Atomics and memory model for Rust code in kernel

Boqun Feng 冯博群 (Microsoft)

Summary on previous discussions

- Use Linux Kernel Memory Model for Rust code in kernel.
  - Atomics and barriers will be implemented by FFI to C functions (or `asm!()`)
  - Rust code can rely on dependency orderings.
    - Collaboration with Rust community/ctrl_dep (a POC from Gary Guo)/code guideline.

Data races

- Normal read vs. atomic read shouldn't be treated as data races.
- If C side relies on `CONFIG_KCSAN_ASSUME_PLAIN_WRITES_ATOMIC=y`, they should be treated as atomic.

```
int r1 = *x; // normal read A
*y = 1; // normal write B
```

```
let r2 = unsafe { AtomicI32::from_ptr(y)}.read(); // C
let x_ref = unsafe { AtomicI32::from_ptr(x)};
let r3 = x_ref.read(); // D
x_ref.set(1); // E
```

Data races: A + E

Not data races: A + D, B + C

RFC

```
impl AtomicI32 {
    pub fn fetch_add(&self, v: i32) -> i32 { ... }
}

// users
let x: &AtomicI32 = ...;
x.fetch_add_relaxed(2);
```

Generic API: `Atomic<T>`

- `T` can be only from a list of types
  - `i32` / `u32` / `i64` / `u64` / `isize` / `usize` / `*mut T`
  - "New type" of basic types.

```
// user
let x: &Atomic<i32> = ...;
x.fetch_add_relaxed(2);

// new type
#[repr(transparent)]
struct NewType(u32);
unsafe impl AllowAtomic for NewType { ... }
```

Ordering parameter

- both a function parameter and a generic type parameter

```
impl<T: ...> Atomic<T> {
    pub fn read<O: AcquireOrRelaxed>(&self, o: O) -> T { ... }
    pub fn fetch_add<O: All>(&self, i: T, o: O) -> T { ... }
}

// user
let x: &Atomic<i32> = ...;

let r1 = x.read(Acquire);
let r2 = x.fetch_add(2, Release);
let r2 = x.read(Full); // <- compile error
```

Q & A

Backup slides

data dependency

```
int r = atomic_read(x);
atomic_set(y, r);
```

address dependency

```
atomic_t *p = atomic_read(x);
int r = atomic_read(p);
```

ctrl dependency

```
if (atomic_read(x)) {
        atomic_set(y, 1);
}
```

Case study (address dependency)

```
struct Foo {
    a: AtomicI32,
}

fn writer(ptr: &AtomicPtr<Foo>, foo: Foo, i: i32) {
    foo.a.store::<Relaxed>(i);
    ptr.store::<Release>(foo.leak() as _);
}
```

Case study (address dependency)

```
fn reader(ptr: &AtomicPtr<Foo>) -> Option<i32> {
    let p = ptr.load::<Relaxed>();
    // p = rcu_dereference(ptr);

    if !p.is_null() {
        // SAFETY: Address dependency guarantees the
        // current thread must observe all accesses to
        // `p` before the corresponding release store.
        Some(unsafe { (*p).a.load::<Relaxed>() })
        // READ_ONCE(p->a)
    } else {
        None
    }
}
```

(~2000 usage of `rcu_dereference()` alone)

Solutions?

- Assume Rust is equavilent as Clang in LLVM IR level, so we get the same deal.
  - Or a special `-kernel` flag of rustc.

Case study (how ctrl dependencies can be broken)

```
q = READ_ONCE(a);

if (q) {
    barrier();
    WRITE_ONCE(b, 1);
    do_something();
} else {
    barrier();
    WRITE_ONCE(b, 1);
    do_something_else();
}
```

Case study (how ctrl dependencies can be broken)

```
q = READ_ONCE(a);
barrier();
WRITE_ONCE(b, 1);

if (q) {
    do_something();
} else {
    do_something_else();
}
```

Solutions?

- volatile_if() or ctrl_dep()-like macro

```
if (ctrl_dep(cond)) { // => generate this into a `asm`.

}
```

Case study (how address dependencies can be broken)

(courtesy of Paul)

```
p = READ_ONCE(gp);
asm volatile("": : : "memory");

if (p == &static_p)
    r = READ_ONCE(p->a);
```

into

```
p = READ_ONCE(gp);
asm volatile("": : : "memory");

if (p == &static_p)
    r = READ_ONCE(static_p->a);
```