unshare system call

This document describes the new system call, unshare(). The document provides an overview of the feature, why it is needed, how it can be used, its interface specification, design, implementation and how it can be tested.

Change Log

version 0.1 Initial document, Janak Desai (janak@us.ibm.com), Jan 11, 2006

Contents

1) Overview

Most legacy operating system kernels support an abstraction of threads as multiple execution contexts within a process. These kernels provide special resources and mechanisms to maintain these "threads". The Linux kernel, in a clever and simple manner, does not make distinction between processes and "threads". The kernel allows processes to share resources and thus they can achieve legacy "threads" behavior without requiring additional data structures and mechanisms in the kernel. The power of implementing threads in this manner comes not only from its simplicity but also from allowing application programmers to work outside the confinement of all-or-nothing shared resources of legacy threads. On Linux, at the time of thread creation using the clone system call, applications can selectively choose which resources to share between threads.

unshare() system call adds a primitive to the Linux thread model that allows threads to selectively 'unshare' any resources that were being shared at the time of their creation. unshare() was conceptualized by Al Viro in the August of 2000, on the Linux-Kernel mailing list, as part of the discussion on POSIX threads on Linux. unshare() augments the usefulness of Linux threads for applications that would like to control shared resources without creating a new process. unshare() is a natural addition to the set of available primitives on Linux that implement the concept of process/thread as a virtual machine.

2) Benefits

unshare() would be useful to large application frameworks such as PAM where creating a new process to control sharing/unsharing of process resources is not possible. Since namespaces are shared by default when creating a new process using fork or clone, unshare() can benefit even non-threaded applications if they have a need to disassociate from default shared namespace. The following lists two use-cases where unshare() can be used.

2.1 Per-security context namespaces

unshare() can be used to implement polyinstantiated directories using the kernel's per-process namespace mechanism. Polyinstantiated directories, such as per-user and/or per-security context instance of /tmp, /var/tmp or per-security context instance of a user's home directory, isolate user processes when working with these directories. Using unshare(), a PAM module can easily setup a private namespace for a user at login. Polyinstantiated directories are required for Common Criteria certification with Labeled System Protection Profile, however, with the availability of shared-tree feature in the Linux kernel, even regular Linux systems can benefit from setting up private namespaces at login and polyinstantiating /tmp, /var/tmp and other directories deemed appropriate by system administrators.

2.2 unsharing of virtual memory and/or open files

Consider a client/server application where the server is processing client requests by creating processes that share resources such as virtual memory and open files. Without unshare(), the server has to decide what needs to be shared at the time of creating the process which services the request. unshare() allows the server an ability to disassociate parts of the context during the servicing of the request. For large and complex middleware application frameworks, this ability to unshare() after the process was created can be very useful.

3) Cost

In order to not duplicate code and to handle the fact that unshare() works on an active task (as opposed to clone/fork working on a newly allocated inactive task) unshare() had to make minor reorganizational changes to copy_* functions utilized by clone/fork system call. There is a cost associated with altering existing, well tested and stable code to implement a new feature that may not get exercised extensively in the beginning. However, with proper design and code review of the changes and creation of an unshare() test for the LTP the benefits of this new feature can exceed its cost.

4) Requirements

unshare() reverses sharing that was done using clone(2) system call, so unshare() should have a similar interface as clone(2). That is, since flags in clone(int flags, void *stack) specifies what should be shared, similar flags in unshare(int flags) should specify what should be unshared. Unfortunately, this may appear to invert the meaning of the flags from the way they are used in clone(2). However, there was no easy solution that was less confusing and that allowed incremental context unsharing in future without an ABI change.

unshare() interface should accommodate possible future addition of new context flags without requiring a rebuild of old applications. If and when new context flags are added, unshare() design should allow incremental unsharing of those resources on an as needed basis.

5) Functional Specification

NAME

unshare - disassociate parts of the process execution context

SYNOPSIS

#include <sched.h>

int unshare(int flags);

DESCRIPTION

unshare() allows a process to disassociate parts of its execution context that are currently being shared with other processes. Part of execution context, such as the namespace, is shared by default when a new process is created using fork(2), while other parts, such as the virtual memory, open file descriptors, etc, may be shared by explicit request to share them when creating a process using clone(2).

The main use of unshare() is to allow a process to control its shared execution context without creating a new process.

The flags argument specifies one or bitwise-or'ed of several of the following constants.

CLONE_FS
If CLONE_FS is set, file system information of the caller is disassociated from the shared file system information.

CLONE_FILES
If CLONE_FILES is set, the file descriptor table of the caller is disassociated from the shared file descriptor table.

CLONE_NEWNS
If CLONE_NEWNS is set, the namespace of the caller is disassociated from the shared namespace.

CLONE_VM
If CLONE_VM is set, the virtual memory of the caller is disassociated from the shared virtual memory.

RETURN VALUE

On success, zero returned. On failure, -1 is returned and errno is

ERRORS

EPERM CLONE_NEWNS was specified by a non-root process (process without CAP_SYS_ADMIN).

ENOMEM Cannot allocate sufficient memory to copy parts of caller's context that need to be unshared.

EINVAL Invalid flag was specified as an argument.

CONFORMING TO

The unshare() call is Linux-specific and should not be used in programs intended to be portable.

SEE ALSO

clone(2), fork(2)

6) High Level Design

Depending on the flags argument, the unshare() system call allocates appropriate process context structures, populates it with values from the current shared version, associates newly duplicated structures with the current task structure and releases corresponding shared versions. Helper functions of clone (copy_*) could not be used directly by unshare() because of the following two reasons.

1. clone operates on a newly allocated not-yet-active task structure, where as unshare() operates on the current active task. Therefore unshare() has to take appropriate task_lock() before associating newly duplicated context structures
2. unshare() has to allocate and duplicate all context structures that are being unshared, before associating them with the current task and releasing older shared structures. Failure do so will create race conditions and/or oops when trying to backout due to an error. Consider the case of unsharing both virtual memory and namespace. After successfully unsharing vm, if the system call encounters an error while allocating new namespace structure, the error return code will have to reverse the unsharing of vm. As part of the reversal the system call will have to go back to older, shared, vm structure, which may not exist anymore.

Therefore code from copy_* functions that allocated and duplicated current context structure was moved into new dup_* functions. Now, copy_* functions call dup_* functions to allocate and duplicate appropriate context structures and then associate them with the task structure that is being constructed. unshare() system call on the other hand performs the following:

1. Check flags to force missing, but implied, flags
2. For each context structure, call the corresponding unshare() helper function to allocate and duplicate a new context structure, if the appropriate bit is set in the flags argument.
3. If there is no error in allocation and duplication and there are new context structures then lock the current task structure, associate new context structures with the current task structure, and release the lock on the current task structure.
4. Appropriately release older, shared, context structures.

7) Low Level Design

Implementation of unshare() can be grouped in the following 4 different items:

a. Reorganization of existing copy_* functions
b. unshare() system call service function
c. unshare() helper functions for each different process context
d. Registration of system call number for different architectures

7.1) Reorganization of copy_* functions

Each copy function such as copy_mm, copy_namespace, copy_files, etc, had roughly two components. The first component allocated and duplicated the appropriate structure and the second component linked it to the task structure passed in as an argument to the copy function. The first component was split into its own function. These dup_* functions allocated and duplicated the appropriate context structure. The reorganized copy_* functions invoked their corresponding dup_* functions and then linked the newly duplicated structures to the task structure with which the copy function was called.

7.2) unshare() system call service function

- Check flags Force implied flags. If CLONE_THREAD is set force CLONE_VM. If CLONE_VM is set, force CLONE_SIGHAND. If CLONE_SIGHAND is set and signals are also being shared, force CLONE_THREAD. If CLONE_NEWNS is set, force CLONE_FS.
- For each context flag, invoke the corresponding unshare_* helper routine with flags passed into the system call and a reference to pointer pointing the new unshared structure
- If any new structures are created by unshare_* helper functions, take the task_lock() on the current task, modify appropriate context pointers, and release the task lock.
- For all newly unshared structures, release the corresponding older, shared, structures.

7.3) unshare_* helper functions

For unshare_* helpers corresponding to CLONE_SYSVSEM, CLONE_SIGHAND, and CLONE_THREAD, return -EINVAL since they are not implemented yet. For others, check the flag value to see if the unsharing is required for that structure. If it is, invoke the corresponding dup_* function to allocate and duplicate the structure and return a pointer to it.

7.4) Finally

Appropriately modify architecture specific code to register the new system call.

8) Test Specification

The test for unshare() should test the following:

1. Valid flags: Test to check that clone flags for signal and signal handlers, for which unsharing is not implemented yet, return -EINVAL.
2. Missing/implied flags: Test to make sure that if unsharing namespace without specifying unsharing of filesystem,

correctly unshares both namespace and filesystem information.

3. For each of the four (namespace, filesystem, files and vm) supported unsharing, verify that the system call correctly unshares the appropriate structure. Verify that unsharing them individually as well as in combination with each other works as expected.
4. Concurrent execution: Use shared memory segments and futex on an address in the shm segment to synchronize execution of about 10 threads. Have a couple of threads execute execve, a couple _exit and the rest unshare with different combination of flags. Verify that unsharing is performed as expected and that there are no oops or hangs.

9) Future Work

The current implementation of unshare() does not allow unsharing of signals and signal handlers. Signals are complex to begin with and to unshare signals and/or signal handlers of a currently running process is even more complex. If in the future there is a specific need to allow unsharing of signals and/or signal handlers, it can be incrementally added to unshare() without affecting legacy applications using unshare().