Idmappings

Most filesystem developers will have encountered idmappings. They are used when reading from or writing ownership to disk, reporting ownership to userspace, or for permission checking. This document is aimed at filesystem developers that want to know how idmappings work.

Formal notes

An idmapping is essentially a translation of a range of ids into another or the same range of ids. The notational convention for idmappings that is widely used in userspace is:

```
u:k:r
```

`u` indicates the first element in the upper idmapset `U` and `k` indicates the first element in the lower idmapset `K`. The `r` parameter indicates the range of the idmapping, i.e. how many ids are mapped. From now on, we will always prefix ids with `u` or `k` to make it clear whether we're talking about an id in the upper or lower idmapset.

To see what this looks like in practice, let's take the following idmapping:

```
u22:k10000:r3
```

and write down the mappings it will generate:

```
u22 -> k10000
u23 -> k10001
u24 -> k10002
```

From a mathematical viewpoint `U` and `K` are well-ordered sets and an idmapping is an order isomorphism from `U` into `K`. So `U` and `K` are order isomorphic. In fact, `U` and `K` are always well-ordered subsets of the set of all possible ids useable on a given system.

Looking at this mathematically briefly will help us highlight some properties that make it easier to understand how we can translate between idmappings. For example, we know that the inverse idmapping is an order isomorphism as well:

```
k10000 -> u22
k10001 -> u23
k10002 -> u24
```

Given that we are dealing with order isomorphisms plus the fact that we're dealing with subsets we can embedd idmappings into each other, i.e. we can sensibly translate between different idmappings. For example, assume we've been given the three idmappings:

```
1. u0:k10000:r10000
2. u0:k20000:r10000
3. u0:k30000:r10000
```

and id `k11000` which has been generated by the first idmapping by mapping `u1000` from the upper idmapset down to `k11000` in the lower idmapset.

Because we're dealing with order isomorphic subsets it is meaningful to ask what id `k11000` corresponds to in the second or third idmapping. The straightfoward algorithm to use is to apply the inverse of the first idmapping, mapping `k11000` up to `u1000`. Afterwards, we can map `u1000` down using either the second idmapping mapping or third idmapping mapping. The second idmapping would map `u1000` down to `21000`. The third idmapping would map `u1000` down to `u31000`.

If we were given the same task for the following three idmappings:

```
1. u0:k10000:r10000
2. u0:k20000:r200
3. u0:k30000:r300
```

we would fail to translate as the sets aren't order isomorphic over the full range of the first idmapping anymore (However they are order isomorphic over the full range of the second idmapping.). Neither the second or third idmapping contain `u1000` in the upper idmapset `U`. This is equivalent to not having an id mapped. We can simply say that `u1000` is unmapped in the second and third idmapping. The kernel will report unmapped ids as the overflowuid `(uid_t)-1` or overflowgid `(gid_t)-1` to userspace.

The algorithm to calculate what a given id maps to is pretty simple. First, we need to verify that the range can contain our target id. We will skip this step for simplicity. After that if we want to know what `id` maps to we can do simple calculations:

- If we want to map from left to right:

  ```
  u:k:r
  id - u + k = n
  ```

- If we want to map from right to left:

  ```
  u:k:r
  id - k + u = n
  ```

Instead of "left to right" we can also say "down" and instead of "right to left" we can also say "up". Obviously mapping down and up invert each other.

To see whether the simple formulas above work, consider the following two idmappings:

```
1. u0:k20000:r10000
2. u500:k30000:r10000
```

Assume we are given `k21000` in the lower idmapset of the first idmapping. We want to know what id this was mapped from in the upper idmapset of the first idmapping. So we're mapping up in the first idmapping:

```
id     - k      + u  = n
k21000 - k20000 + u0 = u1000
```

Now assume we are given the id `u1100` in the upper idmapset of the second idmapping and we want to know what this id maps down to in the lower idmapset of the second idmapping. This means we're mapping down in the second idmapping:

```
id    - u    + k      = n
u1100 - u500 + k30000 = k30600
```

General notes

In the context of the kernel an idmapping can be interpreted as mapping a range of userspace ids into a range of kernel ids:

```
userspace-id:kernel-id:range
```

A userspace id is always an element in the upper idmapset of an idmapping of type `uid_t` or `gid_t` and a kernel id is always an element in the lower idmapset of an idmapping of type `kuid_t` or `kgid_t`. From now on "userspace id" will be used to refer to the well known `uid_t` and `gid_t` types and "kernel id" will be used to refer to `kuid_t` and `kgid_t`.

The kernel is mostly concerned with kernel ids. They are used when performing permission checks and are stored in an inode's `i_uid` and `i_gid` field. A userspace id on the other hand is an id that is reported to userspace by the kernel, or is passed by userspace to the kernel, or a raw device id that is written or read from disk.

Note that we are only concerned with idmappings as the kernel stores them not how userspace would specify them.

For the rest of this document we will prefix all userspace ids with `u` and all kernel ids with `k`. Ranges of idmappings will be prefixed with `r`. So an idmapping will be written as `u0:k10000:r10000`.

For example, the id `u1000` is an id in the upper idmapset or "userspace idmapset" starting with `u1000`. And it is mapped to `k11000` which is a kernel id in the lower idmapset or "kernel idmapset" starting with `k10000`.

A kernel id is always created by an idmapping. Such idmappings are associated with user namespaces. Since we mainly care about how idmappings work we're not going to be concerned with how idmappings are created nor how they are used outside of the filesystem context. This is best left to an explanation of user namespaces.

The initial user namespace is special. It always has an idmapping of the following form:

```
u0:k0:r4294967295
```

which is an identity idmapping over the full range of ids available on this system.

Other user namespaces usually have non-identity idmappings such as:

```
u0:k10000:r10000
```

When a process creates or wants to change ownership of a file, or when the ownership of a file is read from disk by a filesystem, the userspace id is immediately translated into a kernel id according to the idmapping associated with the relevant user namespace.

For instance, consider a file that is stored on disk by a filesystem as being owned by `u1000`:

- If a filesystem were to be mounted in the initial user namespaces (as most filesystems are) then the initial idmapping will be used. As we saw this is simply the identity idmapping. This would mean id `u1000` read from disk would be mapped to id `k1000`. So an inode's `i_uid` and `i_gid` field would contain `k1000`.
- If a filesystem were to be mounted with an idmapping of `u0:k10000:r10000` then `u1000` read from disk would be mapped to `k11000`. So an inode's `i_uid` and `i_gid` would contain `k11000`.

Translation algorithms

We've already seen briefly that it is possible to translate between different idmappings. We'll now take a closer look how that works.

Crossmapping

This translation algorithm is used by the kernel in quite a few places. For example, it is used when reporting back the ownership of a file to userspace via the `stat()` system call family.

If we've been given `k11000` from one idmapping we can map that id up in another idmapping. In order for this to work both idmappings need to contain the same kernel id in their kernel idmapsets. For example, consider the following idmappings:

```
1. u0:k10000:r10000
2. u20000:k10000:r10000
```

and we are mapping `u1000` down to `k11000` in the first idmapping . We can then translate `k11000` into a userspace id in the second idmapping using the kernel idmapset of the second idmapping:

```
/* Map the kernel id up into a userspace id in the second idmapping. */
from_kuid(u20000:k10000:r10000, k11000) = u21000
```

Note, how we can get back to the kernel id in the first idmapping by inverting the algorithm:

```
/* Map the userspace id down into a kernel id in the second idmapping. */
make_kuid(u20000:k10000:r10000, u21000) = k11000

/* Map the kernel id up into a userspace id in the first idmapping. */
from_kuid(u0:k10000:r10000, k11000) = u1000
```

This algorithm allows us to answer the question what userspace id a given kernel id corresponds to in a given idmapping. In order to be able to answer this question both idmappings need to contain the same kernel id in their respective kernel idmapsets.

For example, when the kernel reads a raw userspace id from disk it maps it down into a kernel id according to the idmapping associated with the filesystem. Let's assume the filesystem was mounted with an idmapping of `u0:k20000:r10000` and it reads a file owned by `u1000` from disk. This means `u1000` will be mapped to `k21000` which is what will be stored in the inode's `i_uid` and `i_gid` field.

When someone in userspace calls `stat()` or a related function to get ownership information about the file the kernel can't simply map the id back up according to the filesystem's idmapping as this would give the wrong owner if the caller is using an idmapping.

So the kernel will map the id back up in the idmapping of the caller. Let's assume the caller has the slighly unconventional idmapping `u3000:k20000:r10000` then `k21000` would map back up to `u4000`. Consequently the user would see that this file is owned by `u4000`.

Remapping

It is possible to translate a kernel id from one idmapping to another one via the userspace idmapset of the two idmappings. This is equivalent to remapping a kernel id.

Let's look at an example. We are given the following two idmappings:

```
1. u0:k10000:r10000
2. u0:k20000:r10000
```

and we are given `k11000` in the first idmapping. In order to translate this kernel id in the first idmapping into a kernel id in the second idmapping we need to perform two steps:

1. Map the kernel id up into a userspace id in the first idmapping:

   ```
   /* Map the kernel id up into a userspace id in the first idmapping. */
   from_kuid(u0:k10000:r10000, k11000) = u1000
   ```

2. Map the userspace id down into a kernel id in the second idmapping:

   ```
   /* Map the userspace id down into a kernel id in the second idmapping. */
   make_kuid(u0:k20000:r10000, u1000) = k21000
   ```

As you can see we used the userspace idmapset in both idmappings to translate the kernel id in one idmapping to a kernel id in another idmapping.

This allows us to answer the question what kernel id we would need to use to get the same userspace id in another idmapping. In order to be able to answer this question both idmappings need to contain the same userspace id in their respective userspace idmapsets.

Note, how we can easily get back to the kernel id in the first idmapping by inverting the algorithm:

1. Map the kernel id up into a userspace id in the second idmapping:

   ```
   /* Map the kernel id up into a userspace id in the second idmapping. */
   from_kuid(u0:k20000:r10000, k21000) = u1000
   ```

2. Map the userspace id down into a kernel id in the first idmapping:

   ```
   /* Map the userspace id down into a kernel id in the first idmapping. */
   make_kuid(u0:k10000:r10000, u1000) = k11000
   ```

Another way to look at this translation is to treat it as inverting one idmapping and applying another idmapping if both idmappings have the relevant userspace id mapped. This will come in handy when working with idmapped mounts.

Invalid translations

It is never valid to use an id in the kernel idmapset of one idmapping as the id in the userspace idmapset of another or the same idmapping. While the kernel idmapset always indicates an idmapset in the kernel id space the userspace idmapset indicates a userspace id. So the following translations are forbidden:

```
/* Map the userspace id down into a kernel id in the first idmapping. */
make_kuid(u0:k10000:r10000, u1000) = k11000

/* INVALID: Map the kernel id down into a kernel id in the second idmapping. */
make_kuid(u10000:k20000:r10000, k110000) = k21000
                                ~~~~~~~
```

and equally wrong:

```
/* Map the kernel id up into a userspace id in the first idmapping. */
from_kuid(u0:k10000:r10000, k11000) = u1000

/* INVALID: Map the userspace id up into a userspace id in the second idmapping. */
from_kuid(u20000:k0:r10000, u1000) = k21000
                            ~~~~~
```

Idmappings when creating filesystem objects

The concepts of mapping an id down or mapping an id up are expressed in the two kernel functions filesystem developers are rather familiar with and which we've already used in this document:

```
/* Map the userspace id down into a kernel id. */
make_kuid(idmapping, uid)

/* Map the kernel id up into a userspace id. */
from_kuid(idmapping, kuid)
```

We will take an abbreviated look into how idmappings figure into creating filesystem objects. For simplicity we will only look at what happens when the VFS has already completed path lookup right before it calls into the filesystem itself. So we're concerned with what happens when e.g. `vfs_mkdir()` is called. We will also assume that the directory we're creating filesystem objects in is readable and writable for everyone.

When creating a filesystem object the caller will look at the caller's filesystem ids. These are just regular `uid_t` and `gid_t` userspace ids but they are exclusively used when determining file ownership which is why they are called "filesystem ids". They are usually identical to the uid and gid of the caller but can differ. We will just assume they are always identical to not get lost in too many details.

When the caller enters the kernel two things happen:

1. Map the caller's userspace ids down into kernel ids in the caller's idmapping. (To be precise, the kernel will simply look at the kernel ids stashed in the credentials of the current task but for our education we'll pretend this translation happens just in time.)
2. Verify that the caller's kernel ids can be mapped up to userspace ids in the filesystem's idmapping.

The second step is important as regular filesystem will ultimately need to map the kernel id back up into a userspace id when writing to disk. So with the second step the kernel guarantees that a valid userspace id can be written to disk. If it can't the kernel will refuse the creation request to not even remotely risk filesystem corruption.

The astute reader will have realized that this is simply a varation of the crossmapping algorithm we mentioned above in a previous section. First, the kernel maps the caller's userspace id down into a kernel id according to the caller's idmapping and then maps that kernel id up according to the filesystem's idmapping.

Example 1

```
caller id:             u1000
caller idmapping:      u0:k0:r4294967295
filesystem idmapping: u0:k0:r4294967295
```

Both the caller and the filesystem use the identity idmapping:

1. Map the caller's userspace ids into kernel ids in the caller's idmapping:

   ```
   make_kuid(u0:k0:r4294967295, u1000) = k1000
   ```

2. Verify that the caller's kernel ids can be mapped to userspace ids in the filesystem's idmapping.

   For this second step the kernel will call the function `fsuidgid_has_mapping()` which ultimately boils down to calling `from_kuid()`:

   ```
   from_kuid(u0:k0:r4294967295, k1000) = u1000
   ```

In this example both idmappings are the same so there's nothing exciting going on. Ultimately the userspace id that lands on disk will be `u1000`.

Example 2

```
caller id:             u1000
caller idmapping:      u0:k10000:r10000
filesystem idmapping: u0:k20000:r10000
```

1. Map the caller's userspace ids down into kernel ids in the caller's idmapping:

   ```
   make_kuid(u0:k10000:r10000, u1000) = k11000
   ```

2. Verify that the caller's kernel ids can be mapped up to userspace ids in the filesystem's idmapping:

   ```
   from_kuid(u0:k20000:r10000, k11000) = u-1
   ```

It's immediately clear that while the caller's userspace id could be successfully mapped down into kernel ids in the caller's idmapping the kernel ids could not be mapped up according to the filesystem's idmapping. So the kernel will deny this creation request.

Note that while this example is less common, because most filesystem can't be mounted with non-initial idmappings this is a general problem as we can see in the next examples.

Example 3

```
caller id:             u1000
caller idmapping:      u0:k10000:r10000
filesystem idmapping: u0:k0:r4294967295
```

1. Map the caller's userspace ids down into kernel ids in the caller's idmapping:

   ```
   make_kuid(u0:k10000:r10000, u1000) = k11000
   ```

2. Verify that the caller's kernel ids can be mapped up to userspace ids in the filesystem's idmapping:

   ```
   from_kuid(u0:k0:r4294967295, k11000) = u11000
   ```

We can see that the translation always succeeds. The userspace id that the filesystem will ultimately put to disk will always be identical to the value of the kernel id that was created in the caller's idmapping. This has mainly two consequences.

First, that we can't allow a caller to ultimately write to disk with another userspace id. We could only do this if we were to mount the whole fileystem with the caller's or another idmapping. But that solution is limited to a few filesystems and not very flexible. But this is a use-case that is pretty important in containerized workloads.

Second, the caller will usually not be able to create any files or access directories that have stricter permissions because none of the filesystem's kernel ids map up into valid userspace ids in the caller's idmapping

1. Map raw userspace ids down to kernel ids in the filesystem's idmapping:

   ```
   make_kuid(u0:k0:r4294967295, u1000) = k1000
   ```

2. Map kernel ids up to userspace ids in the caller's idmapping:

   ```
   from_kuid(u0:k10000:r10000, k1000) = u-1
   ```

Example 4

```
file id:               u1000
caller idmapping:      u0:k10000:r10000
filesystem idmapping: u0:k0:r4294967295
```

In order to report ownership to userspace the kernel uses the crossmapping algorithm introduced in a previous section:

1. Map the userspace id on disk down into a kernel id in the filesystem's idmapping:

   ```
   make_kuid(u0:k0:r4294967295, u1000) = k1000
   ```

2. Map the kernel id up into a userspace id in the caller's idmapping:

   ```
   from_kuid(u0:k10000:r10000, k1000) = u-1
   ```

The crossmapping algorithm fails in this case because the kernel id in the filesystem idmapping cannot be mapped up to a userspace id in the caller's idmapping. Thus, the kernel will report the ownership of this file as the overflowid.

Example 5

```
file id:               u1000
caller idmapping:      u0:k10000:r10000
filesystem idmapping: u0:k20000:r10000
```

In order to report ownership to userspace the kernel uses the crossmapping algorithm introduced in a previous section:

1. Map the userspace id on disk down into a kernel id in the filesystem's idmapping:

```
make_kuid(u0:k20000:r10000, u1000) = k21000
```

2. Map the kernel id up into a userspace id in the caller's idmapping:

   ```
   from_kuid(u0:k10000:r10000, k21000) = u-1
   ```

Again, the crossmapping algorithm fails in this case because the kernel id in the filesystem idmapping cannot be mapped to a userspace id in the caller's idmapping. Thus, the kernel will report the ownership of this file as the overflowid.

Note how in the last two examples things would be simple if the caller would be using the initial idmapping. For a filesystem mounted with the initial idmapping it would be trivial. So we only consider a filesystem with an idmapping of `u0:k20000:r10000`:

1. Map the userspace id on disk down into a kernel id in the filesystem's idmapping:

   ```
   make_kuid(u0:k20000:r10000, u1000) = k21000
   ```

2. Map the kernel id up into a userspace id in the caller's idmapping:

   ```
   from_kuid(u0:k0:r4294967295, k21000) = u21000
   ```

Idmappings on idmapped mounts

The examples we've seen in the previous section where the caller's idmapping and the filesystem's idmapping are incompatible causes various issues for workloads. For a more complex but common example, consider two containers started on the host. To completely prevent the two containers from affecting each other, an administrator may often use different non-overlapping idmappings for the two containers:

```
container1 idmapping:  u0:k10000:r10000
container2 idmapping:  u0:k20000:r10000
filesystem idmapping:  u0:k30000:r10000
```

An administrator wanting to provide easy read-write access to the following set of files:

```
dir id:       u0
dir/file1 id: u1000
dir/file2 id: u2000
```

to both containers currently can't.

Of course the administrator has the option to recursively change ownership via `chown()`. For example, they could change ownership so that `dir` and all files below it can be crossmapped from the filesystem's into the container's idmapping. Let's assume they change ownership so it is compatible with the first container's idmapping:

```
dir id:       u10000
dir/file1 id: u11000
dir/file2 id: u12000
```

This would still leave `dir` rather useless to the second container. In fact, `dir` and all files below it would continue to appear owned by the overflowid for the second container.

Or consider another increasingly popular example. Some service managers such as systemd implement a concept called "portable home directories". A user may want to use their home directories on different machines where they are assigned different login userspace ids. Most users will have `u1000` as the login id on their machine at home and all files in their home directory will usually be owned by `u1000`. At uni or at work they may have another login id such as `u1125`. This makes it rather difficult to interact with their home directory on their work machine.

In both cases changing ownership recursively has grave implications. The most obvious one is that ownership is changed globally and permanently. In the home directory case this change in ownership would even need to happen everytime the user switches from their home to their work machine. For really large sets of files this becomes increasingly costly.

If the user is lucky, they are dealing with a filesystem that is mountable inside user namespaces. But this would also change ownership globally and the change in ownership is tied to the lifetime of the filesystem mount, i.e. the superblock. The only way to change ownership is to completely unmount the filesystem and mount it again in another user namespace. This is usually impossible because it would mean that all users currently accessing the filesystem can't anymore. And it means that `dir` still can't be shared between two containers with different idmappings. But usually the user doesn't even have this option since most filesystems aren't mountable inside containers. And not having them mountable might be desirable as it doesn't require the filesystem to deal with malicious filesystem images.

But the usecases mentioned above and more can be handled by idmapped mounts. They allow to expose the same set of dentries with different ownership at different mounts. This is achieved by marking the mounts with a user namespace through the `mount_setattr()` system call. The idmapping associated with it is then used to translate from the caller's idmapping to the filesystem's idmapping and vica versa using the remapping algorithm we introduced above.

Idmapped mounts make it possible to change ownership in a temporary and localized way. The ownership changes are restricted to a specific mount and the ownership changes are tied to the lifetime of the mount. All other users and locations where the filesystem is exposed are unaffected.

Filesystems that support idmapped mounts don't have any real reason to support being mountable inside user namespaces. A filesystem could be exposed completely under an idmapped mount to get the same effect. This has the advantage that filesystems can leave the creation of the superblock to privileged users in the initial user namespace.

However, it is perfectly possible to combine idmapped mounts with filesystems mountable inside user namespaces. We will touch on this further below.

Remapping helpers

Idmapping functions were added that translate between idmappings. They make use of the remapping algorithm we've introduced earlier. We're going to look at two:

- `i_uid_into_mnt()` and `i_gid_into_mnt()`

  The `i_*id_into_mnt()` functions translate filesystem's kernel ids into kernel ids in the mount's idmapping:

  ```
  /* Map the filesystem's kernel id up into a userspace id in the filesystem's idmapping. */
  from_kuid(filesystem, kid) = uid

  /* Map the filesystem's userspace id down ito a kernel id in the mount's idmapping. */
  make_kuid(mount, uid) = kuid
  ```

- `mapped_fsuid()` and `mapped_fsgid()`

  The `mapped_fs*id()` functions translate the caller's kernel ids into kernel ids in the filesystem's idmapping. This translation is achieved by remapping the caller's kernel ids using the mount's idmapping:

  ```
  /* Map the caller's kernel id up into a userspace id in the mount's idmapping. */
  from_kuid(mount, kid) = uid

  /* Map the mount's userspace id down into a kernel id in the filesystem's idmapping. */
  make_kuid(filesystem, uid) = kuid
  ```

Note that these two functions invert each other. Consider the following idmappings:

```
caller idmapping:     u0:k10000:r10000
filesystem idmapping: u0:k20000:r10000
mount idmapping:      u0:k10000:r10000
```

Assume a file owned by `u1000` is read from disk. The filesystem maps this id to `k21000` according to it's idmapping. This is what is stored in the inode's `i_uid` and `i_gid` fields.

When the caller queries the ownership of this file via `stat()` the kernel would usually simply use the crossmapping algorithm and map the filesystem's kernel id up to a userspace id in the caller's idmapping.

But when the caller is accessing the file on an idmapped mount the kernel will first call `i_uid_into_mnt()` thereby translating the filesystem's kernel id into a kernel id in the mount's idmapping:

```
i_uid_into_mnt(k21000):
  /* Map the filesystem's kernel id up into a userspace id. */
  from_kuid(u0:k20000:r10000, k21000) = u1000

  /* Map the filesystem's userspace id down ito a kernel id in the mount's idmapping. */
  make_kuid(u0:k10000:r10000, u1000) = k11000
```

Finally, when the kernel reports the owner to the caller it will turn the kernel id in the mount's idmapping into a userspace id in the caller's idmapping:

```
from_kuid(u0:k10000:r10000, k11000) = u1000
```

We can test whether this algorithm really works by verifying what happens when we create a new file. Let's say the user is creating a file with `u1000`.

The kernel maps this to `k11000` in the caller's idmapping. Usually the kernel would now apply the crossmapping, verifying that `k11000` can be mapped to a userspace id in the filesystem's idmapping. Since `k11000` can't be mapped up in the filesystem's idmapping directly this creation request fails.

But when the caller is accessing the file on an idmapped mount the kernel will first call `mapped_fs*id()` thereby translating the caller's kernel id into a kernel id according to the mount's idmapping:

```
mapped_fsuid(k11000):
   /* Map the caller's kernel id up into a userspace id in the mount's idmapping. */
   from_kuid(u0:k10000:r10000, k11000) = u1000

   /* Map the mount's userspace id down into a kernel id in the filesystem's idmapping. */
   make_kuid(u0:k20000:r10000, u1000) = k21000
```

When finally writing to disk the kernel will then map `k21000` up into a userspace id in the filesystem's idmapping:

```
from_kuid(u0:k20000:r10000, k21000) = u1000
```

As we can see, we end up with an invertible and therefore information preserving algorithm. A file created from `u1000` on an idmapped mount will also be reported as being owned by `u1000` and vica versa.

Let's now briefly reconsider the failing examples from earlier in the context of idmapped mounts.

Example 2 reconsidered

```
caller id:            u1000
caller idmapping:     u0:k10000:r10000
filesystem idmapping: u0:k20000:r10000
mount idmapping:      u0:k10000:r10000
```

When the caller is using a non-initial idmapping the common case is to attach the same idmapping to the mount. We now perform three steps:

1. Map the caller's userspace ids into kernel ids in the caller's idmapping:

```
make_kuid(u0:k10000:r10000, u1000) = k11000
```

2. Translate the caller's kernel id into a kernel id in the filesystem's idmapping:

```
mapped_fsuid(k11000):
  /* Map the kernel id up into a userspace id in the mount's idmapping. */
  from_kuid(u0:k10000:r10000, k11000) = u1000

  /* Map the userspace id down into a kernel id in the filesystem's idmapping. */
  make_kuid(u0:k20000:r10000, u1000) = k21000
```

2. Verify that the caller's kernel ids can be mapped to userspace ids in the filesystem's idmapping:

```
from_kuid(u0:k20000:r10000, k21000) = u1000
```

So the ownership that lands on disk will be `u1000`.

Example 3 reconsidered

```
caller id:            u1000
caller idmapping:     u0:k10000:r10000
filesystem idmapping: u0:k0:r4294967295
mount idmapping:      u0:k10000:r10000
```

The same translation algorithm works with the third example.

1. Map the caller's userspace ids into kernel ids in the caller's idmapping:

```
make_kuid(u0:k10000:r10000, u1000) = k11000
```

2. Translate the caller's kernel id into a kernel id in the filesystem's idmapping:

```
mapped_fsuid(k11000):
   /* Map the kernel id up into a userspace id in the mount's idmapping. */
   from_kuid(u0:k10000:r10000, k11000) = u1000

   /* Map the userspace id down into a kernel id in the filesystem's idmapping. */
   make_kuid(u0:k0:r4294967295, u1000) = k1000
```

2. Verify that the caller's kernel ids can be mapped to userspace ids in the filesystem's idmapping:

```
from_kuid(u0:k0:r4294967295, k21000) = u1000
```

So the ownership that lands on disk will be `u1000`.

Example 4 reconsidered

```
file id:              u1000
caller idmapping:     u0:k10000:r10000
filesystem idmapping: u0:k0:r4294967295
mount idmapping:      u0:k10000:r10000
```

In order to report ownership to userspace the kernel now does three steps using the translation algorithm we introduced earlier:

1. Map the userspace id on disk down into a kernel id in the filesystem's idmapping:

```
make_kuid(u0:k0:r4294967295, u1000) = k1000
```

2. Translate the kernel id into a kernel id in the mount's idmapping:

```
i_uid_into_mnt(k1000):
  /* Map the kernel id up into a userspace id in the filesystem's idmapping. */
  from_kuid(u0:k0:r4294967295, k1000) = u1000
  /* Map the userspace id down into a kernel id in the mounts's idmapping. */
  make_kuid(u0:k10000:r10000, u1000) = k11000
```

3. Map the kernel id up into a userspace id in the caller's idmapping:

```
from_kuid(u0:k10000:r10000, k11000) = u1000
```

Earlier, the caller's kernel id couldn't be crossmapped in the filesystems's idmapping. With the idmapped mount in place it now can be crossmapped into the filesystem's idmapping via the mount's idmapping. The file will now be created with `u1000` according to the mount's idmapping.

Example 5 reconsidered

```
file id:               u1000
caller idmapping:      u0:k10000:r10000
filesystem idmapping: u0:k20000:r10000
mount idmapping:       u0:k10000:r10000
```

Again, in order to report ownership to userspace the kernel now does three steps using the translation algorithm we introduced earlier:

1. Map the userspace id on disk down into a kernel id in the filesystem's idmapping:

```
make_kuid(u0:k20000:r10000, u1000) = k21000
```

2. Translate the kernel id into a kernel id in the mount's idmapping:

```
i_uid_into_mnt(k21000):
  /* Map the kernel id up into a userspace id in the filesystem's idmapping. */
  from_kuid(u0:k20000:r10000, k21000) = u1000

  /* Map the userspace id down into a kernel id in the mounts's idmapping. */
  make_kuid(u0:k10000:r10000, u1000) = k11000
```

3. Map the kernel id up into a userspace id in the caller's idmapping:

```
from_kuid(u0:k10000:r10000, k11000) = u1000
```

Earlier, the file's kernel id couldn't be crossmapped in the filesystems's idmapping. With the idmapped mount in place it now can be crossmapped into the filesystem's idmapping via the mount's idmapping. The file is now owned by `u1000` according to the mount's idmapping.

Changing ownership on a home directory

We've seen above how idmapped mounts can be used to translate between idmappings when either the caller, the filesystem or both uses a non-initial idmapping. A wide range of usecases exist when the caller is using a non-initial idmapping. This mostly happens in the context of containerized workloads. The consequence is as we have seen that for both, filesystem's mounted with the initial idmapping and filesystems mounted with non-initial idmappings, access to the filesystem isn't working because the kernel ids can't be crossmapped between the caller's and the filesystem's idmapping.

As we've seen above idmapped mounts provide a solution to this by remapping the caller's or filesystem's idmapping according to the mount's idmapping.

Aside from containerized workloads, idmapped mounts have the advantage that they also work when both the caller and the filesystem use the initial idmapping which means users on the host can change the ownership of directories and files on a per-mount basis.

Consider our previous example where a user has their home directory on portable storage. At home they have id `u1000` and all files in their home directory are owned by `u1000` whereas at uni or work they have login id `u1125`.

Taking their home directory with them becomes problematic. They can't easily access their files, they might not be able to write to disk without applying lax permissions or ACLs and even if they can, they will end up with an annoying mix of files and directories owned by `u1000` and `u1125`.

Idmapped mounts allow to solve this problem. A user can create an idmapped mount for their home directory on their work computer or their computer at home depending on what ownership they would prefer to end up on the portable storage itself.

Let's assume they want all files on disk to belong to `u1000`. When the user plugs in their portable storage at their work station they can setup a job that creates an idmapped mount with the minimal idmapping `u1000:k1125:r1`. So now when they create a file the kernel performs the following steps we already know from above::

```
caller id:             u1125
caller idmapping:      u0:k0:r4294967295
filesystem idmapping: u0:k0:r4294967295
mount idmapping:       u1000:k1125:r1
```

1. Map the caller's userspace ids into kernel ids in the caller's idmapping:

```
make_kuid(u0:k0:r4294967295, u1125) = k1125
```

2. Translate the caller's kernel id into a kernel id in the filesystem's idmapping:

   ```
   mapped_fsuid(k1125):
     /* Map the kernel id up into a userspace id in the mount's idmapping. */
     from_kuid(u1000:k1125:r1, k1125) = u1000

     /* Map the userspace id down into a kernel id in the filesystem's idmapping. */
     make_kuid(u0:k0:r4294967295, u1000) = k1000
   ```

2. Verify that the caller's kernel ids can be mapped to userspace ids in the filesystem's idmapping:

   ```
   from_kuid(u0:k0:r4294967295, k1000) = u1000
   ```

So ultimately the file will be created with `u1000` on disk.

Now let's briefly look at what ownership the caller with id `u1125` will see on their work computer:

```
file id:               u1000
caller idmapping:      u0:k0:r4294967295
filesystem idmapping: u0:k0:r4294967295
mount idmapping:       u1000:k1125:r1
```

1. Map the userspace id on disk down into a kernel id in the filesystem's idmapping:

   ```
   make_kuid(u0:k0:r4294967295, u1000) = k1000
   ```

2. Translate the kernel id into a kernel id in the mount's idmapping:

   ```
   i_uid_into_mnt(k1000):
     /* Map the kernel id up into a userspace id in the filesystem's idmapping. */
     from_kuid(u0:k0:r4294967295, k1000) = u1000

     /* Map the userspace id down into a kernel id in the mounts's idmapping. */
     make_kuid(u1000:k1125:r1, u1000) = k1125
   ```

3. Map the kernel id up into a userspace id in the caller's idmapping:

   ```
   from_kuid(u0:k0:r4294967295, k1125) = u1125
   ```

So ultimately the caller will be reported that the file belongs to `u1125` which is the caller's userspace id on their workstation in our example.

The raw userspace id that is put on disk is `u1000` so when the user takes their home directory back to their home computer where they are assigned `u1000` using the initial idmapping and mount the filesystem with the initial idmapping they will see all those files owned by `u1000`.