childFunc(void *arg)
{
    struct child_args *args = arg;
    char ch;
    /* Wait until the parent has updated the UID and GID mappings.
       See the comment in main(). We wait for end of file on a
       pipe that will be closed by the parent process once it has
       updated the mappings. */
    close(args->pipe_fd[1]);    /* Close our descriptor for the write
                               end of the pipe so that we see EOF
                               when parent closes its descriptor */
    if (read(args->pipe_fd[0], &ch, 1) != 0) {
        fprintf(stderr,
                "Failure in child: read from pipe returned != 0\n");
        exit(EXIT_FAILURE);
    }
    close(args->pipe_fd[0]);
    /* Execute a shell command */
    printf("About to exec %s\n", args->argv[0]);
    execvp(args->argv[0], args->argv);
    errExit("execvp");
}
```

```
#define STACK_SIZE (1024 * 1024)

static char child_stack[STACK_SIZE];    /* Space for child's stack */

int

main(int argc, char *argv[])

{

    int flags, opt, map_zero;

    pid_t child_pid;

    struct child_args args;

    char *uid_map, *gid_map;

    const int MAP_BUF_SIZE = 100;

    char map_buf[MAP_BUF_SIZE];

    char map_path[PATH_MAX];

    /* Parse command-line options. The initial '+' character in

       the final getopt() argument prevents GNU-style permutation

       of command-line options. That's useful, since sometimes

       the 'command' to be executed by this program itself

       has command-line options. We don't want getopt() to treat

       those as options to this program. */

    flags = 0;

    verbose = 0;

    gid_map = NULL;

    uid_map = NULL;

    map_zero = 0;

    while ((opt = getopt(argc, argv, "+imnpuUM:G:zv")) != -1) {

        switch (opt) {

        case 'i': flags |= CLONE_NEWIPC;        break;

        case 'm': flags |= CLONE_NEWNS;         break;

        case 'n': flags |= CLONE_NEWNET;        break;

        case 'p': flags |= CLONE_NEWPID;        break;

        case 'u': flags |= CLONE_NEWUTS;        break;

        case 'v': verbose = 1;                  break;

        case 'z': map_zero = 1;                 break;

        case 'M': uid_map = optarg;             break;
```

```
    case 'G': gid_map = optarg;             break;
    case 'U': flags |= CLONE_NEWUSER;       break;
    default:  usage(argv[0]);
    }
}
/* -M or -G without -U is nonsensical */
if (((uid_map != NULL || gid_map != NULL || map_zero) &&
            !(flags & CLONE_NEWUSER)) ||
        (map_zero && (uid_map != NULL || gid_map != NULL)))
    usage(argv[0]);
args.argv = &argv[optind];
/* We use a pipe to synchronize the parent and child, in order to
   ensure that the parent sets the UID and GID maps before the child
   calls execve(). This ensures that the child maintains its
   capabilities during the execve() in the common case where we
   want to map the child's effective user ID to 0 in the new user
   namespace. Without this synchronization, the child would lose
   its capabilities if it performed an execve() with nonzero
   user IDs (see the capabilities(7) man page for details of the
   transformation of a process's capabilities during execve()). */
if (pipe(args.pipe_fd) == -1)
    errExit("pipe");
/* Create the child in new namespace(s) */
child_pid = clone(childFunc, child_stack + STACK_SIZE,
                  flags | SIGCHLD, &args);
if (child_pid == -1)
    errExit("clone");
/* Parent falls through to here */
if (verbose)
    printf("%s: PID of child created by clone() is %jd\n",
            argv[0], (intmax_t) child_pid);
/* Update the UID and GID maps in the child */
if (uid_map != NULL || map_zero) {
```

```
            snprintf(map_path, PATH_MAX, "/proc/%jd/uid_map",
                    (intmax_t) child_pid);
            if (map_zero) {
                snprintf(map_buf, MAP_BUF_SIZE, "0 %jd 1",
                        (intmax_t) getuid());
                uid_map = map_buf;
            }
            update_map(uid_map, map_path);
        }
        if (gid_map != NULL || map_zero) {
            proc_setgroups_write(child_pid, "deny");
            snprintf(map_path, PATH_MAX, "/proc/%jd/gid_map",
                    (intmax_t) child_pid);
            if (map_zero) {
                snprintf(map_buf, MAP_BUF_SIZE, "0 %ld 1",
                        (intmax_t) getgid());
                gid_map = map_buf;
            }
            update_map(gid_map, map_path);
        }
        /* Close the write end of the pipe, to signal to the child that we
           have updated the UID and GID maps */
        close(args.pipe_fd[1]);
        if (waitpid(child_pid, NULL, 0) == -1)      /* Wait for child */
            errExit("waitpid");
        if (verbose)
            printf("%s: terminating\n", argv[0]);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

newgidmap(1), newuidmap(1), clone(2), ptrace(2), setns(2), unshare(2), proc(5), subgid(5), subuid(5), capabilities(7), cgroup_namespaces(7), credentials(7), namespaces(7), pid_name?
spaces(7)

The kernel source file Documentation/namespaces/resource-control.txt.

COLOPHON

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