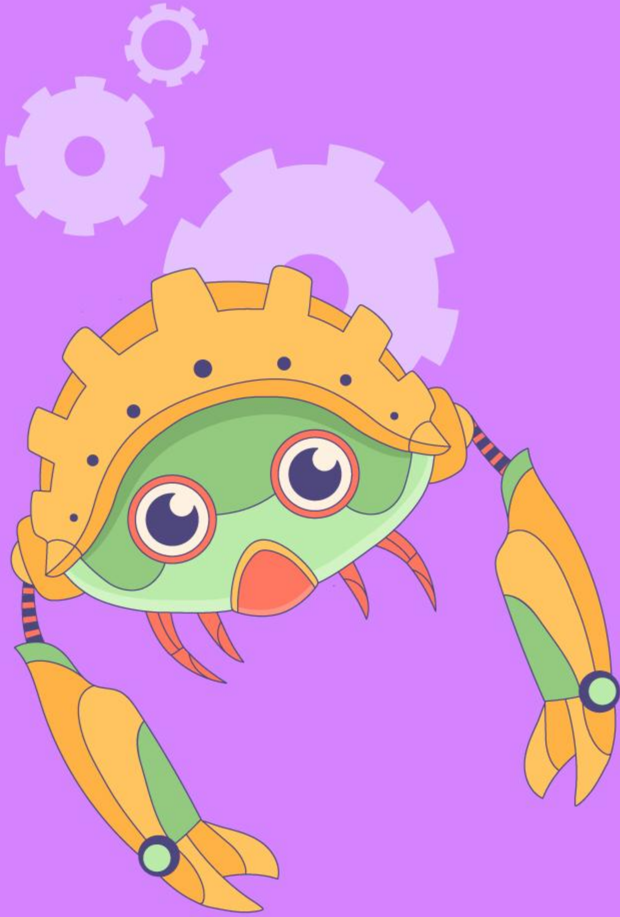
ANGELO RENDINA

Senior Software Engineer @ Prima Assicurazioni

The Guard Pattern



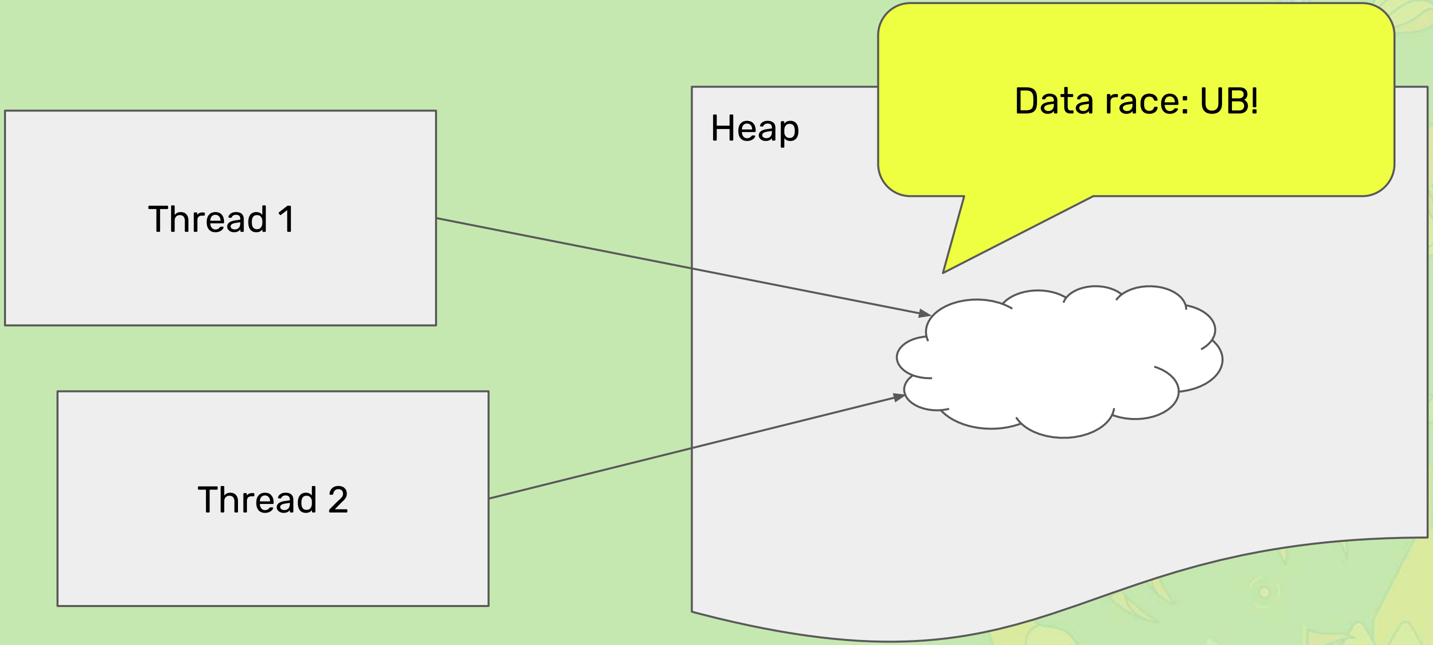
The Guard Pattern

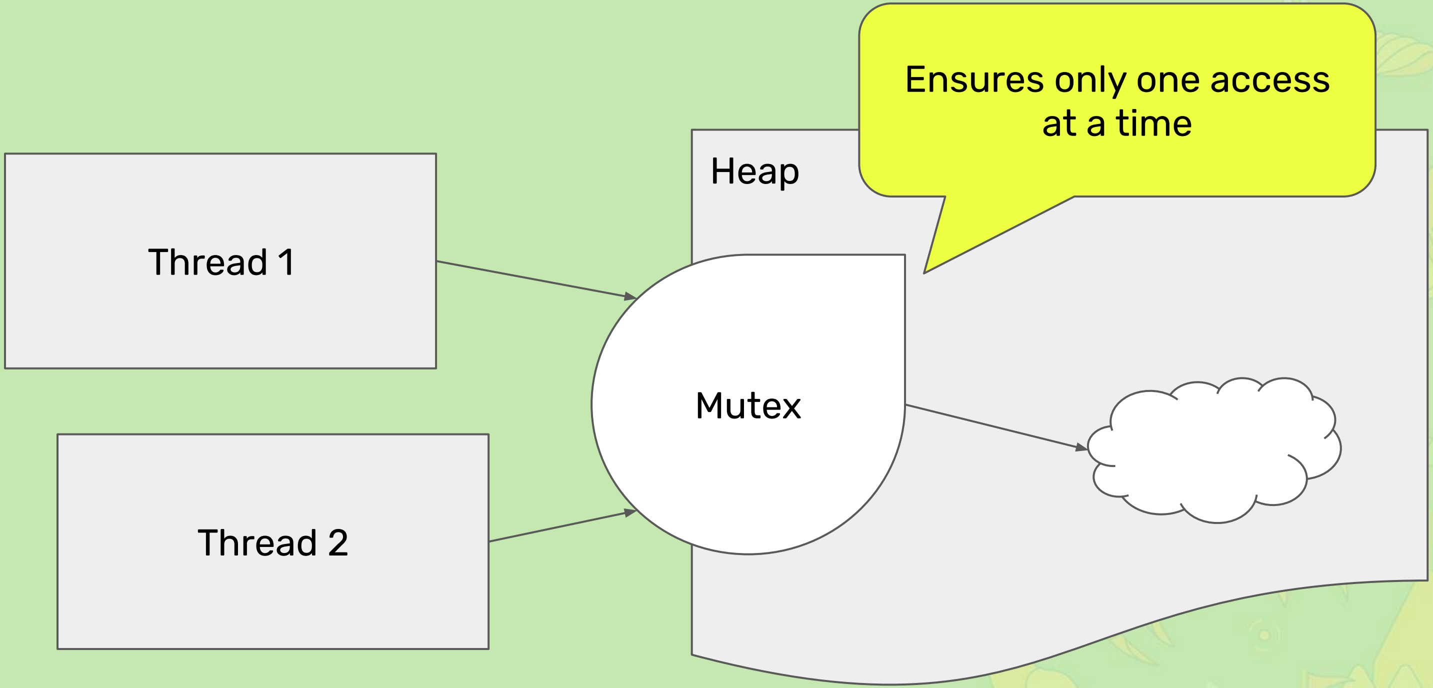
Index

1. Motivation: Mutex
2. RAII
3. Use case: PGMutex
4. Use case: Kernel ScopeGuard
5. Final remarks

Section 1

Mutex

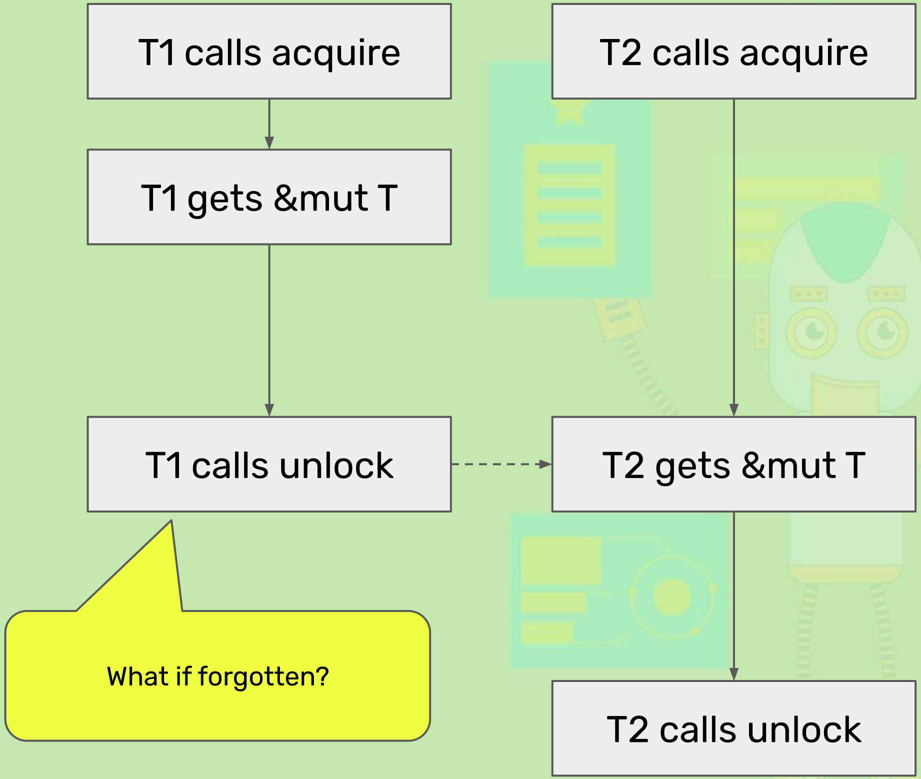

Thread 1
Thread 2
Heap
Data race: UB!


Ensures only one access at a time
Heap
Thread 1
Thread 2
Mutex

Does not support simultaneous callers

```
Mutex::acquire(&mut self) -> &mut T {
    while self.locked { … }

    self.locked = true;
    &mut self.inner
}

Mutex::unlock(&mut self) {
    self.locked = false;
}
```

Who calls this?



Needs interior mutabilty

```
Mutex::acquire(&self) -> LockGuard<T> {
    while self.locked { … }

    self.locked = true;
    LockGuard::new(&self)
}
```

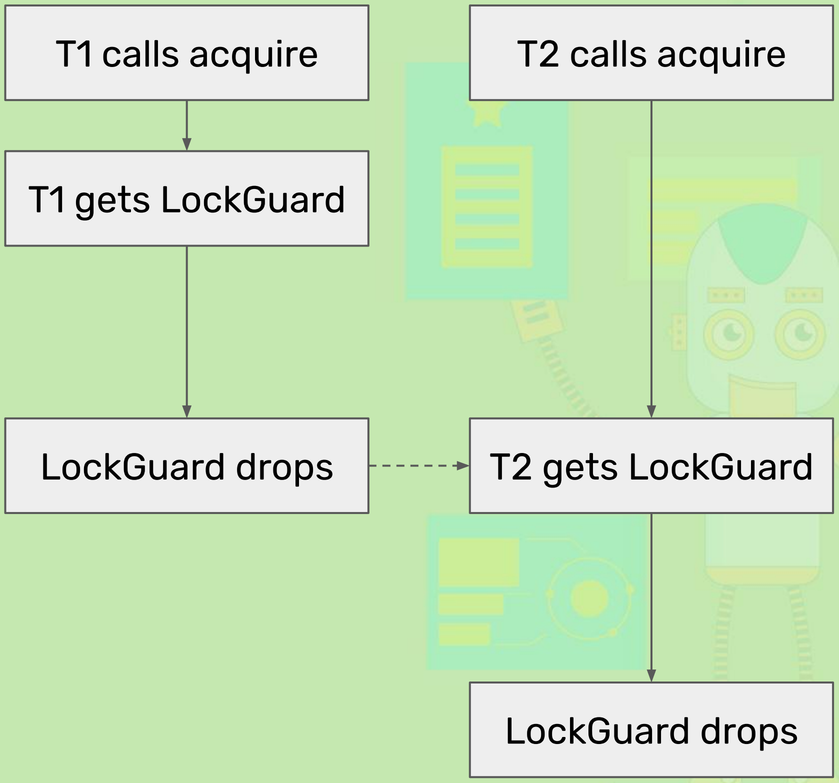
Only one at a time

```
LockGuard::deref_mut(&mut self) -> &mut T {
    &mut self.mutex.inner
}
```

Access to the resource

```
LockGuard::drop(&mut self) {
    self.mutex.locked = false;
}
```

Automatic clean-up



Section 2

RAII

RAII

Resource Acquisition Is Initialisation

Holding a valid resource is an invariant of the type.

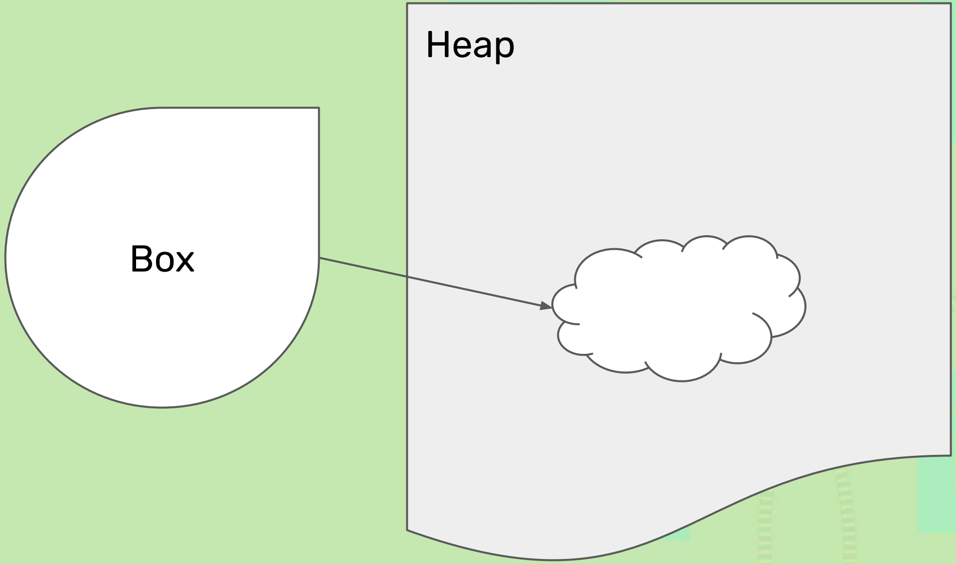
An instance is only acquired after successful allocation and initialisation of the resource.

The resource is deallocated at the end of the lifetime of the instance.

The existence of a Box instance is proof that the value is valid, on the heap and owned by Self

```
Box::new(value: T) -> Self {
    let ptr = alloc(…);
    ptr.write(value);
    Self(ptr)
}

Box::drop(&mut self) {
    self.ptr.drop();
    dealloc(self.ptr);
}
```

Drop is called at most once, so no double free



Guard

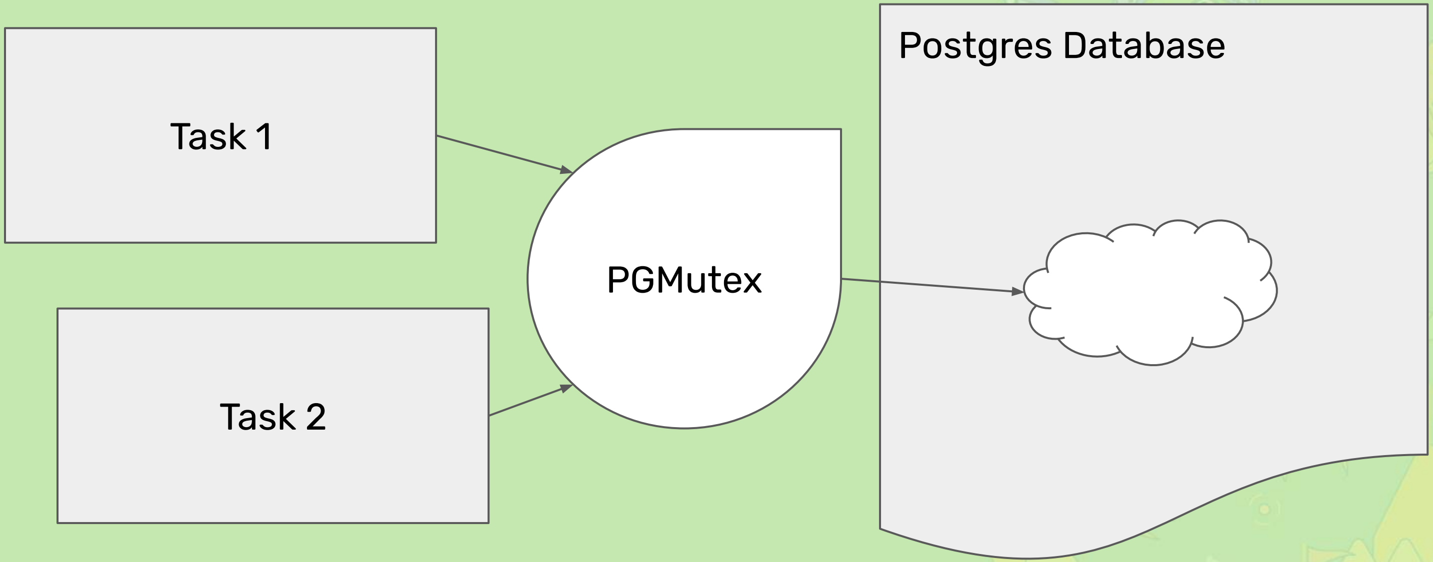
Object that manages a resource and provides compile-time proof of some invariants.

```rust
struct LockGuard<'a, T> {
    mutex: &'a Mutex<T>,
}
```

- constructed when mutex is successfully locked
- access to the underlying data
- compiler guarantees mutex outlives the guard
- mutex unlocked on drop

Section 3

PGMutex

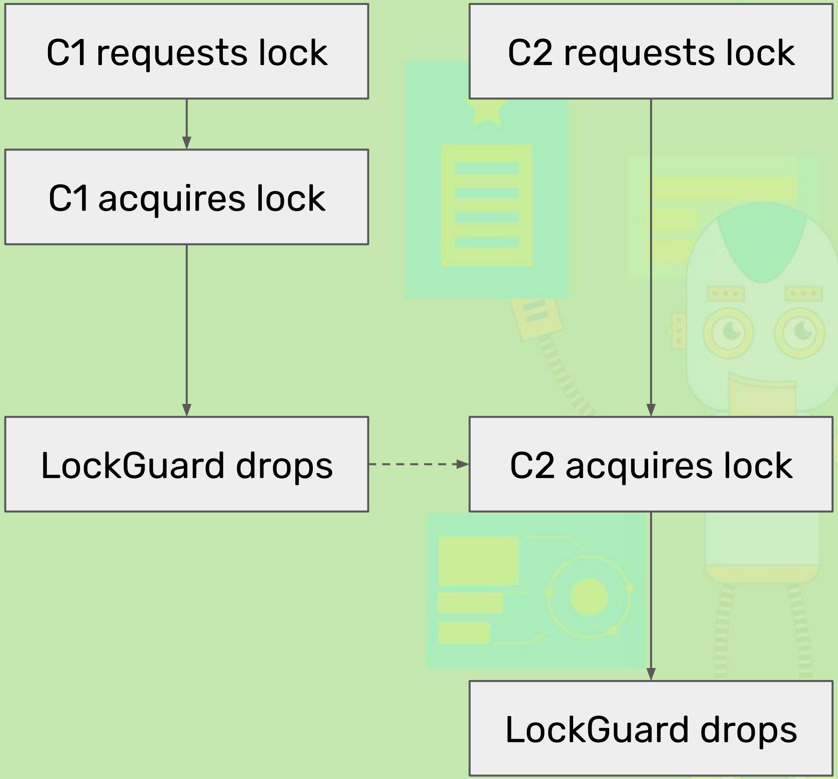

Task 1
Task 2
PGMutex
Postgres Database

```
PGMutex::acquire(
    conn: PGConn,
    id: UUID,
) -> LockGuard<T> {
    conn.set_advisory_lock(id).await;
    LockGuard { conn, id }
}

LockGuard::update(
    &mut self,
    value: T,
) {
    self.conn.update(self.id, value);
}

LockGuard::drop(&mut self) {
    self.conn.release_advisory_lock(
        self.id
    );
}
```

awaits until the DB has set the advisory lock

can only read/write if holding the advisory lock

one day, this will be async



EventStoreLock

We have used the PGMutex concept in the `event_sourcing.rs` library.

Implements pessimistic locking on an aggregate stream.

Source:
https://github.com/primait/event_sourcing.rs/blob/ce4fdf8fbf2e0f9b72c4a17796ce2372c78bc62d/src/store/postgres/event_store.rs#L176-L186

```rust
async fn lock(&self, aggregate_id: Uuid)
-> Result<EventStoreLockGuard, Self::Error> {
    let (key, _) = aggregate_id.as_u64_pair();
    let connection = self.inner.pool.acquire().await?;
    let lock_guard = PgStoreLockGuardAsyncSendTryBuilder {
        lock: PgAdvisoryLock::with_key(PgAdvisoryLockKey::BigInt(key as i64)),
        guard_builder: |lock: &PgAdvisoryLock| Box::pin(async move {
            lock.acquire(connection).await
        }),
    }
    .try_build()
    .await?;
    Ok(EventStoreLockGuard::new(lock_guard))
}
```

yields when the DB has set the lock

information about the lock

```rust
struct PgStoreLockGuard {
    lock: PgAdvisoryLock,
    #[borrows(lock)]
    #[covariant]
    guard: PgAdvisoryLockGuard<
      'this, Connection<Postgres>
    >,
}
```

captures the connection

calls the DB and releases the advisory lock on drop

async operation, flushed as the connection returns to the pool

Lessons learnt

Connection pool exhaustion: `acquire` hogs the connection until the advisory lock is set, without doing any work.

Learn from std: Mutex has a `try_acquire` that returns immediately if the lock is not acquired.

Synchronisation is hard: on high contention, optimistic locking might be more suitable.

Section 4

Kernel ScopeGuard

Kernel ScopeGuard

Source: https://rust.docs.kernel.org/kernel/types/struct.ScopeGuard.html

```rust
pub struct ScopeGuard<F: FnOnce>(Option<F>);

impl<F: FnOnce> Drop for ScopeGuard<F> {
    fn drop(&mut self) {
        if let Some(f) = self.0.take() {
            f()
        }
    }
}

impl<F: FnOnce> ScopeGuard<F> {
    pub fn dismiss(mut self) {
        self.0 = None;
    }
}
```

wraps a FnOnce when constructed

which is executed on drop

unless explicitly dismissed

consumes self

drop is called here, but the callback has been removed

ScopeGuard

```
fn example(arg: bool) {
    let log = ScopeGuard::new(|| pr_info!("example returned early\n"));

    if arg {
        return;
    }

    // (Other early returns...)

    log.dismiss();
}
```

closure executed here

and here

disarm the callback

no logs here

ScopeGuard

```
fn example(arg: bool) -> Result {
    let mut vec =
        ScopeGuard::new_with_data(Vec::new(),
            |v| pr_info!("vec had {} elements\n", v.len()));

    vec.push(10u8, GFP_KERNEL)?;
    if arg {
        return Ok(());
    }

    vec.push(20u8, GFP_KERNEL)?;
    Ok(())
}
```

can manage data

logs 1 element

logs 2 elements

Section 5

Final remarks

Caveats

Affine types: Rust guarantees `drop` is invoked *at most* once. It is safe and allowed to `forget` an object, which will not call its destructor.

No async drop: currently it is not possible to run async code in the destructor, which might be desirable for clean-up operations.

Summary

Guards provide proof of some invariants for the data they *manage*, during their lifecycle.

Drop ensures *automatic* cleanup when the Guard goes out of scope, or on panic.

Thank you!