Loading…

shouldn't std::sync::Mutex::get_mut() be unsafe

Ask Question

Asked 2 years, 1 month ago

Modified 2 years, 1 month ago

Viewed 899 times

6

Docs say

> Returns a mutable reference to the underlying data. Since this call borrows the Mutex mutably, no actual locking needs to take place – the mutable borrow statically guarantees no locks exist.

So the problem is **no locking needed**. Sure, if the mutex (in)directly is not used from any unsafe block then it is not possible any thread holds a reference on it and therefore can't use it. That also means, in order to compile the call of get_mut, if any thread was using it the thread must already finish its execution. It is also fine that any new thread will access the data protected by the mutex, because it will be read from the memory as if acquire semantic, so any changes made with data via get_mut will be visible to any new thread.

However, if a mutex was used from an unsafe block, it is possible that some thread (likely ffi like in my case) still has a pointer on the mutex and can either hold the lock and use data or just finish mutating the data and unlock the mutex. In the former case, it is clearly UB, in the latter case, it is a race condition, which is also UB, because the thread, which calls get_mut, might not see the latest state of the protected data, which was just updated by another thread, which accesses the mutex through the pointer. In order to make the latest changes visible on the thread calling get_mut acquire/release semantic should be held, but get_mut doesn't call any acquire operation.

get_mut is a useful function as it is but I don't believe it is thread-safe and therefore should be unsafe.

- rust

Share

Edit

Follow

asked Oct 13, 2023 at 3:33

Anton Dyachenko

43333 silver badges1111 bronze badges

7

> There's a relatively simple reasoning for `get_mut` being safe: You cannot use it to produce races/UB without unsafe code. That's it. Hence, there must be a hole in your reasoning somewhere.
>
> Caesar – Caesar
>
> 2023-10-13 04:27:42 +00:00
>
> Commented Oct 13, 2023 at 4:27
>
> 4
>
> Recall the type system! `get_mut` requires a mutable reference to the `Mutex`. If you hold a mutable reference to the `Mutex`, then nobody else holds any references to it, period. And since there are no other references to it, there can be no race condition. Hence, no locking is needed.
>
> alter_igel – alter_igel
>
> 2023-10-13 04:32:34 +00:00
>
> Commented Oct 13, 2023 at 4:32
>
> Note that this is not a separated example, but rather a pattern that occurs often anywhere where you have an interior mutability. For example std::cell::Cell has method Cell::get_mut which allows you to get a mutable reference to item inside cell, as long as you have an *exclusive* reference to the cell itself.
>
> Aleksander Krauze – Aleksander Krauze
>
> 2023-10-13 06:47:30 +00:00

Commented Oct 13, 2023 at 6:47

2

Just to be clear, do you mean something like this playground? You think there is an UB because there is no *aquire* between the last *release* and the access of the `get_mut()`, isn't it?

rodrigo – rodrigo

2023-10-13 10:51:43 +00:00

Commented Oct 13, 2023 at 10:51

2

@AntonDyachenko Can you write a MRE that demonstrates the UB?

BallpointBen – BallpointBen

2023-10-14 00:15:57 +00:00

Commented Oct 14, 2023 at 0:15

Add a comment | Show **2** more comments

3 Answers

Sorted by: Reset to default

Highest score (default) Trending (recent votes count more) Date modified (newest first) Date created (oldest first)

8

The main property of mutable references in Rust is that they are unique. That is, when the code as a whole is sound, it is guaranteed that the *only* way to access the value referenced by the mutable reference is through that exact reference. In safe code, this is enforced automatically. In the unsafe code, including FFI, that's the invariant *you*, as the one writing `unsafe`, are expected to uphold.

In particular, when you have the mutable reference to the `Mutex`, it's explicit and immediate UB to have any other place access the same mutex or the data behind it. In particular, it's essentially UB to have it locked by anyone else, since the unlock would, by definition, access the mutex (to store the "not locked" bit).

So, if some `unsafe` code leads to `get_mut` being non-thread-safe, then this unsafe code is itself unsound.

Share

Edit

Follow

answered Oct 13, 2023 at 4:25

Cerberus

10.7k11 gold badge3535 silver badges5353 bronze badges

You can now comment to request clarification or add additional context.

7 Comments

Add a comment

Caesar

Caesar Over a year ago

Isn't it already UB once you create the two mutable references? I.e., no need to actually access anything.

2023-10-13T04:27:30.93Z+00:00

2

Reply

- Copy link

Cerberus

Cerberus Over a year ago

I think OP is referring to the case when we have not two mutable references, but e.g. `&mut T` and `*mut T`. The *existence* of these at the same time is not UB, but actually *using* the raw pointer while the reference is active definitely is.

2023-10-13T05:51:09.023Z+00:00

2

Reply

- Copy link

Caesar

Caesar Over a year ago

What can you use the raw pointer for without creating a reference?

2023-10-13T06:46:07.623Z+00:00

0

Reply

- Copy link

Cerberus

Cerberus Over a year ago

`core::ptr::write`, for example.

2023-10-13T08:10:52.99Z+00:00

0

Reply

- Copy link

Anton Dyachenko

Anton Dyachenko Over a year ago

as usual, people fighting with the weak argument - simultaneous access / not unique reference first case, but completely ignoring the strong argument - my second case. In my second example, there are NO references when a mut ref is used, BUT because another thread has just finished mutation through ref under the lock, this thread that getting data via get_mut and the ONLY mut ref will not see the latest version of the data, because it doesn't lock the mutex. So, in my second example, nothing is unsound but race condition is present -> UB -> unsafe.

2023-10-13T08:39:50.543Z+00:00

0

Reply

- Copy link

Add a comment | Show 2 more comments

4

If you are only using safe code, then the mere existence of the mutable reference implies that you have a happens-before relationship, thus synchronization, at compile time.

If you are using unsafe code to access the lock, then *you* are responsible to make sure there is a happens-before relationship. If you don't do that, your code is unsound even without actually changing the data, because you're creating two overlapping mutable references without synchronization, therefore UB. So *you* are responsible for creating proper synchronization. If you do it, everything is fine. If you don't, *your* unsafe code is unsound and the UB comes from *your* unsafe code. std's `get_mut()` plays no role here, and therefore is sound.

Share

Edit

Follow

answered Oct 13, 2023 at 9:57

Chayim Friedman

76.2k55 gold badges9797 silver badges140140 bronze badges

Comments

Add a comment

-3

The answer to my question is, yes, sure it should be unsafe. Run this on ARM (x64 works fine because of strong memory). There are exactly 2 unsafe blocks and all the safety conditions for them are met therefore it is a sound (from the borrow checker point of view) code but it is not thread-safe. Replacing both get_mut onto the lock solves the problem as expected.

```
use std::sync::atomic::AtomicPtr;
use std::sync::atomic::Ordering::*;
use std::sync::Mutex;

static RX: AtomicPtr<Mutex<i32>> = AtomicPtr::new(std::ptr::null_mut());
static TX: AtomicPtr<Mutex<i32>> = AtomicPtr::new(std::ptr::null_mut());
const COUNT: i32 = 64 * 1024 * 1024;

fn main() {
   let t = std::thread::spawn(|| {
       let mut current = std::ptr::null_mut();
       let mut m;
       for i in 0..COUNT {
           loop {
               match RX.compare_exchange(current, std::ptr::null_mut(), Relaxed, Relaxed) {
                   Ok(ptr) if !ptr.is_null() => {
                       m = unsafe { Box::from_raw(ptr) };
                       break;
                   }
                   Ok(ptr) | Err(ptr) => current = ptr,
               }
```

```
            }
            assert_eq!(m.get_mut().unwrap(), &-i);
            *m.get_mut().unwrap() = i;
            TX.store(Box::<_>::into_raw(m), Relaxed);
        }
    });

    let mut m = Box::new(Mutex::new(0));
    for i in 0..COUNT {
        *m.get_mut().unwrap() = -i;
        RX.store(Box::<_>::into_raw(m), Relaxed);
        let mut current = std::ptr::null_mut();
        loop {
            match TX.compare_exchange(current, std::ptr::null_mut(), Relaxed, Relaxed) {
                Ok(ptr) if !ptr.is_null() => {
                    m = unsafe { Box::from_raw(ptr) };
                    break;
                }
                Ok(ptr) | Err(ptr) => current = ptr,
            }
        }
        assert_eq!(m.get_mut().unwrap(), &i);
    }
    t.join().unwrap();
}
```

PS: I decided to put an extra explanation here as comments do not really work for this. Let's look at `Arc::drop` and their comments

```
    fn drop(&mut self) {
        // Because `fetch_sub` is already atomic, we do not need to synchronize
        // with other threads unless we are going to delete the object. This
        // same logic applies to the below `fetch_sub` to the `weak` count.
        if self.inner().strong.fetch_sub(1, Release) != 1 {
            return;
        }

        // This fence is needed to prevent reordering of use of the data and
        // deletion of the data. Because it is marked `Release`, the decreasing
        // of the reference count synchronizes with this `Acquire` fence. This
        // means that use of the data happens before decreasing the reference
        // count, which happens before this fence, which happens before the
        // deletion of the data.
        //
        // As explained in the [Boost documentation][1],
        //
        // > It is important to enforce any possible access to the object in one
        // > thread (through an existing reference) to *happen before* deleting
        // > the object in a different thread. This is achieved by a "release"
        // > operation after dropping a reference (any access to the object
        // > through this reference must obviously happened before), and an
        // > "acquire" operation before deleting the object.
        //
        // In particular, while the contents of an Arc are usually immutable, it's
        // possible to have interior writes to something like a Mutex<T>. Since a
        // Mutex is not acquired when it is deleted, we can't rely on its
        // synchronization logic to make writes in thread A visible to a destructor
        // running in thread B.
        //
        // Also note that the Acquire fence here could probably be replaced with an
        // Acquire load, which could improve performance in highly-contended
        // situations. See [2].
        //
        // [1]: (www.boost.org/doc/libs/1_55_0/doc/html/atomic/usage_examples.html)
        // [2]: (https://github.com/rust-lang/rust/pull/41714)
        acquire!(self.inner().strong);

        unsafe {
            self.drop_slow();
        }
```

```
}
```

So now, I have an even stronger opinion about the whole unsoundness of `std::sync::Mutex`. Only the lock family of methods of mutex are ok, `get_mut` and `drop` are not thread-safe. Remember the whole point of Mutex abstraction is to provide thread-safe access via public safe API -> ensuring any public safe method has a happens-before relationship among each other in ANY safe scenario.

Back to `Arc::drop` it is already having a scratch to workaround bug in the `Mutex::drop`. If `Mutex::drop` would ensure the happens-before relationship with all other `Mutex` public safe API functions then `Arc` could use simple relaxed ordering for every type.

1. Let's imagine that `T` in `Arc<T>` has no interior mutability then by rust language guarantee it is always safe to use relaxed order on the counter because the only access to the protected data is load.
2. Let's imagine that `T` in `Arc<T>` exposes interior mutability then either `Arc` should ensure the happens-before relationship (which c++ `std::shared_ptr` does and mimicked by rust) or (which is much more logical) the type T with interior mutability has to ensure happens-before relationship in its public safe API.

Now, because of the wrong responsibility distribution between `Arc` and `Mutex` design choice, all other code similar to `Arc` has to mimic that bad design and ensure the happens-before relationship.

Another argument, what is the ratio of `Arc::drop` calls over `Mutex::drop` calls in an ordinary program? I bet much less than 1. Therefore it is even more optimal to ensure the happens-before relationship in `Mutex` rather than in everywhere else, at the end of the day this is what `Mutex` abstraction should do in a safe language like rust.

Again having &mut ref doesn't assume any happens-before relationship on the PROTECTED data it only guarantees the absence of any other refs, and in my example, this is the case. Remember the internal structure of the `Mutex` it contains `UnsafeCell` and `sys::Mutex`, in my example with relaxed order ANY operation on the `sys::Mutex` is thread-safe, and it is guaranteed by `sys::Mutex` implementation, but access to `UnsafeCell` isn't and requires happens-before relationship, that what `Mutex` for isn't it?

In other words, if you deconstruct `Mutex` abstraction into a pair of `sys::Mutex` and `UnsafeCell` and pass a pointer over this boxed pair then after converting the pointer back to reference you can safely use `sys::Mutex` part in all scenarios but you can't do this with `UnsafeCell` without ensuring happens-before relationship.

PPS: Basically, all the arguments, from people who downvoted this answer, could be narrowed down to one simple sentence:

In rust, it is illegal to use `std::sync::AtomicPtr` with `std::sync::atomic::Ordering::Relaxed` in any safe code.

I can leave with and accept this point of view when this will be encoded in the rust type system.

Share

Edit

Follow

edited Oct 17, 2023 at 21:35

answered Oct 14, 2023 at 9:22

Anton Dyachenko

43333 silver badges1111 bronze badges

25 Comments

Add a comment

Chayim Friedman

Chayim Friedman Over a year ago

No, your code is unsound and is its execution is outright UB. As I said in my answer, you have two `Box`es pointing to the same memory at the same time. **This is UB.** The fact that it might not be at the same time physically doesn't matter, since on the abstract machine it *is* one the same time (as there is no synchronization), and this is the only thing that matters.

2023-10-14T16:53:15.003Z+00:00

2

Reply

- Copy link

Chayim Friedman

Chayim Friedman Over a year ago

@alter_igel This is not entirely correct; an unsafe code might be considered unsound even if it can only be used to cause UB by unsafe code, *iff* that unsafe code is sound. Of course, the definition of soundness is conventional. There are some things that are obviously unsound, but some are not and are arguably are sound or not. In this case, like I said, this code is unsound according to the rules, but even without this, we generally consider std code to be sound and therefore we assume that code that unsafe triggers UB with std code is unsound.

2023-10-14T21:39:35.47Z+00:00

1

Reply

- Copy link

Chayim Friedman

Chayim Friedman Over a year ago

No. I'm saying that multithreaded code that shares anything **without proper synchronization** (and of course, mutates it too, or at least creates a mutable reference) is UB.

2023-10-16T03:06:35.703Z+00:00

1

Reply

- Copy link

Chayim Friedman

Chayim Friedman Over a year ago

But again, **physical time does not matter**. We are not running in the physical world - we are running in the Abstract Machine world. And in that world, things from two threads are interleaving unless there is proper synchronization (a happens-before relationship) within them.

2023-10-16T03:18:05.947Z+00:00

1

Reply

- Copy link

Cerberus

Cerberus Over a year ago

"Let's imagine that T in Arc<T> has no interior mutability then by rust language guarantee it is always safe to use relaxed order on the counter because the only access to the protected data is load" - and load must happen before deallocation (we don't want use-after-free, right?). How could you guarantee that without requiring that load happens before this atomic fence?

2023-10-18T03:41:41.077Z+00:00

1

Reply

- Copy link

Add a comment | Show 20 more comments

Your Answer

Thanks for contributing an answer to Stack Overflow!

- Please be sure to *answer the question*. Provide details and share your research!

But *avoid* …

- Asking for help, clarification, or responding to other answers.
- Making statements based on opinion; back them up with references or personal experience.

To learn more, see our tips on writing great answers.

Some of your past answers have not been well-received, and you're in danger of being blocked from answering.

Please pay close attention to the following guidance:

- Please be sure to *answer the question*. Provide details and share your research!

But *avoid* …

- Asking for help, clarification, or responding to other answers.
- Making statements based on opinion; back them up with references or personal experience.

To learn more, see our tips on writing great answers.

Post Your Answer Discard

Start asking to get answers

Find the answer to your question by asking.

Ask question

Explore related questions

- rust

See similar questions with these tags.

- The Overflow Blog
  Introducing Stack Overflow AI Assist—a tool for the modern developer
  Treating your agents like microservices
- Featured on Meta
  Chat room owners can now establish room guidelines
  AI Assist is now available on Stack Overflow
  Policy: Generative AI (e.g., ChatGPT) is banned

Cast your votes this week!

0/5

Vote on content that is interesting, well-researched and helpful! You have 5 votes a week without reputation. Learn more about free votes

Love this site?

Get the **weekly newsletter!** In it, you'll get:

- The week's top questions and answers
- Important community announcements
- Questions that need answers

Sign up for the digest

see an example newsletter

Related

50
What are move semantics in Rust?
1
Mutably borrowing fields of struct results in an apparent contradiction in compiler error message
0
How to implement Kahn's algorithm for a topological sort in rust?
0
How can I use `unsafe` to create multiple references?
2
Does operating system thread scheduling factor in to the decision to use std::sync::Mutex vs tokio::sync::Mutex?
1
Is it possible to borrow different parts of self with different mutability statically?
0
Passing self into a function of a struct instance of a field of a struct in Rust
85
Understanding the Send trait
1
Lockless unsafe data sharing bus in rust
3
Rust: Is it safe to cast a mutable borrow to a pointer and back (to placate the borrow checker), if I know that there will only be one instance of it?

Hot Network Questions

- \ifthenelse and modulo
- Looking for a proper formal substitute for "quick fix" for formal letter
- How does supporting "One China" work without also supporting Taiwan-China reunification?
- Statistical testing for bimodal/multimodal sample
- Strength of AES when 32-bits of the 128-bit key are known?
- Zero Inflated Beta Regression (or not)?
- Does short range teleporting end floating disk?

- Who is Patrick and why is he referred to at anti-AfD demonstrations?
- Is there something missing with this simple geometry problem?
- How do I add a text editor to my (debian-based) initramfs?
- Step in Brezis' proof of chain rule for Sobolev spaces
- Why is "Papist" a derogatory term?
- What does Felix mean by telling Bond to pick a contact point while "standing up" in Diamonds Are Forever?
- How to draw cone with infinitely many extremal rays
- Passive Butterworth filter synthesis with Cauer topology
- How to protect 100-year-old wooden floors for the next 100 years?
- Chrome says "To get security updates you need at least macOS 10.15. Please upgrade your OS." on 10.13 (High Sierra). How can I disable that message?
- Can I Automatically List Supported Kernel Parameters in GRUB?
- Ping sweep of IP subnets
- Critiques of the real vs. nominal definition distinction
- Correct website for Thailand ETA application form
- Optimize Manipulate
- What do the terms postponement, slip and scrub mean in the context of the space shuttle program?
- "Creatures your opponents control" after the ability is activated

more hot questions

Question feed

Subscribe to RSS

Question feed

To subscribe to this RSS feed, copy and paste this URL into your RSS reader.

- Facebook
- Twitter
- LinkedIn
- Instagram