

# Threading | Deadlocks

# ■ Deadlock

- ◆ A deadlock is a situation where locks are waiting on one another
  - Threads become “stuck” and are unable to continue
- ◆ Deadlocks can occur when:
  - Using multiple locks
  - Recursing while taking a lock
  - Locking the same lock twice

# Recursive Deadlock Example

```
use parking_lot::Mutex;
```

```
fn recurse(  
    data: Rc<Mutex<u32>>,  
    remaining: usize,  
) -> usize {  
    let mut locked = data.lock();  
    match remaining {  
        rem if rem == 0 => 0,  
        rem => recurse(Rc::clone(&data), rem - 1),  
    }  
}
```

# ■ Fix Deadlock – ReentrantMutex

```
use parking_lot::ReentrantMutex;

fn recurse(
    data: Rc<ReentrantMutex<u32>>,
    remaining: usize,
) -> usize {
    let mut locked = data.lock();
    match remaining {
        rem if rem == 0 => 0,
        rem => recurse(Rc::clone(&data), rem - 1),
    }
}
```

# Threaded Deadlock Example

```
type ArcAccount = Arc<Mutex<Account>>;
```

```
struct Account {  
    balance: i64,  
}
```

```
fn transfer(from: ArcAccount, to: ArcAccount, amount: i64) {  
    let mut from = from.lock();  
    let mut to = to.lock();  
    from.balance -= amount;  
    to.balance += amount;  
}
```

```
let t1 = thread::spawn(move || {  
    transfer(a, b, 500);  
});  
let t2 = thread::spawn(move || {  
    transfer(b, a, 800);  
});
```

# ■ Fix Deadlock – Retry On Failure

```
fn transfer(from: ArcAccount, to: ArcAccount, amount: i64) {  
    loop {  
        if let Some(mut from) = from.try_lock() {  
            if let Some(mut to) = to.try_lock() {  
                from.balance -= amount;  
                to.balance += amount;  
                return;  
            }  
        }  
        thread::sleep(Duration::from_millis(2));  
    }  
}
```

# Thread Contention / Backoff

```
use backoff::ExponentialBackoff;

fn transfer(from: ArcAccount, to: ArcAccount, amount: i64) {
    let op = || {
        if let Some(mut from) = from.try_lock() {
            if let Some(mut to) = to.try_lock() {
                from.balance -= amount;
                to.balance += amount;
                return Ok(());
            }
        }
        Err(0)?
    };
    let backoff = ExponentialBackoff::default();
    backoff::retry(backoff, op);
}
```

# ■ Recap

- ◆ Deadlocks are permanently stuck locks
- ◆ ***ReentrantMutex*** allows multiple locks from the same thread
  - Use for recursive functions
  - Anytime you need to lock the same lock more than once
- ◆ ***try\_lock()*** can prevent deadlocks
  - Drop all locks used in function and try again after a short period
    - ▶ Use the ***backoff*** crate for optimal performance