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

Gary Guo

Kangrejos, 16 Sep 2023

The peril of sleep in atomic context

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

- We all know that this is bad code.

The peril of sleep in atomic context

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

- We all know that this is bad code.
- This can happen by accident.

The peril of sleep in atomic context

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

- We all know that this is bad code.
- This can happen by accident.
- Is this safe?

The peril of sleep in atomic context

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

- We all know that this is bad code.
- This can happen by accident.
- Is this safe?
- It can cause deadlock, but deadlock is memory-safe.

A simplified RCU use case 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 */
```

A simplified RCU use case 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.

A simplified RCU use case 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.

A simplified RCU use case 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 being generated for RCU read lock/unlock.

A simplified RCU use case in kernel

```
/* CPU 0 */
barrier(); // <-
ptr = rcu_dereference(v);
/* use ptr */



barrier(); // <-
```

```
/* CPU 1 */

old_ptr = rcu_access_pointer(v);
rcu_assign_pointer(v, new_ptr);
synchronize_rcu();
/* waiting for RCU read to finish */

/* synchronize_rcu() returns */
/* destruct and free old_ptr */
```

- If CONFIG_PREEMPT_RCU is off.
- If CONFIG_PREEMPT_COUNT is off.
- No code being generated for RCU read lock/unlock.
- synchronize_rcu returns when context switch happened on all CPUs.

A simplified RCU use case 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 being generated for RCU read lock/unlock.
- synchronize_rcu returns when context switch happened on all CPUs.
- This *assumes* that context switch will not happen in RCU read-side critical section.

A broken RCU use case

```
/* 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();
```

- The code exhibits undefined behaviour.

A broken RCU use case

```
/* 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();
```

- The code exhibits undefined behaviour.
- Sleep inside RCU read-side critical section breaks assumption of synchronize_rcu.

Take-away

- Correct function of `synchronize_rcu` relies on “no code sleeps inside atomic context”.

Take-away

- Correct function of `synchronize_rcu` relies on “no code sleeps inside atomic context”.
- Memory safety of `synchronize_rcu` users relies on correct function of `synchronize_rcu`.

Take-away

- Correct function of `synchronize_rcu` relies on "no code sleeps inside atomic context".
- Memory safety of `synchronize_rcu` users relies on correct function of `synchronize_rcu`.
- Therefore, not sleeping inside atomic context is a *safety requirement* of Rust kernel code, not just a *correctness requirement*.

Why it matters to Rust-for-Linux

- In C there is no notion of safe and unsafe code.
- But in Rust there is.
- We want to abstract kernel API Rust bindings in a *safe* and *sound* way.

Possible solution: unsafe RCU

- Make all RCU abstraction unsafe?

Possible solution: unsafe RCU

- Make all RCU abstraction unsafe?
- But this does not solve where Rust callback is called from C code in atomic context: sleeping in such case still causes C code to exhibit UB.

Possible solution: sleep is unsafe

- Make all sleepable function unsafe?

Possible solution: sleep is unsafe

- Make all sleepable function unsafe?
- Obvious bad idea.

Possible solution: 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<'_>));
}
```

- This is a simplified model without considering raw atomic context.

Possible solution: 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<'_>));
}
```

- This is a simplified model without considering raw atomic context.
- All functions that assume or change the context have to be written this way.

Possible solution: 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<'_>));
}
```

- This is a simplified model without considering raw atomic context.
- All functions that assume or change the context have to be written this way.
- You probably already feel the pain.

Possible solution: dynamic check

- Make `CONFIG_PREEMPT_COUNT` always enabled and check before sleep.

Possible solution: dynamic check

- Make `CONFIG_PREEMPT_COUNT` always enabled and check before sleep.
- My favourite solution.

Possible solution: dynamic check

- Make `CONFIG_PREEMPT_COUNT` always enabled and check before sleep.
- My favourite solution.
- Proposed by Wedson in the mailing list.

Possible solution: dynamic check

- Make CONFIG_PREEMPT_COUNT always enabled and check before sleep.
- My favourite solution.
- Proposed by Wedson in the mailing list.
- Linus don't like it.

Pragmatism over soundness

- Our need does not fit into Rust safety model

Pragmatism over soundness

- Our need does not fit into Rust safety model
- Runtime checks are not acceptable

Pragmatism over soundness

- Our need does not fit into Rust safety model
- Runtime checks are not acceptable
- Our API will not protect against misuse when `CONFIG_DEBUG_ATOMIC_SLEEP` is off.

Pragmatism over soundness

- Our need does not fit into Rust safety model
- Runtime checks are not acceptable
- Our API will not protect against misuse when `CONFIG_DEBUG_ATOMIC_SLEEP` is off.

- How about custom compile-time check?

Pragmatism over soundness

- Our need does not fit into Rust safety model
- Runtime checks are not acceptable
- Our API will not protect against misuse when `CONFIG_DEBUG_ATOMIC_SLEEP` is off.

- How about custom compile-time check?
- Entering Klint

Klint: design goals

- Simple rules: easy to understand by kernel developer.

Klint: design goals

- Simple rules: easy to understand by kernel developer.
- Must provide useful diagnostics.

Klint: design goals

- Simple rules: easy to understand by kernel developer.
- Must provide useful diagnostics.
- Tuneable: developer must be able to annotate to override when necessary.
- A sane default that requires little annotation.

Klint: design goals

- Simple rules: easy to understand by kernel developer.
- Must provide useful diagnostics.
- Tuneable: developer must be able to annotate to override when necessary.
- A sane default that requires little annotation.
- Fast: need to be feasible to run on *every* compilation.

The rule

- Klint tracks possible preemption count at each location as if `preempt_count()` is enabled.

The rule

- Klint tracks possible preemption count at each location as if `preempt_count()` is enabled.
- Each function is given two properties:
  - The **adjustment** to the preemption count after calling this function.
  - The **expected range** of preemption counts allowed when calling the function.

The rule

- Klint tracks possible preemption count at each location as if `preempt_count()` is enabled.
- Each function is given two properties:
  - The **adjustment** to the preemption count after calling this function.
  - The **expected range** of preemption counts allowed when calling the function.
- Examples:
  - `spin_lock` or `rcu_read_lock` adjusts by 1 and expects `0..`
  - `spin_unlock` or `rcu_read_unlock` adjusts by `-1` and expects `1..`
  - mutex operations adjusts by `0` and expects `0`

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() { /* ... */ }
```

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() { /* ... */ }
```

The check logic

Four step process:

1. Infer preemption count adjustments for each function
2. Infer preemption count expectations for each function
3. Check preemption count adjustments for each annotated function
4. Check preemption count expectations for each annotated function

Inference and check needs to be separate for recursive functions, more on this later.

Preemption count inference

- This is a dataflow analysis.
- Domain = sets of possible preemption counts, currently represented by a range
- For each call site or drop site, adjust possible preemption counts by the preemption count of the callee.
- For each basic block, union possible preemption counts of all previous blocks

Preemption count inference: convergence

Problem: the domain is not of finite height

```
loop {
    spin_lock();
    if rand() {
        break;
    }
}
```

Preemption count inference: convergence

Problem: the domain is not of finite height

```
loop {
    spin_lock();
    if rand() {
        break;
    }
}
```

Chain: $[0, 1) \to [0, 2) \to [0, 3) \to ...$ is infinite. The analysis will not converge.

Preemption count inference: convergence

Problem: the domain is not of finite height

```
loop {
    spin_lock();
    if rand() {
        break;
    }
}
```

Chain: $[0, 1) \to [0, 2) \to [0, 3) \to \ldots$ is infinite. The analysis will not converge.

Solution:

- Extending positive upper bound towards inf produces inf
- e.g. $[0, 1) \vee [0, 2) := [0, \inf)$
- Extending negative lower bound towards $-\inf$ produces $-\inf$
- Extending negative upper bound or positive lower bound still does the expected range join
- e.g. $[1, 3) \vee [0, 3) := [0, 3)$

The check logic

Four step process:

1. Infer preemption count adjustments for each function
2. Infer preemption count expectations for each function
3. Check preemption count adjustments for each annotated function
4. Check preemption count expectations for each annotated function

The check logic

Four step process:

1. Infer preemption count adjustments for each function
   - Infer the preemption count at `Return`, check if it's single-valued, and use it
2. Infer preemption count expectations for each function

3. Check preemption count adjustments for each annotated function

4. Check preemption count expectations for each annotated function

The check logic

Four step process:

1. Infer preemption count adjustments for each function
   - Infer the preemption count at `Return`, check if it's single-valued, and use it
2. Infer preemption count expectations for each function
   - $\bigwedge_{c \in \text{callsite}}$ (expectation of callee − adjustment at callsite)
3. Check preemption count adjustments for each annotated function

4. Check preemption count expectations for each annotated function

The check logic

Four step process:

1. Infer preemption count adjustments for each function
   - Infer the preemption count at `Return`, check if it's single-valued, and use it
2. Infer preemption count expectations for each function
   - $\bigwedge_{c \in \text{callsite}}$ (expectation of callee − adjustment at callsite)
3. Check preemption count adjustments for each annotated function
   - Infer the preemption count at `Return`, check if it matches annotation
4. Check preemption count expectations for each annotated function

The check logic

Four step process:

1. Infer preemption count adjustments for each function
   - Infer the preemption count at `Return`, check if it's single-valued, and use it
2. Infer preemption count expectations for each function
   - $\bigwedge_{c \in \text{callsite}}$ (expectation of callee − adjustment at callsite)
3. Check preemption count adjustments for each annotated function
   - Infer the preemption count at `Return`, check if it matches annotation
4. Check preemption count expectations for each annotated function
   - At each callsite, check
     (annotated expectation + adjustment at callsite) $\wedge$ expectation of callee $\neq \bot$

Complication: recursion

```
enum Node<T> {
    Leaf(T),
    Branch(Box<Node<T>>, Box<Node<T>>),
}

fn iter<T>(node: &Node<T>, f: &mut impl FnMut(&T)) {
    match node {
        Node::Leaf(v) => f(v),
        Node::Branch(l, r) => {
            iter(l, f);
            iter(r, f);
        }
    }
}
```

Complication: recursion

```
enum Node<T> {
    Leaf(T),
    Branch(Box<Node<T>>, Box<Node<T>>),
}

fn iter<T>(node: &Node<T>, f: &mut impl FnMut(&T)) {
    match node {
        Node::Leaf(v) => f(v),
        Node::Branch(l, r) => {
            iter(l, f);
            iter(r, f);
        }
    }
}
```

Solution: use default value (no adjustment, no expectation) when query cycle occurs, and check that the assumption holds.

Complication: recursion

Solution: use default value (no adjustment, no expectation) when query cycle occurs, and check that the assumption holds. If the default is not correct, annotate:

```
#[klint::drop_preempt_count(expect = 0)]
enum Node<T> { /* ... */ }

#[klint::preempt_count(expect = 0)]
fn iter<T>(node: &Node<T>, f: &mut impl FnMut(&T)) { /* ... */ }
```

Complication: generics

Problem: property of generic functions depend on generic arguments.

Complication: generics

Problem: property of generic functions depend on generic arguments.

- Whether `Option::map` sleep depends on the function that we give it.
- Whether `Lock::lock` sleep depends on if the backend is Mutex.

Complication: generics

Problem: property of generic functions depend on generic arguments.

- Whether `Option::map` sleep depends on the function that we give it.
- Whether `Lock::lock` sleep depends on if the backend is Mutex.

Solution: we check monomorphised instances.

Complication: generics

Problem: property of generic functions depend on generic arguments.

- Whether `Option::map` sleep depends on the function that we give it.
- Whether `Lock::lock` sleep depends on if the backend is Mutex.

Solution: we check monomorphised instances.

- This is optimised so that `klint` will try to check polymorphically first, and will fall back to monomorphised check if the function is too generic.
- Full detail: all `klint` analyses accept `ParamEnv` and can will instantiate when necessary.

Complication: generics

Problem: property of generic functions depend on generic arguments.

- Whether `Option::map` sleep depends on the function that we give it.
- Whether `Lock::lock` sleep depends on if the backend is Mutex.

Solution: we check monomorphised instances.

- This is optimised so that `klint` will try to check polymorphically first, and will fall back to monomorphised check if the function is too generic.
- Full detail: all `klint` analyses accept `ParamEnv` and can will instantiate when necessary.

- We store inferred/annotated results of each monomorphised function from a crate in `crate_name.klint`, so a downstream crate don't need to check upstream crate.

Complication: indirect function call

Problem: indirect function call is a boundary for inference.

Complication: indirect function call

Problem: indirect function call is a boundary for inference.

- Inference stops working on function pointer or trait object method calls.

Complication: indirect function call

Problem: indirect function call is a boundary for inference.

- Inference stops working on function pointer or trait object method calls.

Solution: assume on use-site, and check on def-site

Complication: indirect function call

Problem: indirect function call is a boundary for inference.

- Inference stops working on function pointer or trait object method calls.

Solution: assume on use-site, and check on def-site

- Function pointers are assumed to be sleepable and make no adjustment.

Complication: indirect function call

Problem: indirect function call is a boundary for inference.

- Inference stops working on function pointer or trait object method calls.

Solution: assume on use-site, and check on def-site

- Function pointers are assumed to be sleepable and make no adjustment.
- `klint` will warn when a Rust function with different property is casted to a function pointer

Complication: indirect function call

The same applies to traits, except that trait methods can be annotated.

```
/// A waker that is wrapped in [`Arc`] for its reference counting.
///
/// Types that implement this trait can get a [`Waker`] by calling
↪  [`ref_waker`].
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..)] // Functions callable from
    ↪  `wake_up` must not sleep
    fn wake(self: Arc<Self>) {
        self.as_arc_borrow().wake_by_ref();
    }
}
```

Limitation

No way to represent conditional lock acquisition, e.g. `try_lock`.

```
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<'_>> { ... }
}
```

A callback-based API would solve this issue, but...

Limitation

klint don't reason about variable values yet:

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

Limitation

RAII doesn't help:

```
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
}
```

Limitation

It was discovered that RAII actually situation even worse

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

is desugared to something like this:

```
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
}
```

Future Work

- Address `try_lock`.

Future Work

- Address `try_lock`.
- Address condition on variable issue.
    - A more complex inference logic that can track variant and value of local variables is WIP.

Future Work

- Address `try_lock`.
- Address condition on variable issue.
  - A more complex inference logic that can track variant and value of local variables is WIP.
- Maybe instead of using numbers to represent preempt counts, a stack may be more appropriate?

Future Work

- Address `try_lock`.
- Address condition on variable issue.
  - A more complex inference logic that can track variant and value of local variables is WIP.
- Maybe instead of using numbers to represent preempt counts, a stack may be more appropriate?
- Improve diagnostics to be more developer friendly
- ...