Read-only FS abstractions & tarfs

Wedson Almeida Filho, Microsoft
Kangrejos, 2023/09/17

Motivation

- Confidential containers product
  - Kubernetes orchestration
  - containerd
  - kata containers
  - AMD SEV-SNP and Intel TDX CPUs
- Requirements
  - Resource sharing
  - Image integrity
  - "Lift and shift" existing workloads
    - Existing Open Container Initiative (OCI) images
    - Existing signature of OCI images

Solution using tarfs

- Containerd snapshotter
  - Downloads OCI image layers (typically tar.gz files)
  - Decompresses them
  - Appends "tar" index
  - Appends dm-verity merkle tree
- Containerd kata runtime
  - Hotplugs layers to VMs as read-only block devices
- Containerd kata agent (running in VM)
  - Creates dm-verity volumes from block devices
  - Verifies signature of tar volumes
  - Verifies indexes are valid
  - Mounts dm-verity volumes using tarfs
  - Overlays all tarfs layers with overlayfs

Read-only fs abstractions

- Only adding what's needed to implement a read-only fs
  - This avoids issues with upstreaming not-yet-used code
- dcache is hidden from modules
  - Because there is only one interaction with dentries that can be easily absorbed
- Used by tarfs and puzzlefs (which Ariel will talk about next)
- Goal: zero-cost, safe abstractions
  - If modules don't use unsafe blocks, they don't get undefined behaviour
  - No additional runtime cost when compared to C implementation
- Tarfs is used as sample code in slides
  - Code available here: https://github.com/wedsonaf/linux/tree/rtarfs

Read-only fs module

```
//! File system based on tar files and an index.
use kernel::{ /* ... */ };

kernel::module_ro_fs! {
    type: TarFs,
    name: "tarfs",
    author: "Wedson Almeida Filho <walmeida@microsoft.com>",
    license: "GPL",
}

struct TarFs { /* ... */ }

impl fs::ro::Type for TarFs {
    type Data = Box<Self>;
    type INodeData = INodeData;
    const NAME: &'static CStr = c_str!("tar");
    const SUPER_TYPE: Super = Super::BlockDev;
    /* ... */
}
```

Read-only fs trait

```rust
/// A read-only file system type.
pub trait Type {
    /// Data associated with each file system instance (super-block).
    type Data: ForeignOwnable + Send + Sync;

    /// Type of data allocated for each inode.
    type INodeData: Send + Sync;

    /// The name of the file system type.
    const NAME: &'static CStr;

    /// Determines how superblocks for this file system type are keyed.
    const SUPER_TYPE: Super = Super::Independent;
```

Read-only fs trait (cont'd)

```
    /// Initialises a super block for this file system type.
    fn fill_super(sb: NewSuperBlock<'_, Self>) -> Result<&SuperBlock<Self>>;

    /// Reads directory entries.
    fn read_dir(
        inode: &INode<Self>,
        pos: i64,
        report: impl FnMut(&[u8], i64, u64, DirEntryType) -> bool,
    ) -> Result<i64>;

    /// Looks up an entry with the given under the given parent inode.
    fn lookup(parent: &INode<Self>, name: &[u8]) -> Result<ARef<INode<Self>>>;

    /// Reads the contents of the inode into the given folio.
    fn read_folio(inode: &INode<Self>, folio: crate::folio::LockedFolio<'_>) -> Result;
}
```

Going from `NewSuperBlock` to `SuperBlock`

- `NewSuperBlock` has the following type states:
  - NeedsInit
  - NeedsData
  - NeedsRoot
- Each type state has the following exclusive method:
  - init
  - init_data
  - init_root

Going from `NewSuperBlock` to `SuperBlock` (cont'd)

```
fn fill_super(sb: NewSuperBlock<'_, Self>) -> Result<&SuperBlock<Self>> {
    let sb = sb.init(&SuperParams { blocksize_bits: TARFS_BSIZE_BITS, /* ... */ })?;
    /* ... */
    let tarfs = {
        let h = sb.bread(count * SECTOR_SIZE / TARFS_BSIZE - 1)?;
        /* ... */
        Box::try_new(TarFs { /* ... */ })?
    };
    let sb = sb.init_data(tarfs)?;
    let root = Self::iget(&sb, 1)?;
    sb.init_root(root)
}
```

`bread` without the block size results in a division by zero. So it's not available in `NeedsInit` typestate, but it's available in `NeedsData`.

This creates an inode, which gives access to the super-block data, so it needs to be initialised before we can allow the creation of inodes.

fill_super is supposed to initialise the root once before returning successfully. This prevents it from succeeding without setting the root.

Creating an inode

- If it already exists in the cache, just use it
  - Since this is a read-only fs, it's not stale
- If it doesn't exist yet, create a new "under-construction" one
- If we succeed in initialising it
  - Remove the "under-construction" flag
  - Wake up any threads waiting for this inode
- If we fail in initialising it
  - Mark the "under-construction" inode as bad
  - Wake up any threads waiting for this inode
  - Decrement the refcount

Creating an inode (cont'd)

We get an existing, initialised `INode`, or a new one that needs to be initialised.

```
impl<T: Type + ?Sized> SuperBlock<T> {
    fn get_or_create_inode(&self, ino: u64) -> Result<Either<ARef<INode<T>>, NewINode<T>>>;
}

impl TarFs {
    fn iget(sb: &SuperBlock<Self>, ino: u64) -> Result<ARef<INode<Self>>> {
        /* ... */
        // Create an inode or find an existing (cached) one.
        let inode = match sb.get_or_create_inode(ino)? {
            Either::Left(existing) => return Ok(existing),
            Either::Right(new) => new,
        };
        /* ... */
        inode.init(INodeParams { /* ... */ })
    }
}
```

If `init` or any previous step fails, the drop implementation of `NewINode` cleans it up. On success, `init` transitions the inode out of the "under-construction" state.

Locked folio in `read_folio` function

- When the operation completes, the callee must unlock the folio
- In Rust, we do that automatically in the `Drop` implementation
  - Callees may choose to keep it alive by not dropping it
- In C, there is no indication that the folio is locked
  - But one must ensure that all exit paths unlock the folio

Bug in romfs

```c
static int romfs_read_folio(struct file *file, struct folio *folio)
{
        struct page *page = &folio->page;

        /* ... */
        if (!buf)
                return -ENOMEM;

        /* ... */
        unlock_page(page);
        return ret;
}
```

No indication that the folio is locked.

No unlocking of the folio (page in this case) on the early return.

Accessing super block context data

- In C, it's via the `struct super_block::s_fs_info` field
  - Its type is `void *`
- In Rust, it's via `SuperBlock<T>::data()`
  - Its type is dependent on `T::Data`
- Example:

```
impl TarFs {
    fn iget(sb: &SuperBlock<Self>, ino: u64) -> Result<ARef<INode<Self>>> {
        // Check that the inode number is valid.
        let h = sb.data();
        if ino == 0 || ino > h.inode_count {
            return Err(ENOENT);
        }

        // ...
    }
}
```

Accessing inode context data

- In C, it's via the `container_of` macro from a `struct inode *`
- In Rust, it's via `INode<T>::data()`
    - Its type is dependent on `T::INodeData`
- Example:

```
impl TarFs {
    fn lookup(parent: &INode<Self>, name: &[u8]) -> Result<ARef<INode<Self>>> {
        // ...
        for v in sb.read(parent.data().offset, parent.size().try_into()?)? {
            // ...
        }
        // ...
    }
}
```

Super block init and cleanup matching

- Variant of the get_tree used in init dictates variant of kill to use on cleanup
- Examples:
    - get_tree_bdev and kill_block_super
    - get_tree_nodev and kill_anon_super
- In C, there is no compiler enforcement of this requirement
- In Rust, `T::SUPER_TYPE` defines what to use
    - Since it's a compile-time constant, all branches based on it are eliminated at compile time
    - There is no way (without unsafe blocks) to mismatch the functions
    - Reading from a block device also fails if the type doesn't have a bdev
        - This check is elided when it does

inode context data

- Attaching data to an inode:
  - Allocate a bigger struct with an embedded `struct inode`
  - When freeing, use `container_of` to go from inner struct inode to outer struct
- File systems that do this also allocate a `kmem_cache`
  - Needs to be freed when the file system is unregistered
- In C, this is manual and repeated in all file systems
- In Rust, we have an implementation parametrised on `T::INodeData`
  - As efficient as the C version when monomorphised, but type safe

Reading from block device

- When using bread, the cache is used to read blocks
- The block heads are ref-counted
  - In Rust, it uses the ARef abstraction, so the ref-counting is guaranteed to be safe
- When accessing the data, lifetimes enforce that the data access doesn't outlive the refcount
  - It's not possible to access the contents of a block with we release the refcount
- These enforcements are all done at compile-time: no runtime cost

Iterating over blocks in a block device

- When we need to read a range of bytes from a block device
  - That isn't necessarily block aligned
  - That may span multiple blocks
- Occurs several times in tarfs
- To avoid code duplication, In Rust we provide an iterator

```
/// Reads `size` bytes starting from `offset` bytes.
///
/// Returns an iterator that returns slices based on blocks.
pub fn read(
    &self,
    offset: u64,
    size: u64,
) -> Result<impl Iterator<Item = Result<super::buffer::View>> + '_>;
```

Iterating over blocks in a block device: example

```
fn name_eq(sb: &SuperBlock<Self>, mut name: &[u8], offset: u64) -> Result<bool> {
    for v in sb.read(offset, name.len().try_into()?)? {
        let v = v?;
        let b = v.data();
        if b != &name[..b.len()] {
            return Ok(false);
        }
        name = &name[b.len()..];
    }
    Ok(true)
}
```

Thank you!

Questions?