If you want to use RCU in Rust...

Boqun Feng, Self

Paul E. McKenney, Meta Platforms Kernel Team

... what you can do?

- Just call a C function:
  - If there is already a C function doing the heavy-lifting including accesses to RCU protect data, you can just call the C function
- Also you can always open-unsafe-code RCU primitives in a Rust function

```
fn some_func() {
  unsafe { bindings::rcu_read_lock(); }
  ...
  unsafe { bindings::rcu_read_unlock(); }
}
```

- Or use a Rust version RCU API

RCU core APIs

- `rcu_read_lock()` and `rcu_read_unlock()`
- `rcu_dereference()`
- `rcu_assign_pointer()`
- `synchronize_rcu()` and `call_rcu()`

RCU core APIs

`rcu_read_lock()`

```
/// RAII struct representing holding a RCU read lock.
struct RcuGuard {
    _not_send: PhantomData<*mut ()>,
}

impl RcuGuard {
    pub fn new() -> Self {
        unsafe { bindings::rcu_read_lock(); }
        Self {
            _not_send: PhantomData,
        }
    }
}
```

RCU core APIs (cont.)

`rcu_read_unlock()`

```
impl Drop for RcuGuard {
    fn drop(&mut self) {
        unsafe { bindings::rcu_read_unlock(); }
    }
}
```

RCU core APIs (cont.)

`rcu_dereference()`

```
/// # Invariants: `ptr` must be
/// * either null
/// * or a valid pointer to T and under RCU grace period protection.
struct UnsafeRcu<T> {
    ptr: AtomicPtr<T>
}

impl<T> UnsafeRcu<T> {
    pub fn dereference<'a, 'b>(&'a self, _guard: &'b RcuGuard) -> Option<&'b T> {
        let p = self.ptr.load(Relaxed);
        if p.is_null() {
            return None;
        }

        // For the address dependency ordering
        fence(Acquire);

        // SAFETY: `&RcuGuard` must outlive `&T`, so the reference is
        // valid under RCU Guarantee
        Some(unsafe { &* p })
    }
}
```

RCU core APIs (cont.)

`rcu_assign_pointer`

```
impl<T> UnsafeRcu<T> {
    /// # Safety
    /// `ptr` needs to fulfill the invariants of `UnsafeRcu`.
    pub unsafe fn set(&self, ptr: *mut T) {
        self.ptr.store(ptr, Release);
    }

    pub fn load(&self) -> *mut T {
        self.ptr.load(Relaxed)
    }

    pub unsafe swap(&self, ptr: *mut T) -> *mut T {
        self.ptr.swap(ptr, AcqRel) // xchg
    }
}
```

Existing RCU use cases: an incomplete list

- Quasi reader-writer lock.
- Quasi reference count.
- Quasi multi-version consistency control.
- Light-weight garbage collector.
- Delete-only list (relatively rare)..
- Add-only list (relatively rare)..
- Type-safe memory (relatively rare)..
- Existence guarantee.
- Phased state change (relatively rare)..

Design/Requirement dimensions

1. Read-side permissions for non-pointer fields (and non-RCU pointers):
    - (a) None.
    - (b) Read-only.
    - (c) Read-write. (For example, counting rare read-side error conditions.)
        - Interior mutability for the win?
    - (d) One of the above on a per-field basis.

Design/Requirement dimensions

2. Update-side permissions for non-pointer fields (and non-RCU pointers):
    - (a) None.
    - (b) Read-only.
    - (c) Read-write.
    - (d) One of the above on a per-field basis.

Design/Requirement dimensions

3. Read-side permissions for pointer fields linking RCU-protected data:
    - (a) None. (But how is this useful given no way to traverse the structure?)
    - (b) Read-only, which is the common case.
    - (c) Read-write, enabling reader traversal to fine-grained in-structure update.
    - (d) One of the above on a per-field basis, supporting multilinked structures.

Design/Requirement dimensions

4. Update-side permissions for pointer fields linking RCU-protected data:
    - (a) None. (But how is this useful given no way to traverse the structure?)
    - (b) Read-only, which enables updating with full-structure replacement.
    - (c) Read-write, enabling updater traversal to fine-grained in-structure update.
    - (d) One of the above on a per-field basis, supporting multilinked structures.

Design/Requirement dimensions

5. Changing update-side permissions when restricted by readers:
    - (a) Manually, for example, using unsafe blocks.
    - (b) Automatic for insertion, e.g., `p2 = rcu_assign_pointer(gp, p1)`.
    - (c) Automatic for removal, e.g., `call_rcu()` updates permission. `synchronize_rcu` () would needs pointers in and out?
    - (d) Update-side function to be executed during reader accessibility. Seems uselessly restrictive to me, but failure of imagination on my part?

Design/Requirement dimensions

6. C-language interoperability:
    - (a) Rust-only (easy case!).
    - (b) C-only (easy case!)
    - (c) Rust read-only, C update-only.
    - (d) Rust read-only, C readers and updaters. (Practical difference from #c?)
    - (e) C read-only, Rust update-only.
    - (f) C read-only, Rust readers and updaters. (Practical difference from #e?)
    - (g) Rust readers and updaters with C updaters.
    - (h) Rust updaters and C readers and updaters. (Practical difference from #g?)
    - (i) Rust and C both read and update

Rust/C RCU Interoperability Table (1/4)

Rust		C		
Read	Update	Read	Update	Commentary
N	N	N	N	Pointless ;-)
N	N	N	Y	Not RCU.
N	N	Y	N	No synchronization needed
N	N	Y	Y	C-only (6b)

Rust/C RCU Interoperability Table (2/4)

Rust		C		
Read	Update	Read	Update	Commentary
N	Y	N	N	Not RCU.
N	Y	N	Y	Not RCU.
N	Y	Y	N	Rust updaters, C readers (6e).
N	Y	Y	Y	Rust updaters, C readers and updaters (6h).

Rust/C RCU Interoperability Table (3/4)

Rust		C		
Read	Update	Read	Update	Commentary
Y	N	N	N	No synchronization needed
Y	N	N	Y	Rust readers, C updaters (6c).
Y	N	Y	N	No synchronization needed
Y	N	Y	Y	Rust readers, C readers and updaters (6d).

Rust/C RCU Interoperability Table (4/4)

Rust		C		
Read	Update	Read	Update	Commentary
Y	Y	N	N	Rust-only (6a).
Y	Y	N	Y	Rust readers and updaters, C update (6g).
Y	Y	Y	N	Rust readers and updaters, C readers (6f).
Y	Y	Y	Y	Both Rust and C readers and updaters (6i).

More Dimensions???

Probably!

Especially if we decide to attempt automatic conversion from RCU-protected C data structures to RCU-protected Rust data structures, which we might want to do in order to avoid Rust/C skew.

Case studies #1: Read (Copy) Update

```
// struct config {
//   int x;
//   int y;
//   int z;
// }
struct Config {
    x: i32,
    y: i32,
    z: i32,
}
```

```
// void update(struct config __rcu **config, int x, int y, int z)
pub fn update(cfg: &UnsafeRCU<Config>, x: i32, y: i32, z: i32);

// (int, int, int) read(struct config __rcu **config)
pub fn read(cfg: &UnsafeRcu<Config>) -> (i32, i32, i32);
```

Two-level pointer to simulate a global pointer.

Case studies #1: Read (Copy) Update

reader side

```
pub fn read(config: &UnsafeRcu<Config>) -> Option<(i32, i32, i32)> {
    // rcu_read_lock();
    let g = RcuGuard::new();

    // struct config *cfg = rcu_dereference(*config);
    let cfg = cfg.dereference(&g)?;

    (cfg.x, cfg.y, cfg.z)
    // drop(g): rcu_read_unlock();
}
```

Case studies #1: Read (Copy) Update

updater side?

```
pub fn update(config: &UnsafeRCU<Config>, x: i32, y: i32, z: i32) {
    // struct config *new = your_alloc_config(x, y, z);
    let new: *mut Config = your_alloc_config(x, y, z);

    // struct config *old = rcu_access_pointer(*config);
    let old = cfg.load();

    // rcu_assign_pointer(*config, new);
    // SAFETY: `update` is the only function updates `config`, and it will
    // call a synchronize_rcu() before release `old`
    unsafe { cfg.set(new) }

    // SAFETY: Worst case is deadlock, so it's still safe?
    unsafe { bindings::synchronize_rcu(); }

    your_free_config(old);
}
```

Case studies #1: Read (Copy) Update

updater side, need atomic update, otherwise maybe double free.

```
pub fn update(config: &UnsafeRCU<Config>, x: i32, y: i32, z: i32) {
    // struct config *new = your_alloc_config(x, y, z);
    let new: *mut Config = your_alloc_config(x, y, z);

    // struct config *old = xchg_acqrel(*config, new);
    // SAFETY: `update` is the only function updates `config`, and it will
    // call a synchronize_rcu() before release `old`, also `old` will be
    // only freed by here, no double free.
    let old = unsafe { cfg.swap(new) };

    // SAFETY: Worst case is deadlock, so it's still safe?
    unsafe { bindings::synchronize_rcu(); }

    your_free_config(old);
}
```

Case studies #1: Read (Copy) Update

- `your_alloc_config()` and `your_free_config()` can be implemented by `kmalloc/kfree`
- also `your_free_config()` can be implemented by `kfree_rcu`, no need to call `synchronize_rcu` then.

Case studies #1: Read (Copy) Update

Generify with `Box`

```
pub fn update<T>(data: &UnsafeRcu<T>, new_box: Box<T>) {
    let new: *mut T = Box::into_raw(new_box);

    let old = unsafe { data.swap(new) };

    if !old.is_null() {
        unsafe { bindings::synchronize_rcu(); }

        // SAFETY: `old` was previously set by the update function, so it comes
        // from a previous `Box::into_raw`, and since we called `synchronize_rcu`
        // after `swap`, no reader is referencing the old data now.
        drop(unsafe { Box::from_raw(old) });
    }

    Ok(())
}
```

Case studies #1: Read Copy Update

```
pub fn copy_update<T>(data: &UnsafeRcu<T>, update_fn: fn(&mut T)) { ... }
```

Example:

```
copy_update(&some_config, |cfg: &mut Config| {
    // cfg is a copy of the old data
    cfg.x = 12
})
```

Case studies #1: Read Copy Update

```
pub fn copy_update<T: Copy>(data: &UnsafeRcu<T>, update_fn: fn(&mut T)) {
    <lock>
    let old = data.load();

    <handle the old == null and return>
    // Requires T: Copy
    let mut new_box: Box<T> = Box::try_new(unsafe {*old})?;

    update_fn(&mut new_box);

    unsafe { data.set(Box::into_raw(new_box)); }
    <unlock>
    <synchronize_rcu or call_rcu>
}
```

```
#[derive(Copy, Clone)]
struct Config { ... }
```

Case studies #1: Read Copy Update

How can we fill the `<lock>` and `<unlock>` part?

- Big RCU updater lock!
- Make `copy_update` an `&mut` function?

```
pub fn copy_update<T: Copy>(data: &mut UnsafeRcu<T>, update_fn: fn(&mut T)) { ... }
```

(but this in theory doesn't work, since updater shouldn't block readers in RCU)
(it's unsafe that `&mut` and `&` co-exist)

- Make `copy_update` an `unsafe` function

```
pub unsafe fn copy_update<T: Copy>(data: &UnsafeRcu<T>, update_fn: fn(&mut T)) { ... }
```

Case studies #1: Read Copy Update

Put it together, we have a safer API

```
impl<T> BoxRcu<T> {
    pub fn update(&self, new: Box<T>) { ... }
    pub fn dereference<'rcu>(&self, &'rcu RcuGuard) -> &'rcu T { ... }
    // Needs to wait for a grace period
    pub unsafe fn swap(&self, new: Box<T>) -> Option<Box<T>> { ... }
    // Needs to wait for a grace period and ensure exclusive among updaters
    pub unsafe fn replace(&self, new: Box<T>) -> Option<Box<T>> { ...}
}

impl<T: Copy> BoxRcu<T> {
    // safe is Big RCU updater lock is used.
    pub fn copy_update(&self, update_fn: fn(&mut T)) { ... }
}
```

Case studies #2: Fine-grained lock

```
// struct foo {
//    int a;
//    int b;
//    struct config __rcu *cfg;
// }
struct Foo {
    a: i32,
    b: i32,
    cfg: BoxRcu<Config>,
}
```

Case studies #2: Fine-grained lock

Where is the lock?

```
// struct foo_locked {
//   spinlock_t lock;
//   struct foo foo;
// }
impl SpinLock<Foo> {
    pub fn lock(&self) -> SpinLockGuard<'_, Foo> { <lock> }
}

impl<'a> Drop for SpinLockGuard<'a, Foo> {
    fn drop(&mut self) { <unlock> }
}

impl<'a> DerefMut for SpinLockGuard<'a, Foo> {
    fn deref_mut(&mut self) -> &mut Foo { ... }
}
```

Case studies #2: Fine-grained lock

How would updaters access `Foo::cfg`?

```
let some_foo: &SpinLock<Foo> = ...
let guard: SpinLockGuard<...> = some_foo.lock();
let foo: &mut Foo = guard.deref_mut();
// Exclusive accesses until foo's life ends.
<readers in the same time?>
```

`&mut Foo` is required to access `Foo::a` and `Foo::b`, but `Foo::cfg` should be able to be accessed by readers and updaters at the same time!

Case studies #2: Fine-grained lock

A quick solution (regrouping the fields)

```
struct FooLocked {
    a: i32,
    b: i32,
}

struct Foo {
    locked: SpinLock<FooLocked>,
    cfg: BoxRcu<Config>,
}
```

It works, but is not flexible, also `unsafe` is still needed for updating `Foo::cfg`.

Case studies #2: Fine-grained lock

An RCU-reader-aware lock?

```
impl SpinAndRcu<Foo> {
    pub fn updater(&self) -> Updater<'_, Foo> { ... }
    pub fn get_cfg(&self) -> &BoxRcu<Config> { ... }
}

impl<'a> Updater<'a, Foo> {
    pub fn mut_a(&mut self) -> &mut i32 { ... }
    pub fn mut_b(&mut self) -> &mut i32 { ... }
    pub fn update_cfg(&mut self, new: Box<Config>) { ... }
    pub fn copy_update_cfg(&mut self, update_fn: fn(&mut Config)) { ... }
}
```

Works but not a generic solution.

Case studies #2: Fine-grained lock

A generic solution would be:

```
pub fn deref(updater: &Updater<'a, T>) -> &T
```

- $\forall f$: `F` as a normal (exclusively writable) field of `T`, we have

```
pub fn ???(updater: &mut Updater<'a, T>) -> &mut F
```

- $\forall r$: `BoxRcu<F>` s a RCU field of `T`, we have

```
pub fn ???(lock: &SpinAndRcu<T>) -> &BoxRcu<F>
```

- $\forall r$: `BoxRcu<F>` s a RCU field of `T`, we also have

```
pub fn ???(updater: &mut Updater<T>, new: Box<F>)
```

Case studies #2: Fine-grained lock

Define a field of a struct

```
pub unsafe trait Field<Base: ?Sized> {
    /// The type of the field.
    type Type: ?Sized;

    /// The name of the field.
    const NAME: &'static str;

    /// Adjust the pointer from the containing type.
    ///
    /// # Safety
    ///
    /// `base` must be a non-null and aligned pointer to [`Self::Base`].
    unsafe fn field_of(base: *const Base) -> *const Self::Type;
}
```

Case studies #2: Fine-grained lock

For example, define this for `Foo::a`

```
unsafe impl Field<Foo> for Foo::a {
    type Type = i32;
    cost NAME: &'static str = "a";

    unsafe fn field_of(base: *const Foo) -> *const Self::Type {
        // &foo->a
        addr_of!(unsafe { * base }.a)
    }
}
```

Case studies #2: Fine-grained lock

How to get these `impl`s **automatically**?

"Use the field projection, Luke!"

- Library work by Gary Guo
- Language/compiler proposal by Benno Lossin

Case studies #2: Fine-grained lock

Mutable accesses to lock-protected fields

```
impl<'a, T> Updater<'a, T> {
    /// Accesses a non-RCU field mutably.
    pub fn project_mut<F>(&mut self) -> &mut <F as Field<T>>::Type
    where
        F: FieldMut<T>,
    {
        // SAFETY: `self.lock.data.get()` is non-null and aligned to `T`.
        let ptr = unsafe {
            <F as Field<T>>::field_of(self.guard.lock.data.get())
        };

        // SAFETY: We are holding the lock,
        // so we have the mutable accessibilty to the field.
        unsafe { &mut *(ptr as *mut _) }
    }
}
```

```
let updater: &mut Updater<'a, Foo> = ...;

// foo->a = 1
*(updater.project_mut::<Foo::a>()) = 1;
```

Case studies #2: Fine-grained lock

Lockless accesses to RCU fields

```
impl<T> SpinAndRcu<T> {
    pub fn project<F, S>(&self) -> &BoxRcu<S>
    where
        F: Field<T, Type = BoxRcu<S>>,
    {
        // SAFETY: `self.lock.data.get()` is non-null and aligned to `T`.
        let ptr = unsafe { F::field_of(self.lock.data.get()) };

        unsafe { &*ptr }
    }
}
```

```
let foo: &SpinAndRcu<Foo> = ...;

// rcu_read_lock();
// int x = rcu_dereference(foo->cfg)->x;
let x = foo.project::<Foo::cfg, Config>().dereference(&RcuGuard::new()).x;
```

Case studies #2: Fine-grained lock

Update RCU fields with lock held

```
impl<'a, T> Updater<'a, T> {
    pub fn update<F, S>(&self, new: Box<S>)
    where
        F: Field<T, Type = BoxRcu<S>>,
    {
        // SAFETY: `self.lock.data.get()` is non-null and aligned to `T`.
        let ptr = unsafe { F::field_of(self.guard.lock.data.get()) };

        // SAFETY: `T` owns the RCU field, and with `Guard` of `T` means exclusive accesses among
        // updaters.
        let old = unsafe { (*ptr).unsafe_swap(Box::into_raw(new)) }

        <synchronize_rcu() and free old>
    }
}
```

Case studies #2: Fine-grained lock

usage

```
let foo :&SpinAndRcu<foo> = ...;
// spin_lock(&foo->lock);
let mut updater: Updater<'a, Foo> = foo.updater();
// struct config *new = kmalloc(...);
// other initialization
let mut new: Box<Config> = ...;
// new->x = foo->cfg->x;
new.x = updater.cfg.x;
// struct config *old = foo->cfg;
// rcu_assign_pointer(foo->cfg, new);
// free_rcu(old);
updater.update::<Foo::cfg, Config>(new);
// spin_unlock(&foo->lock);
```

Summary

- `UnsafeRcu` : 1b, 2a, 3c, 4c, 5b-, 6i
- `BoxRcu` : 1b, 2a, 3c, 4c, 5c- (5c if using Big RCU updater side lock)
  - 6a is regrouping is needed
  - 6i otherwise
- Field projection + `Rcu` : 1d, 2d, 3d, 4d, 5c, 6i

Future work

- Build the proper type hierarchy to make `call_rcu()` work
- Explore more use cases (linked list, etc.)

Acknowledgments

- The original idea of using field projection for RCU was based on discussion in Rust-for-Linux meetings/zulip, all glory to the team!
- `rcu_read_lock()` implementation is from Wedson ;-)
- Special thanks to: Paul E. McKenney, Neeraj Upadhyay, Frederic Weisbecker, Uladzislau Rezki and Joel Fernandes for a rehearsal of the Rust RCU API.