# klint:
# Compile-time Detection of Atomic Context Violations for Kernel Rust Code

Gary Guo

Rust MC, Linux Plumbers Conference, 15 Nov 2023

# Safety of Rust

- Soundness Property:
  - Safe Rust can't cause Undefined Behaviour

# Safety of Rust

- Soundness Property:
  - Safe Rust can't cause Undefined Behaviour
- Undefined Behaviour includes:
  - Dereference dangling, null or unaligned pointer (e.g. use-after-free)
  - Buffer overrun
  - Data races
  - Break alias rule
  - …

# Safety of Rust

- Soundness Property:
  - Safe Rust can't cause Undefined Behaviour
- Undefined Behaviour includes:
  - Dereference dangling, null or unaligned pointer (e.g. use-after-free)
  - Buffer overrun
  - Data races
  - Break alias rule
  - …
- What's considered safe:
  - Memory leak
  - Deadlock
  - Panic (kernel BUG) or abort (kernel panic)

# Some Bad Code

- Some obviously bad code:

```
spin_lock(&lock);
...
mutex_lock(&mutex);
...
spin_unlock(&lock);
```

- This can happen by accident.
- There’s no compile-time guarantee that this won’t happen.

# Some Bad Code

- Some obviously bad code:

```
spin_lock(&lock);
...
mutex_lock(&mutex);
...
spin_unlock(&lock);
```

- This can happen by accident.
- There’s no compile-time guarantee that this won’t happen.
- Is this “safe”?

# Some Bad Code

- Some obviously bad code:

```
spin_lock(&lock);
...
mutex_lock(&mutex);
...
spin_unlock(&lock);
```

- This can happen by accident.
- There’s no compile-time guarantee that this won’t happen.
- Is this “safe”?
- Yes, deadlock can happen, but it’s “safe”!

# Some Bad Code

- How about this:

```
rcu_read_lock();
...
schedule();
...
rcu_read_unlock();
```

- Is this “safe”?

# Read-copy-update (RCU) in Kernel

```
/* CPU 0 */                  /* CPU 1 */
rcu_read_lock();
ptr = rcu_dereference(v);    old_ptr = rcu_access_pointer(v);
/* use ptr */                rcu_assign_pointer(v, new_ptr);
                             synchronize_rcu();
                             /* waiting for RCU read to finish */
rcu_read_unlock();
                             /* synchronize_rcu() returns */
                             /* destruct and free old_ptr */
```

- This is a very simplified RCU use case
- What does this compile to?

# Read-copy-update (RCU) in Kernel

```
/* CPU 0 */                          /* CPU 1 */
→ preempt_disable();
ptr = rcu_dereference(v);            old_ptr = rcu_access_pointer(v);
/* use ptr */                        rcu_assign_pointer(v, new_ptr);
                                     synchronize_rcu();
                                     /* waiting for RCU read to finish */
→ preempt_enable();
                                     /* synchronize_rcu() returns */
                                     /* destruct and free old_ptr */
```

- If CONFIG_PREEMPT_RCU is off

# Read-copy-update (RCU) in Kernel

```
    /* CPU 0 */                        /* CPU 1 */
──→ barrier();
    ptr = rcu_dereference(v);          old_ptr = rcu_access_pointer(v);
    /* use ptr */                      rcu_assign_pointer(v, new_ptr);
                                       synchronize_rcu();
                                       /* waiting for RCU read to finish */
──→ barrier();
                                       /* synchronize_rcu() returns */
                                       /* destruct and free old_ptr */
```

- If CONFIG_PREEMPT_RCU is off
- If CONFIG_PREEMPT_COUNT is off

# Read-copy-update (RCU) in Kernel

```
/* CPU 0 */                     /* CPU 1 */
→ barrier();
ptr = rcu_dereference(v);       old_ptr = rcu_access_pointer(v);
/* use ptr */                   rcu_assign_pointer(v, new_ptr);
                                synchronize_rcu();
                                /* waiting for RCU read to finish */
→ barrier();
                                /* synchronize_rcu() returns */
                                /* destruct and free old_ptr */
```

- If CONFIG_PREEMPT_RCU is off
- If CONFIG_PREEMPT_COUNT is off
- No code is being generated for RCU read/unlock

# Read-copy-update (RCU) in Kernel

```
/* CPU 0 */                     /* CPU 1 */
barrier();
ptr = rcu_dereference(v);       old_ptr = rcu_access_pointer(v);
/* use ptr */                   rcu_assign_pointer(v, new_ptr);
                                synchronize_rcu();
                                /* waiting for RCU read to finish */
barrier();
                                /* synchronize_rcu() returns */
                                /* destruct and free old_ptr */
```

- `synchronize_rcu` returns after context switch has happened on all CPUs

# Read-copy-update (RCU) in Kernel

```
    /* CPU 0 */                    /* CPU 1 */
--> barrier();
    ptr = rcu_dereference(v);      old_ptr = rcu_access_pointer(v);
    /* use ptr */                  rcu_assign_pointer(v, new_ptr);
                                   synchronize_rcu();
                                   /* waiting for RCU read to finish */
--> barrier();
                                   /* synchronize_rcu() returns */
                                   /* destruct and free old_ptr */
```

- `synchronize_rcu` returns after context switch has happened on all CPUs
- This works by *assuming* that context switch will not happen in RCU read-side critical section.

# Revisit the Bad Code

```
/* CPU 0 */
rcu_read_lock();
ptr = rcu_dereference(v);


schedule();

/* use ptr after free! */
rcu_read_unlock();
```

```
/* CPU 1 */

old_ptr = rcu_access_pointer(v);
rcu_assign_pointer(v, new_ptr);
synchronize_rcu();
/* synchronize_rcu returns */
/* destruct and free old_ptr */
```

- Sleep inside RCU read-side critical section breaks assumption of `synchronize_rcu`

# Revisit the Bad Code

```
/* CPU 0 */                      /* CPU 1 */
rcu_read_lock();
ptr = rcu_dereference(v);        old_ptr = rcu_access_pointer(v);
                                 rcu_assign_pointer(v, new_ptr);
                                 synchronize_rcu();
schedule();                      /* synchronize_rcu returns */
                                 /* destruct and free old_ptr */
/* use ptr after free! */
rcu_read_unlock();
```

- Sleep inside RCU read-side critical section breaks assumption of `synchronize_rcu`
- This causes use-after-free, an undefined behaviour

# Implication

- Atomic context violation does not only cause deadlock, but can cause memory safety issue.

# Implication

- Atomic context violation does not only cause deadlock, but can cause memory safety issue.
- So not sleep inside atomic context is not only a *correctness requirement*, it’s also a *safety requirement*.

# Implication

- Atomic context violation does not only cause deadlock, but can cause memory safety issue.
- So not sleep inside atomic context is not only a *correctness requirement*, it’s also a *safety requirement*.
- This is fine for C, since there’s no distinction between safe and unsafe.

# Implication

- Atomic context violation does not only cause deadlock, but can cause memory safety issue.
- So not sleep inside atomic context is not only a *correctness requirement*, it's also a *safety requirement*.
- This is fine for C, since there's no distinction between safe and unsafe.
- But combining this with the soundness property of Rust, this means
  - We must not design a safe API that allows Rust kernel code to sleep inside atomic context.

# Possible solution: make sleep unsafe

- The issue disappears if we make sleep unsafe.
- Obviously a bad idea.

# Possible solution: token types

- A common pattern in Rust is to represent capabilities with token types:

```
trait Context {}
struct Atomic;
struct Process;
impl Context for Atomic {}
impl Context for Process {}
fn sleep(token: &mut Process);
impl Spinlock {
    fn lock(
        &self,
        context: &mut impl Context,
        callback: impl FnOnce(&mut Atomic, Guard<'_>)
    );
}
```

Assert the capability to sleep

Take away the current capability

Grant a restricted capability that does not permit sleeping

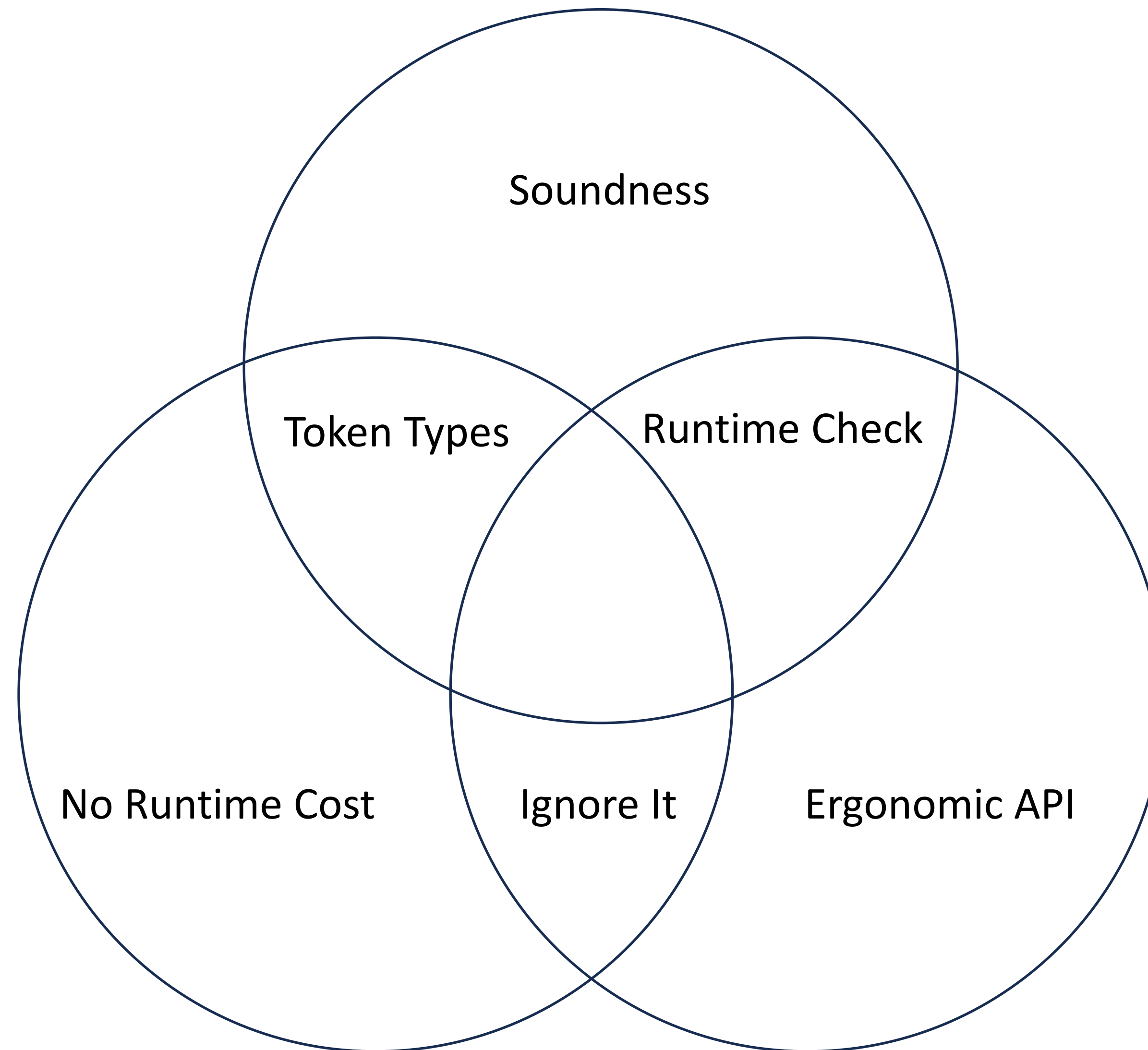
# Possible solution: dynamic check

- In preemptible kernel, we already have a runtime pre-emption count.
- Before any context switch, we can check that count.
- This is `CONFIG_DEBUG_ATOMIC_SLEEP`.
- This however has a runtime overhead.

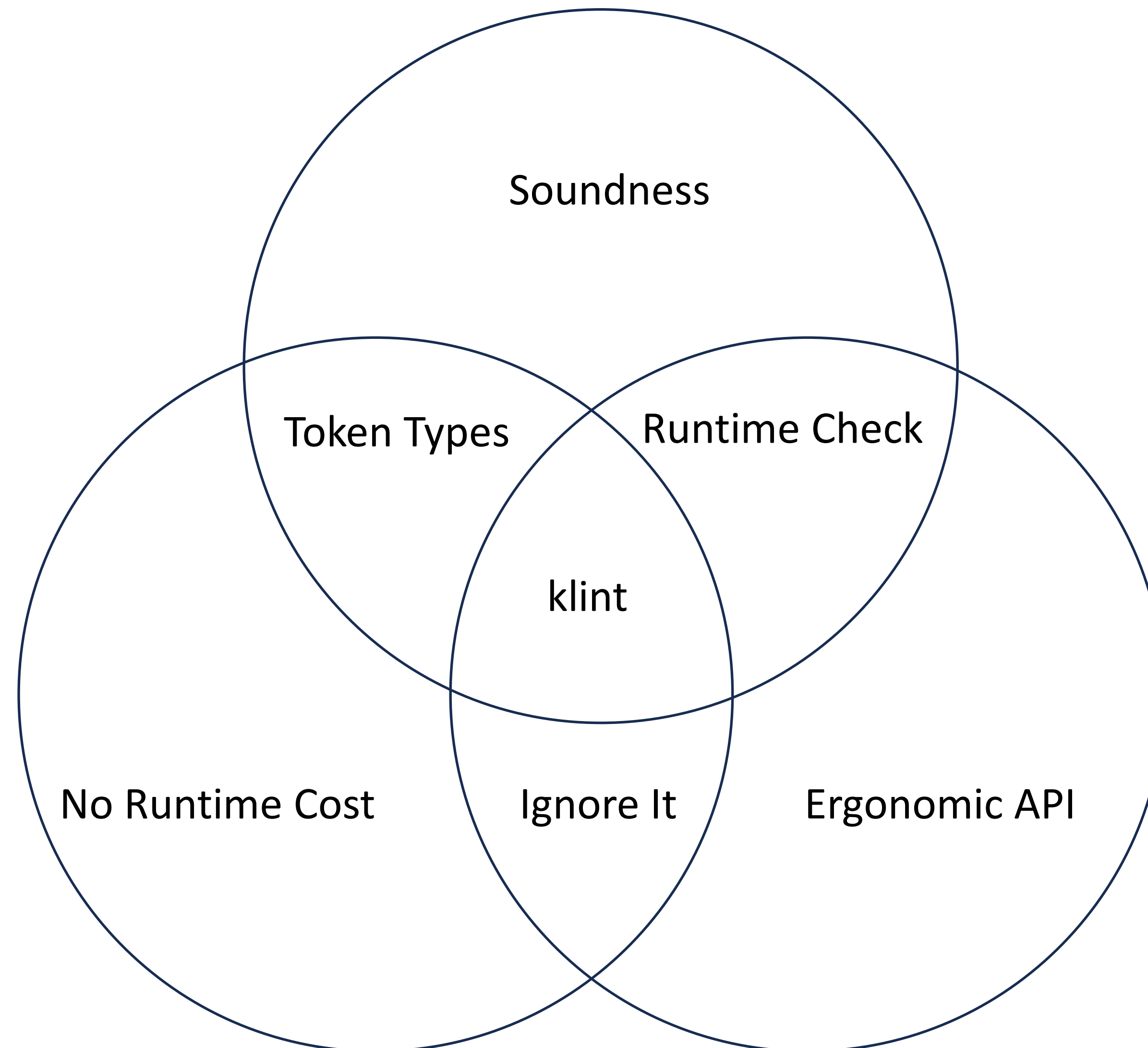
# Possible solution: just ignore it

- Use a normal API design.
- Just trust the developer.
- Use lockdep/`DEBUG_ATOMIC_SLEEP` to find bug during development.
- This is unsound.

# Choose two



# Choose ~~two~~ three



# Rationale

- Our need does not fit into Rust safety model
  - (Without complex typesystem dancing or runtime check)

# Rationale

- Our need does not fit into Rust safety model
  - (Without complex typesystem dancing or runtime check)
- But we can extend the compiler to provide what we need.
  - Servo also uses custom plugin to compiler to ensure proper GC rooting.

# Rationale

- Our need does not fit into Rust safety model
  - (Without complex typesystem dancing or runtime check)
- But we can extend the compiler to provide what we need.
  - Servo also uses custom plugin to compiler to ensure proper GC rooting.
- Check atomic context misuse during compilation time, as extensive as possible.
- For the ones that can’t be checked, provide escape hatch so that developer can override with runtime check or “unsafe”.

# klint: Design Goals

- Simple rule: easy to explain and understand
- Provide useful diagnostics
- Provide escape hatch to give developer full control when necessary
- A sane default that requires little annotation
- Fast: need to be feasible to run on *every* compilation

# klint: The Rule

- Each function is given two properties:
  - The **adjustment** to the preemption count after calling the function.
  - The **expected value** of preemption count allowed when calling the function.
- klint tracks possible preemption count at each location as if `preempt_count()` is enabled.
- As an approximation, adjustment must be an integer, and expected value must be a range.
- Examples:
  - `spin_lock` or `rcu_read_lock` adjusts by 1 and expect $[0, \infty)$
  - `spin_unlock` or `rcu_read_unlock` adjusts by -1 and expects $[1, \infty)$
  - Mutex operations adjusts by 0 and expects $[0, 1)$

# Annotation

Unchecked annotation

```
#[klint::preempt_count(adjust = 1, expect = 0.., unchecked)]
pub fn rcu_read_lock() -> RcuReadGuard { /* ... */ }

#[klint::drop_preempt_count(adjust = -1, expect = 1.., unchecked)]
struct RcuReadGuard { /* ... */ }

#[klint::preempt_count(adjust = 0, expect = 0, unchecked)]
pub fn schedule() { /* ... */ }

#[klint::preempt_count(expect = 0..)]
pub fn callable_from_atomic_context() { /* ... */ }
```

Checked annotation

# Inference

- Inference works in majority of cases, eliminating the need for annotation.

# Inference

- Inference works in majority of cases, eliminating the need for annotation.
- For generic functions, each monomorphised instance are inferred separately.

# Inference

- Inference works in majority of cases, eliminating the need for annotation.
- For generic functions, each monomorphised instance are inferred separately.
- Exceptions:
  - FFI boundaries
  - Recursion functions
  - Indirect function calls (function pointers, trait objects)
    - For trait object, trait methods can be annotated

# Case Study

- klint was tested on `rust` branch
- Inference works for most functions, annotation only required on the `ArcWake` trait.

# Case Study

```rust
pub trait ArcWake: Send + Sync {
    /// Wakes a task up.
    #[klint::preempt_count(expect = 0..)]
    fn wake_by_ref(self: ArcBorrow<'_, Self>);

    /// Wakes a task up and consumes a reference.
    #[klint::preempt_count(expect = 0..)]
    fn wake(self: Arc<Self>) {
        self.as_arc_borrow().wake_by_ref();
    }
}
```

- The wake functions are called from wake_up and therefore can't sleep.

# Case Study

```rust
pub trait ArcWake: Send + Sync {
    /// Wakes a task up.
    #[klint::preempt_count(expect = 0..)]
    fn wake_by_ref(self: ArcBorrow<'_, Self>);

    /// Wakes a task up and consumes a reference.
    #[klint::preempt_count(expect = 0..)]
    fn wake(self: Arc<Self>) {
        self.as_arc_borrow().wake_by_ref();
    }
}
```

- The wake functions are called from `wake_up` and therefore can’t sleep.
- And it turns out this exact annotation catches a bug.

```
error: trait method annotated to have preemption count expectation of 0..
   --> rust/kernel/kasync/executor/workqueue.rs:147:5
    |
147 |     fn wake(self: Arc<Self>) {
    |     ^^^^^^^^^^^^^^^^^^^^^^^^
    |
    = note: but the expectation of this implementing function is 0
note: the trait method is defined here
   --> rust/kernel/kasync/executor.rs:73:5
    |
73  |     fn wake(self: Arc<Self>) {
    |     ^^^^^^^^^^^^^^^^^^^^^^^^
note: which may drop type `kernel::sync::Arc<kernel::kasync::executor::workqueue::Task<core::future::from_generator::GenFuture<[static
generator@samples/rust/rust_echo_server.rs:25:75: 31:2]>>>` with preemption count 0..
  --> rust/kernel/kasync/executor/workqueue.rs:149:5
    |
147 |     fn wake(self: Arc<Self>) {
    |             ---- value being dropped is here
148 |         Self::wake_by_ref(self.as_arc_borrow());
149 |     }
    |     ^
<snip>
note: which may drop type `kernel::sync::arc::ArcInner<kernel::kasync::executor::workqueue::Executor>` with preemption count 0..
  --> rust/kernel/sync/arc.rs:255:22
    |
255 |             unsafe { core::ptr::drop_in_place(inner) };
    |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    |
    = note: which may drop type `kernel::kasync::executor::workqueue::Executor` with preemption count 0..
    = note: which may drop type `kernel::Either<kernel::workqueue::BoxedQueue, &kernel::workqueue::Queue>` with preemption count 0..
    = note: which may drop type `kernel::workqueue::BoxedQueue` with preemption count 0..
note: which may call this function with preemption count 0..
   --> rust/kernel/workqueue.rs:433:5
   |
433 |     fn drop(&mut self) {
    |     ^^^^^^^^^^^^^^^^^^
    |
    = note: but this function expects preemption count 0
```

# Current Limitation

```
impl<T> SpinLock<T> {
    // Preemption count adjustment of this function is 0 or 1
depending on the variant of the return value.
    fn try_lock(&self) -> Option<Guard<'_>> { ... }
}
```

- Conditional spinlock acquisition is currently not representable in klint.

# Current Limitation

```
impl<T> SpinLock<T> {
    // Preemption count adjustment of this function is 0 or 1
depending on the variant of the return value.
    fn try_lock(&self) -> Option<Guard<'_>> { ... }
}
```

- Conditional spinlock acquisition is currently not representable in klint.
- Possibly in the future?

```
impl<T> SpinLock<T> {
    #[klint::preempt_count(adjust =
        match return { Some(_) => 1, None => 0 }
    )]
    fn try_lock(&self) -> Option<Guard<'_>> { ... }
}
```

# Current Limitation

```
fn foo(take_lock: bool) {
    if take_lock {
        spin_lock(...);
    }
    ...
    if take_lock {
        spin_unlock(...);
    }
}
```

klint can’t yet tell that preemption count is always going to restored

- klint doesn’t currently reason about variable values.
- So data dependant lock acquisition also doesn’t work.

# Current Limitation

```
fn foo(take_lock: bool) {
    let guard;
    // An implicit bool will be introduced here by the compiler
to track if `guard` is initialised
    if take_lock {
        guard = SPINLOCK.lock();
    }
    ...
    // An implicit branch will be introduced here by the compiler
to drop `guard` only if it has been initialised
}
```

klint can’t yet tell that preemption count is always going to restored

- klint doesn’t currently reason about variable values.
- This includes drop flags, so this code also doesn’t compile under klint.

# Current Limitation

```rust
fn foo(x: Option<Guard>) -> Option<Guard> {
    if let Some(x) = x {
        return Some(x);
    }
    None
}
```

# Current Limitation

```
fn foo(x: Option<Guard>) -> Option<Guard> {
    if let Some(x) = x {
        return Some(x);
    }
    None
}

fn foo(x: Option<Guard>) -> Option<Guard> {
    if x.is_some() {
        return (x as Some).0;
    }
    drop(x); // <- rustc generates this since `x` needs drop
             // this drops `Option<Guard>`, so may drop `Guard`!
    None
}
```

# Current Limitation

```rust
fn foo(x: Option<Guard>) -> Option<Guard> {
    if let Some(x) = x {
        return Some(x);
    }
    None
}
```

This is currently blocking klint from wider testing

```rust
fn foo(x: Option<Guard>) -> Option<Guard> {
    if x.is_some() {
        return (x as Some).0;
    }
    drop(x); // <- rustc generates this since `x` needs drop
             // this drops `Option<Guard>`, so may drop `Guard`!
    None
}
```

# Questions

https://github.com/rust-for-linux/klint

# Links

- Repository:
  - https://github.com/rust-for-linux/klint
- For implementation details: refer to Kangrejos slides:
  - https://kangrejos.com/2023/Klint:%20Compile-time%20Detection%20of%20Atomic%20Context%20Violations%20for%20Kernel%20Rust%20Code.pdf
- Servo's GC rooting:
  - https://github.com/servo/servo/tree/master/components/script_plugins