Converting file systems to support idmapped mounts

Stéphane Graber
Owner, Zabbly
stgraber@stgraber.org

Aleksandr Mikhalitsyn
Software engineer, Canonical
aleksandr.mikhalitsyn@canonical.com

Intro

Idmappings

- caller’s idmapping
  - You always have it, just look into /proc/self/{u,g}id_map
    - 0 0 4294967295
- filesystem’s idmapping (also known as a superblock idmapping)
  - (struct super_block *)->s_user_ns
  - Taken from current_user_ns() on mount() or fsconfig(FSCONFIG_CMD_CREATE*)
- **mount’s idmapping**
  - Attached to the mount not super block

Caller’s idmapping

- All UID/GIDs from the user space perspective are mapped in accordance with it
  - stat()
  - getuid()
  - getsockopt(... SO_PEERCRED …)
  - …
- For userspace we have uid_t and gid_t types
  - getuid() -> from_kuid_munged(current_user_ns(), current_uid())
- Internally, we have k{u,g}id_t types
  - make_kuid(user_ns, [uid_t value])
  - setuid(uid_t = 100) -> make_kuid(current_user_ns(), 100) -> kuid_t value

File system’s idmapping

- uid_t i_uid_read(const struct inode *inode)
  - Called on the write path
- void i_uid_write(struct inode *inode, uid_t uid)
  - inode->i_uid = make_kuid(sb->s_user_ns, uid_t value)

How it works together

caller id: u1000

caller's idmapping: u0:k10000:r10000

file system's idmapping: u0:k0:r4294967295

mount's idmapping: u0:v10000:r10000

1. make_kuid(u0:k10000:r10000, u1000) = k11000
2. from_kuid(u0:v10000:r10000, v11000) = u1000
3. make_kuid(u0:k0:r4294967295, u1000) = k1000 (think what happens for u0:k1000:r1 and for u1000:k0:r1)
4. from_kuid(u0:k0:r4294967295, k1000) = u1000

An inode will be created with UID = 1000

How to create idmapped mount

- https://github.com/brauner/mount-idmapped
  - ./mount-idmapped --map-mount b:1000:0:1 /source /idmapped
    - 1000 (file system) -> 0 (idmapped mount)
    - 0 (file system) -> overflowuid[=65534] (idmapped)
- mount --bind -o X-mount.idmap=b:1000:0:1 /source /idmapped
  - Landed into util-linux in Jan 2023
- In both cases, you can use /proc/<pid>/ns/user instead of explicit mapping definition
- Inside we have: open_tree, mount_setattr (with MOUNT_ATTR_IDMAP), move_mount

Current state

File systems with idmap support (6.8-rc2)

1. ext4
2. btrfs
3. xfs
4. *fat
5. f2fs
6. ntfs3
7. squashfs
8. tmpfs
9. erofs
10. Ceph (starting from 6.7)
11. ZFS (out of tree)

How to port a file system

How to port

- &nop_mnt_idmap -> idmap
- current_fsuid() -> mapped_fsuid()
- Add FS_ALLOW_IDMAP to fs_flags
- … it’s not that simple, unfortunately

Things to look at:

- Read code paths
  - i_op->getattr (if fs have one)
  - i_op->permission (if fs have one)
  - i_op->get_acl (*)
  - …
- Write code paths
  - i_op->(mknod|mkdir|symlink|create|atomic_open)
  - i_op->setattr
  - i_op->set_acl
  - …

Local file systems

Local file systems

As we have everything in the kernel => we have an access to all the data and file system configuration (mount options) to handle everything properly.

Remote(-like) file systems

Potential problems

- File system handles UID/GID-based permission checks on the server side
  - fuse: if “default_permissions” mode is not enabled
- File system performs some permission checks in the unusual places, for example in the i_op->lookup where we don’t have an idmapping passed! [we do these checks in the generic VFS code, see may_lookup()]
- File system does some UID/GID translation (NFS idmapper, some fuse-based file systems also support that)

General principle in there is to make all the VFS idmappings-related stuff in the kernel and never send it over the network. But it’s close to impossible.

Example: ceph

- Can do some permission checks on the server-side (in some configurations) in addition to a classical “generic_permission” helper used
  - Obviously, sends UID/GIDs over the wire
- Does permissions checks for almost any operations (including lookup)
  - Only a problem if you have a path-based restrictions in place
- Uses get_current_cred()->fs{u,g}id everywhere
  - We usually expect that once FD is opened, fs uses (struct file *)->f_cred->...

Example: ceph (what we did)

- Did not touch the existing MDS-side permission checks machinery
- Extended on-wire protocol and added two new fields (inode_{u,g}id), which makes sense for inode-creating requests like symlink, mknod, mkdir, create/atomic_open
  - Put an id-mapped UID/GID values in there

Example: fuse

- Permission checks can be fully offloaded to the user space
- Has a “default_permissions” mode (in-kernel)
- The kernel sends a caller’s fsuid/fsgid with each request
    - these values are used to set ownership on the new inodes

Example: fuse (current approach)

- Support only “default_permissions” mode
  - We assume that no extra UID/GID-based checks are performed in the user space
- Extend fuse protocol and add two additional fields for inode owner UID/GID
  - Obviously, these fields are mapped in accordance with mount’s idmapping and based on the caller’s fsuid/fsgid
- Have done (PoC) user space conversions for:
  - overlayfs-fuse
  - cephfs-fuse
  - GlusterFS

TODO

Planned

1. fuse (patches are sent)
2. 9pfs
3. virtiofs

Thank you! Questions?

Stéphane Graber
Owner, Zabbly
stgraber@stgraber.org

Aleksandr Mikhalitsyn
Software engineer, Canonical
aleksandr.mikhalitsyn@canonical.com

Links

1. ceph: support idmapped mounts
2. fuse: basic support for idmapped mounts
3. Documentation/filesystems/idmappings.rst