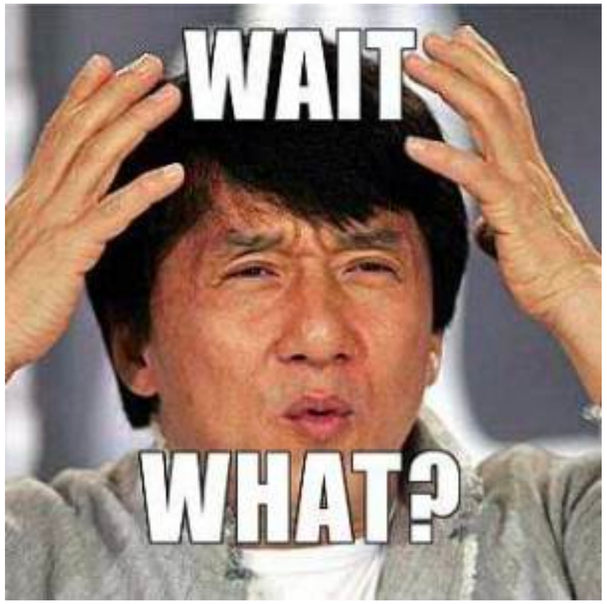
RD

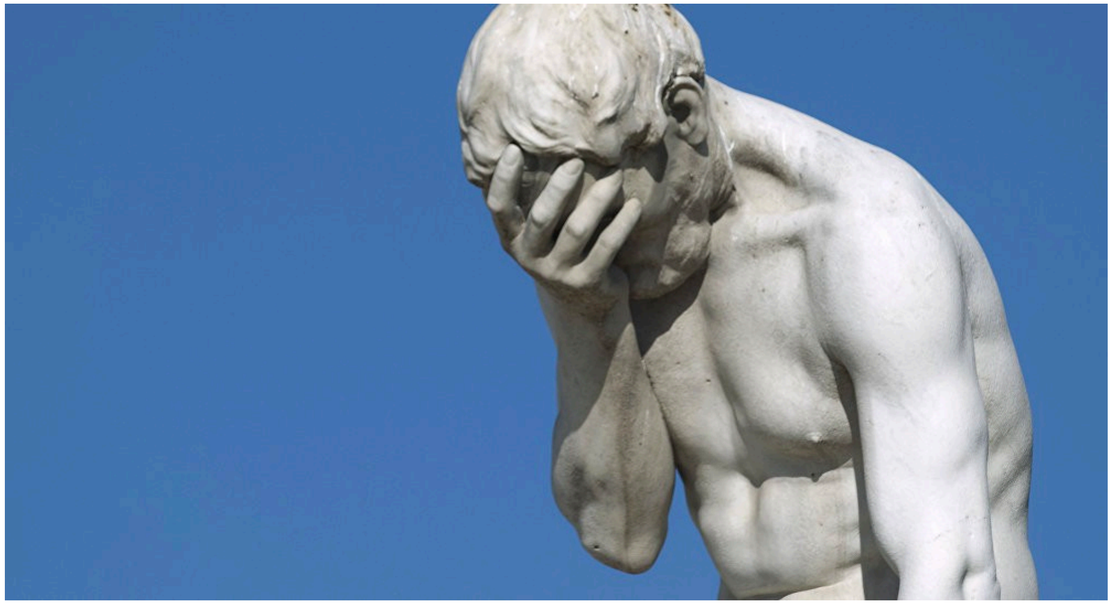
Ideas and solutions behind JetBrains Rider

Dmitry Ivanov, JetBrains

Начнём год с погружения в технические решения, лежащие в основе Rider. Рассмотрим в деталях фреймворк JetBrains RD (Reactive Distributed) в котором собрано огромное количество нестандартных решений. За глубину российских инноваций будет отвечать Дмитрий Иванов — гуру, мыслитель, техлид.

Из анонса выступления





Что расскажу…

1. Идеи и инсайты
2. Проблемы и решения
3. Технические приёмы
4. Конкретные библиотеки
   - https://github.com/JetBrains/rd
   - NuGet **JetBrains.Lifetimes**
   - NuGet **JetBrains.RdFramework**
5. Как попасть в JetBrains?

Как попасть в JetBrains ?

Прямой путь

- Подтянуть алгоритмы, структуры данных, многопоточность
- Потренить 2 недели перед собесом на *Codeforces*, *Topcoder*
- *Хорошо знать языки программирования*
- Решить задачку на дизайн (сегодня покажу пример)
- Сделать тестовое задание

Окольный путь

- Сделать плагин для ReSharper/Rider
- Показать очень крутое продуктовое мышление
- Контрибьютить в OpenSource

LIFETIMES

NuGet

JetBrains.Lifetimes 2019.3.0

JetBrains Core library for graceful disposal, concurrency and reactive programming

Проблема: управление ресурсами

```
IDisposable TakeSomeResource()
```

Пример:

```
IDisposable Observable<T>.Observe(Action<T> handler)
```

```
event += handler
```

Отписка?

Многопоточность?

Нет ни одной подписки?

Проблемы с IDisposable

IDisposable TakeSomeResource()

- Забыли вызвать
- Вызвали 2 раза
- Куда складировать?
- Кто отвечает за терминацию?
- Зависимости на ресурсах (порядок терминации)

```
f1 = Take1();
f2 = Take2(f1);
…

void Dispose() {
  f1.Dispose();
  f2.Dispose();
}
```

Решение IntellijIDEA

Статический **Disposer**

Disposer.register(parentDisposable, childDisposable)

Disposer.dispose(disposable)

Вручную вызывать нельзя

Решение ReSharper

Lifetimes

IDisposable Observable<T>.Observe(Action<T> handler)

OLD

1. **Observable.Observe(handler) =>** *handlers*.Add(handler)
2. **Observable.Dispose() =>** *handlers*.Remove(handler)

Observable<T>.Observe(**Lifetime** lifetime, Action<T> handler)

NEW

```
{
  handlers.Add(handler);
  lifetime.OnTermination(() => handlers.Remove(handler))
}
```

Решение ReSharper

Lifetimes

IDisposable Observable<T>.Observe(Action<T> handler)

OLD

1. **Observable.Observe(handler) =>** *handlers*.Add(handler)
2. **Observable.Dispose() =>** *handlers*.Remove(handler)

ISource<T>.**Advise**(**Lifetime** lifetime, Action<T> handler)

NEW

```
{
  handlers.Add(handler);
  lifetime.OnTermination(() => handlers.Remove(handler))
}
```

Атомарные скобки

Lifetimes

ISource<T>.Advise(**Lifetime** lifetime, Action<T> handler)

```
{
  lifetime.Bracket(
    () => handlers.Add(handler),
    () => handlers.Remove(handler)
  )
}
```

can work with terminated lifetime

ICollection.Add(lifetime, handler);

LifetimeDefinition

LifetimeDefinition: IDisposable

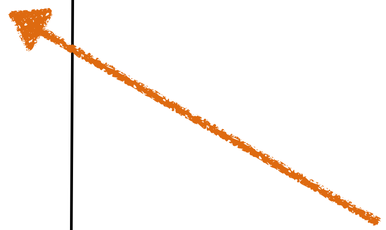
void Terminate()

LifetimeDefefinition(Lifetime parent = default)

Lifetime

Action[] terminationActions

void OnTermination(Action)



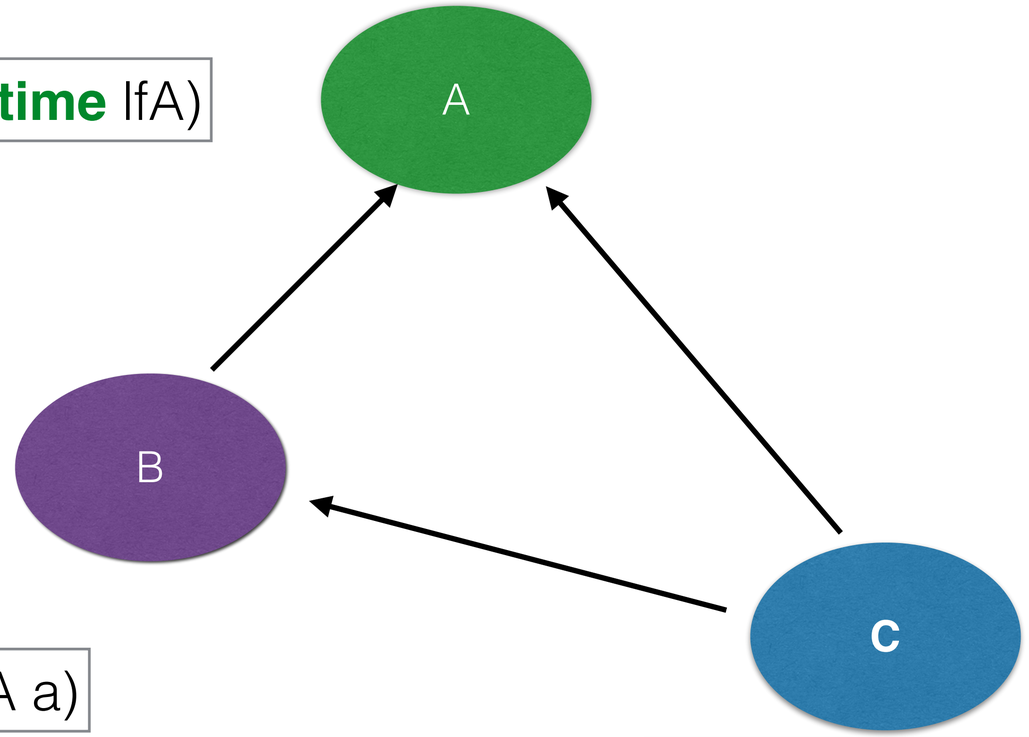
```
using (var def = new LifetimeDefinition()) {
    DoSomething(def.Lifetime);
}
```

```
parent.OnTermination(() => child.Terminate())
```

Component Container

Lifetime - special object, created by CC

ctor A(**Lifetime** lfA)



ctor B(**Lifetime** lfB, A a)

ctor C(**Lifetime** lfC, A a, B b)

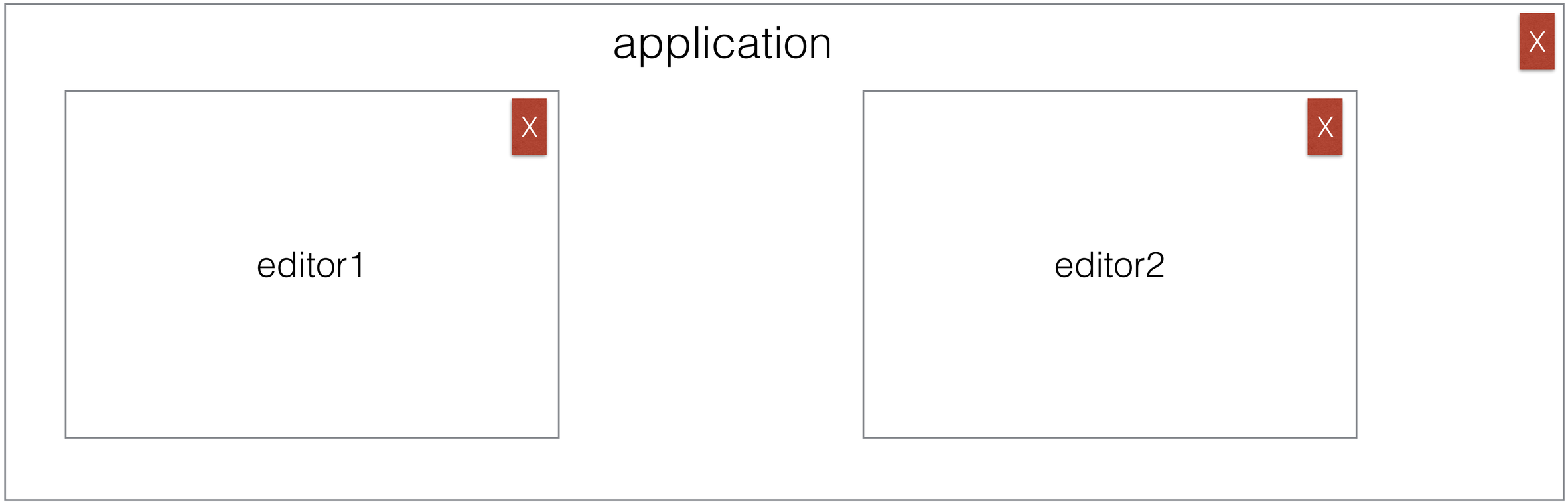
Lifetime vs Definition

- **Lifetimes** *usually* come from outer code: don't create **LifetimeDefinitions** manually except it absolutely necessary (say UI control with 'Close' button).
- If you own **LifetimeDefinition** you must terminate it or pass someone for termination. Otherwise use **Lifetime**.
- Sources for lifetimes:
  - **using** (var **def** = **new** LifetimeDefinition()) { … **def.Lifetime** …}
  - **Component container**
  - SequentialLifetimes.Next(): 1.def.Terminate(); 2. def = new LifetimeDefinition()
  - IViewable<T>.View(**Lifetime**, Action<**Lifetime**, T>))

View

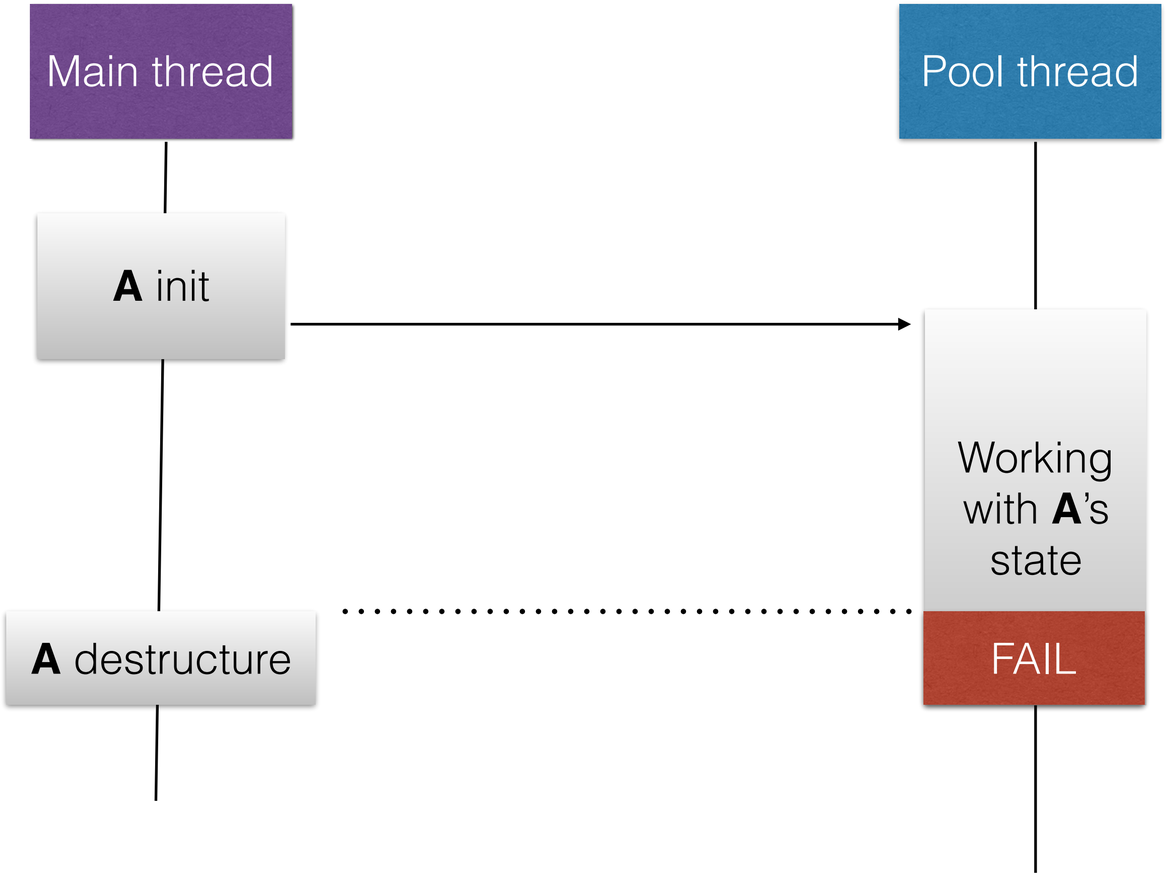
ViewableSet<Editor> editors = *ObservableSet + View method*

editors.**View**(applicationLifetime, (editorLifetime, editor) => …)



MULTITHREADING

Проблема: доступ к ресурсам из другого потока



Atomic execution

void lifetime.**Execute**(action)

lifetime.IsAlive - не дает начаться терминации ресурсов

lifetime.IsNotAlive - не выполняется

Task lifetime.**StartAttached**(scheduler, action)

Task lifetime.**Start**(scheduler, action)

lifetime *is* CancellationToken

~~TaskFactory.~~StartNew(action, lifetime, scheduler)

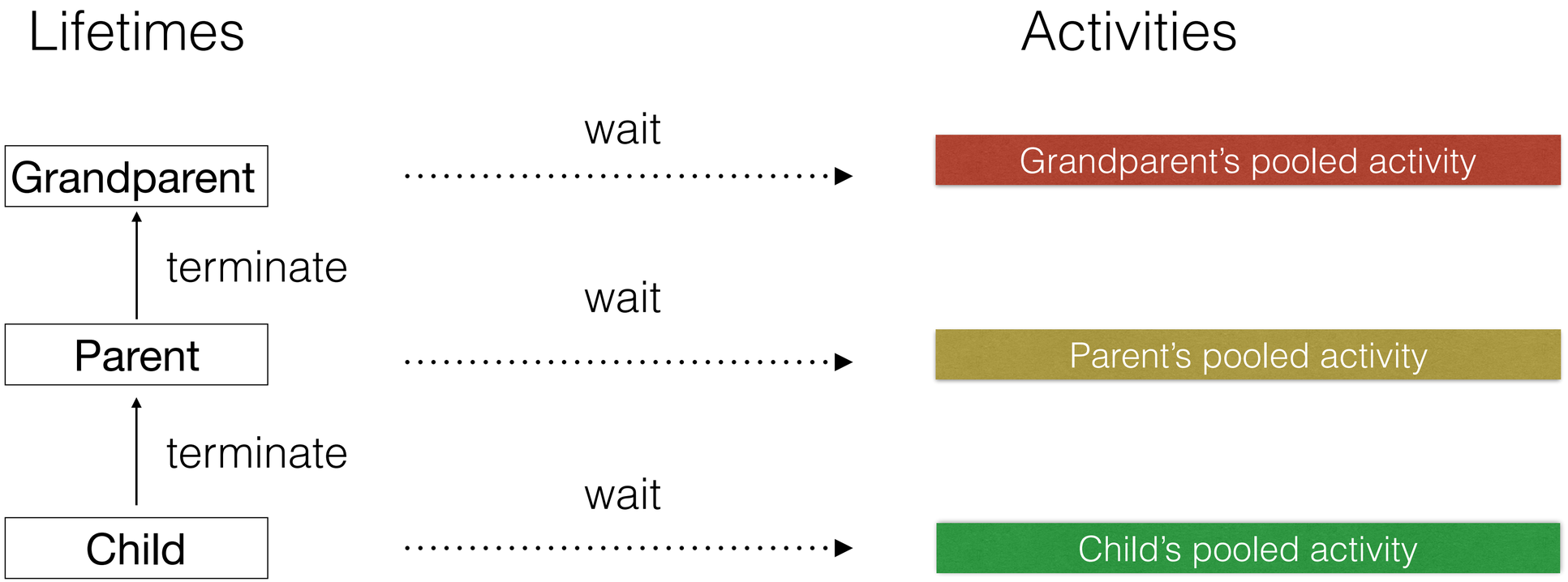
void lifetime.ThrowIfCanceled()

Lifetime.AsyncLocal.ThrowIfCanceled

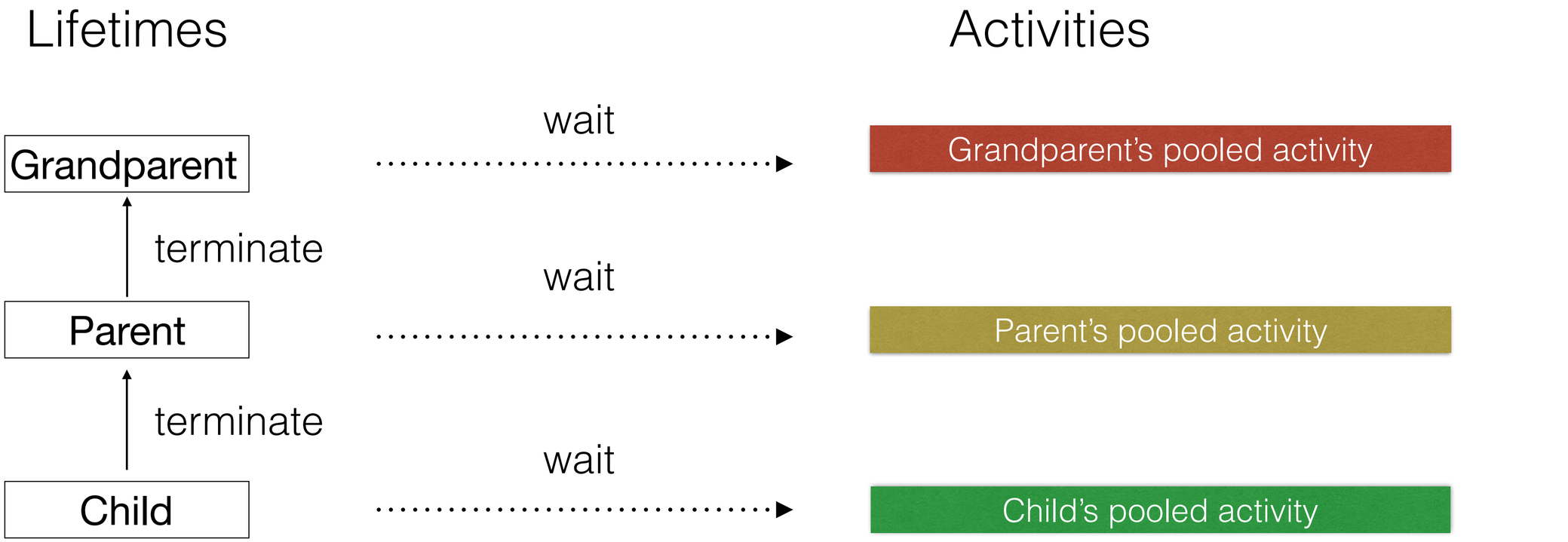
Пример использования

```
Task<T> Lifetime.ExecuteAsync(Func<Task<T>> asyncFunc)


return lifetime.ExecuteAsync(async () => {}
    var  x = ourComponent.Foo(); //component is guarded by lifetime
    await Task.Delay(200, lifetime);
    lifetime.ThrowIfTerminated();
    var y = ourComponent.Boo();
    return x + y
}}

BooVeryDeepInside() {
    Lifetime.AsyncLocal.ThrowIfTerminated();
}
```

Проблема: долгое закрытие



Решение: статус Canceled



Lifetime state
Action[] resources
bool isTerminated

Status
Alive
Canceled
Terminating
Terminated

Lifetime state
object[] resources
Status status
int executing

Resource types
LifetimeDefinition
Action
IDisposable

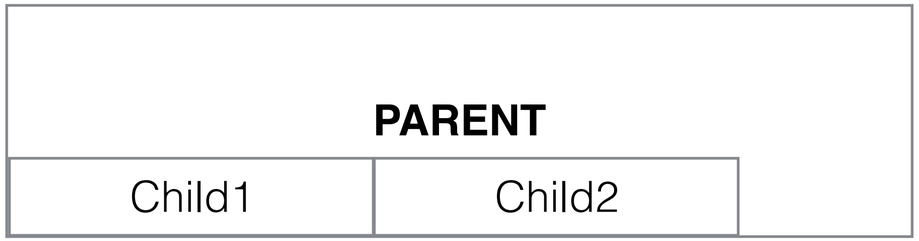
Алгоритм Terminate

```
void Terminate() {
    if (Status > Canceled) return
    MarkCancellationRecursively() // for subtree: Status = Canceled
    WaitForExecutions(timeout)
    Status = Terminating // from this point adding resource throw exception
    resources.ForEachReversed ( item => {
        Log.Catch (() => {
            1.      Action a -> a()
            2.      LifetimeDefinition ld -> ld.Terminate()
            3.      IDisposable d -> d.Dispose()
        }
    }
    Status = Terminated
}
```

ТЕХНИЧЕСКИЕ РЕШЕНИЯ

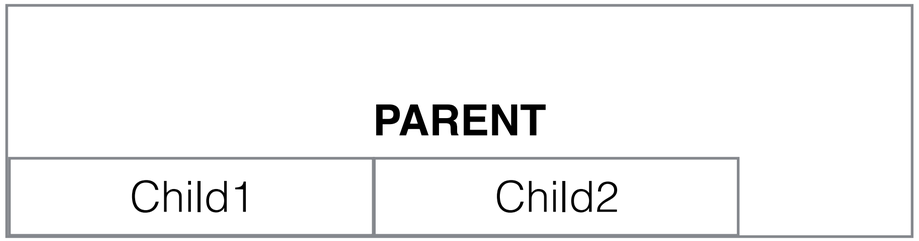
Проблема: ссылка вверх

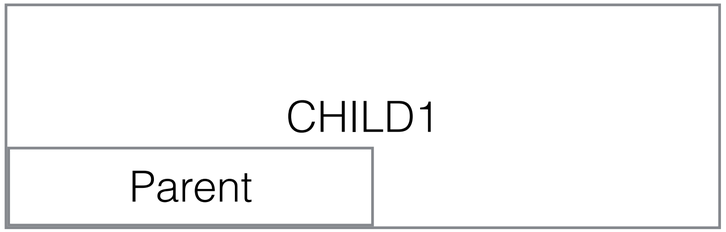
Parent.Terminate()

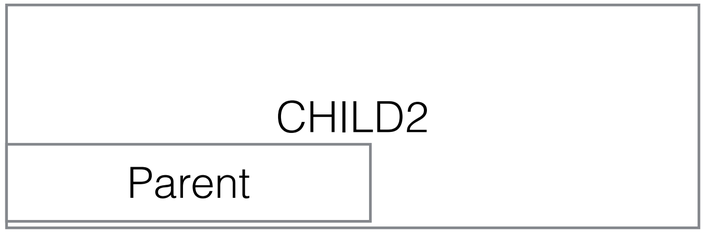


Проблема: ссылка вверх

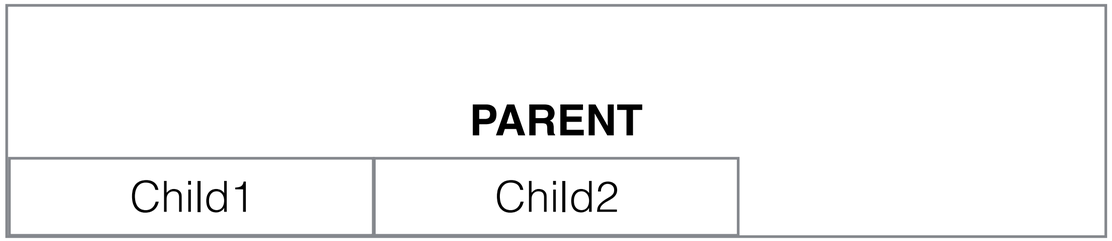
CHILD2.Terminate()







Решение: очищать амортизированно



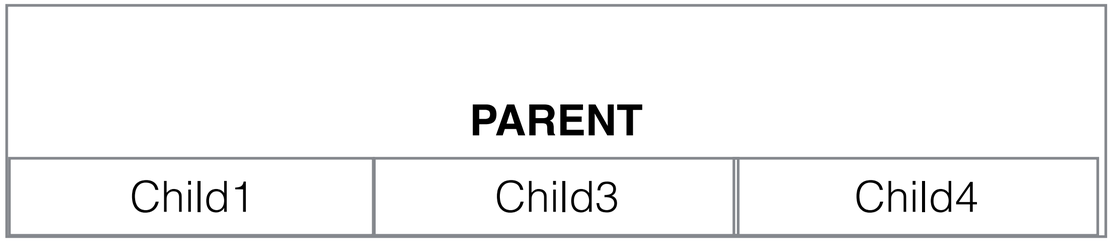
24 bytes

Решение: очищать амортизированно

24 bytes

Решение: очищать амортизированно

Решение: очищать амортизированно

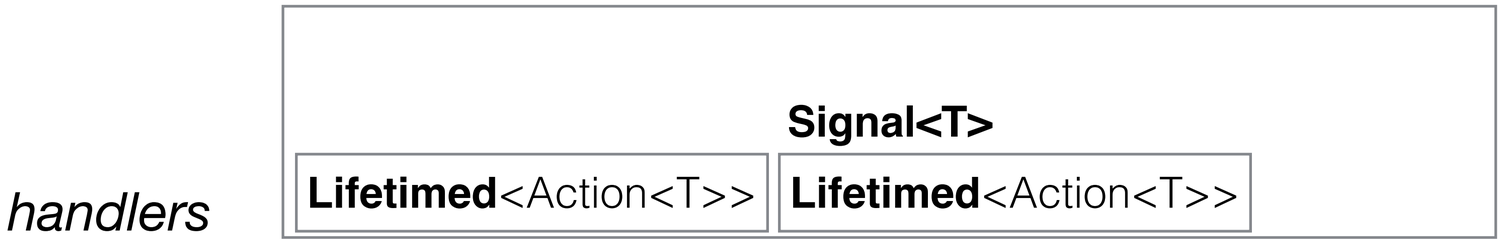


Подход для Observable

lifetime.OnTermination(() => *handlers*.Remove(handler))

Решение: Lifetimed

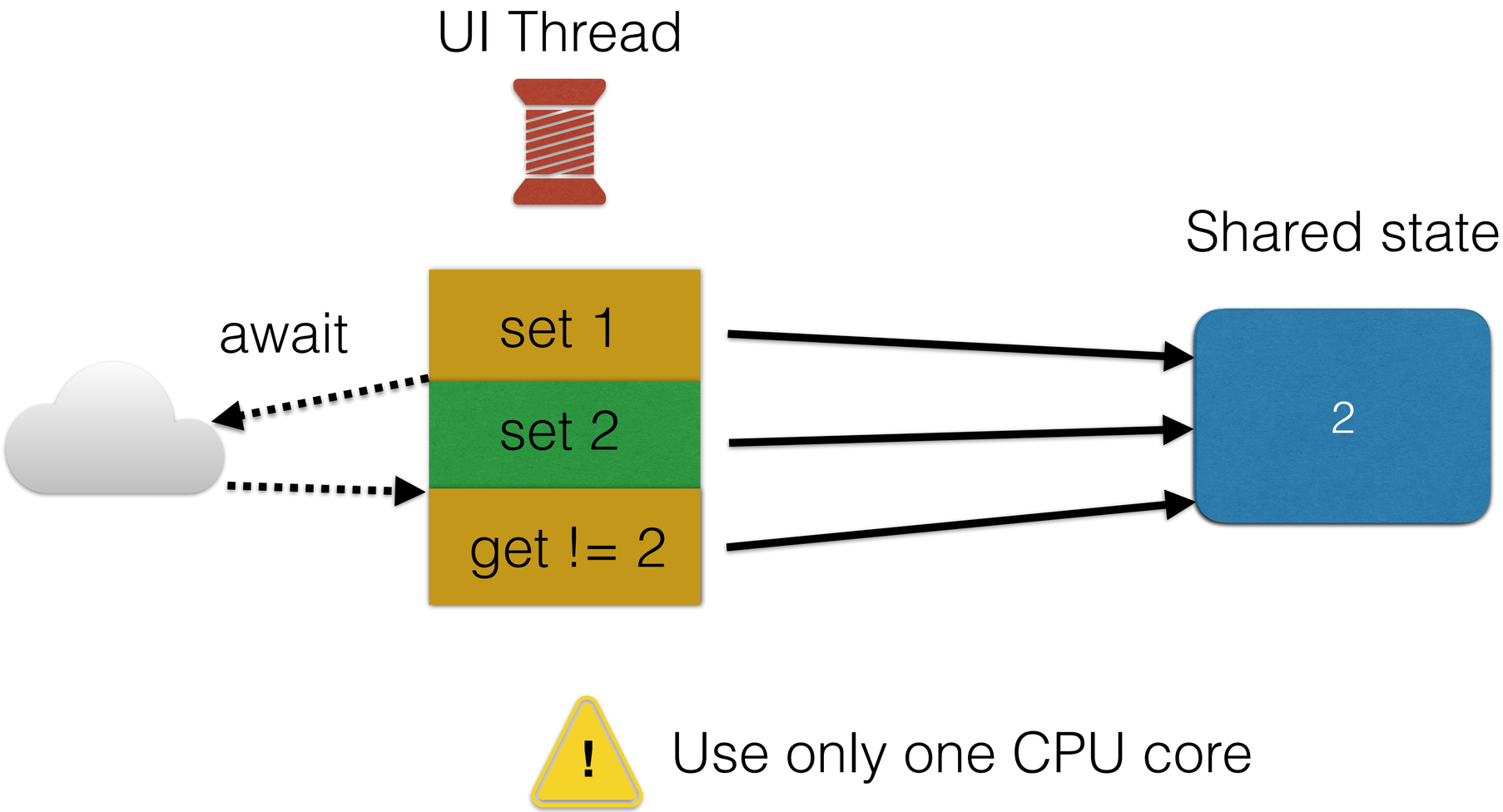
```
Lifetimed<T> : IDisposable {
    Lifetime lifetime;
    T Value;
    Dispose() => Value = default;
}
```



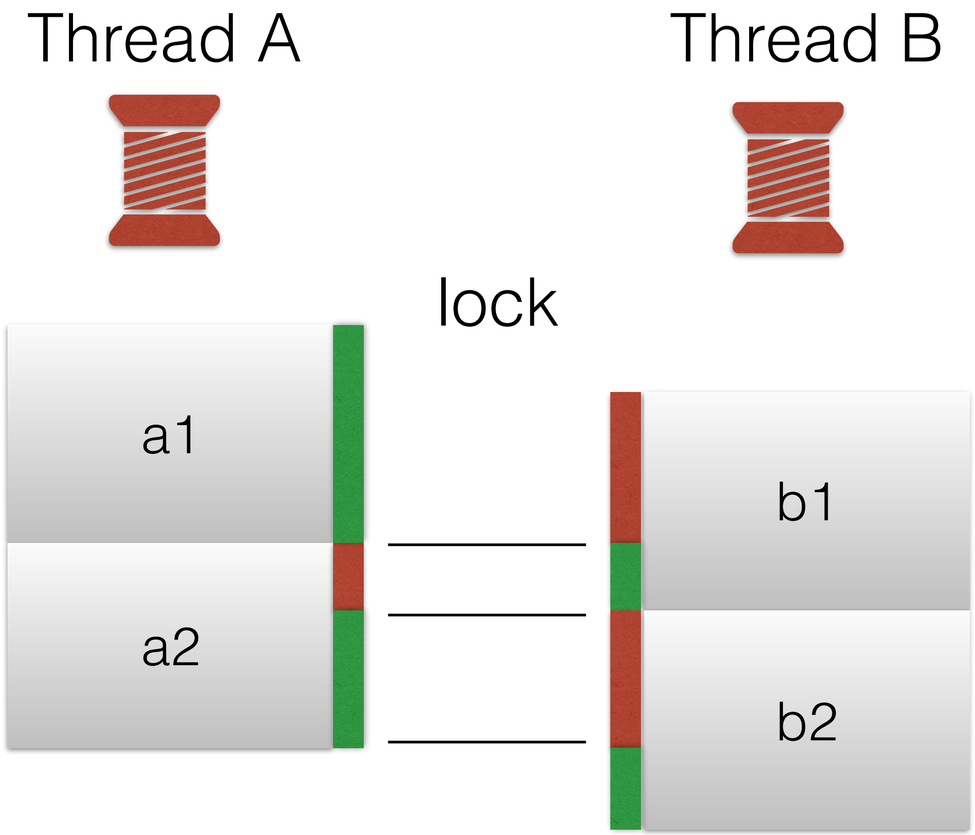
lifetime.OnTermination(*lifetimed*)

THREADING MODELS

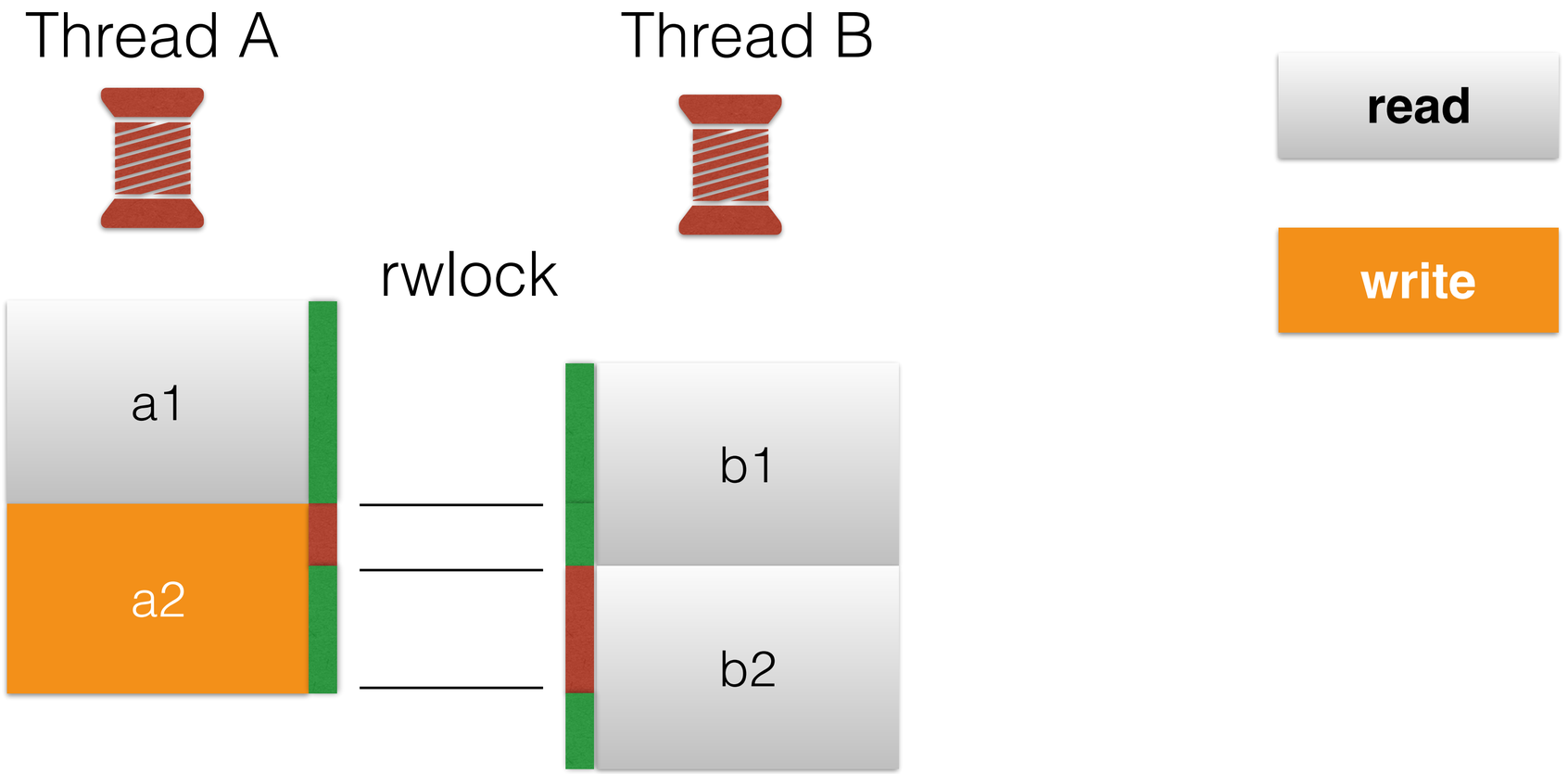
Single thread



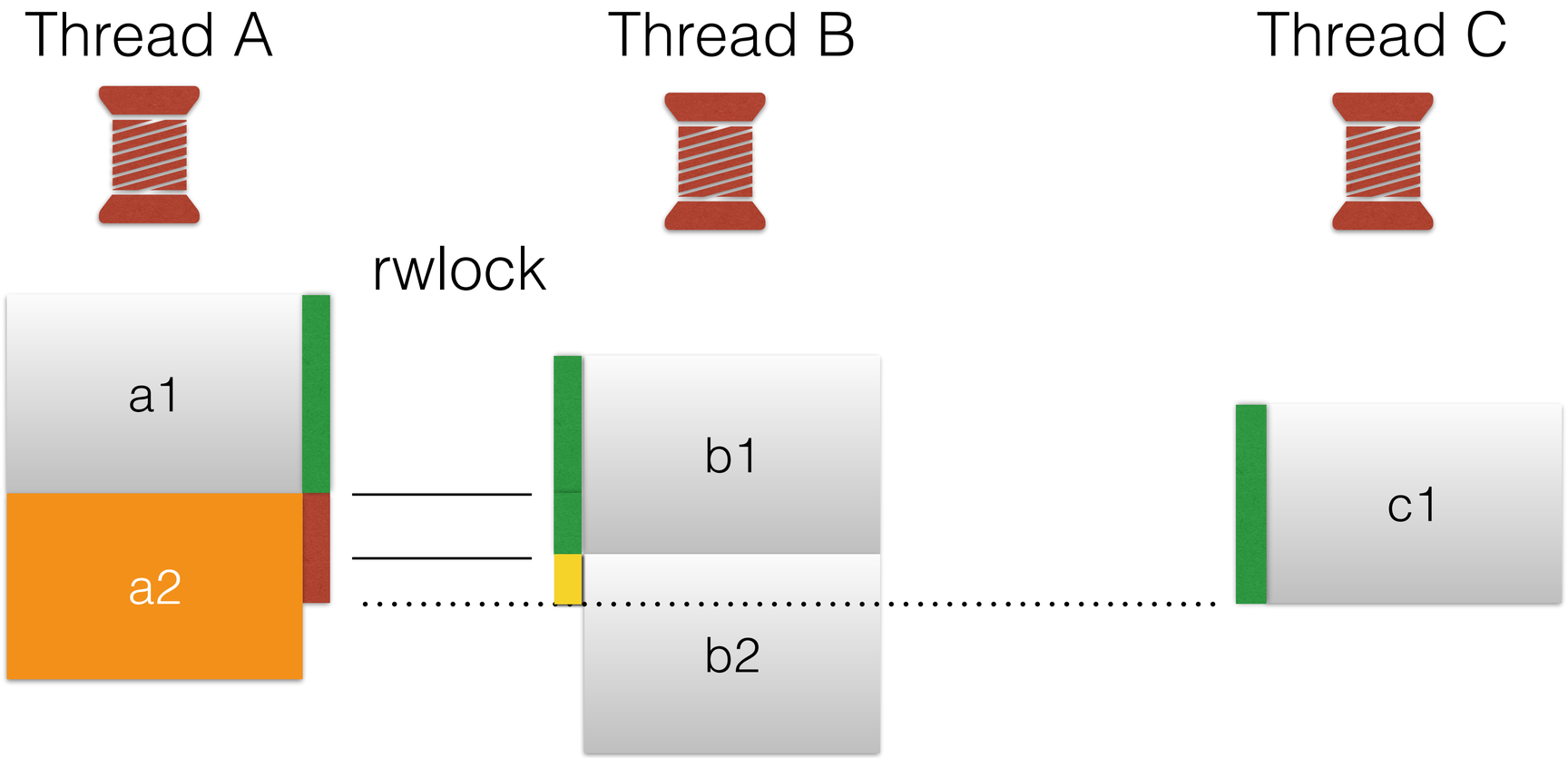
Pessimistic locking



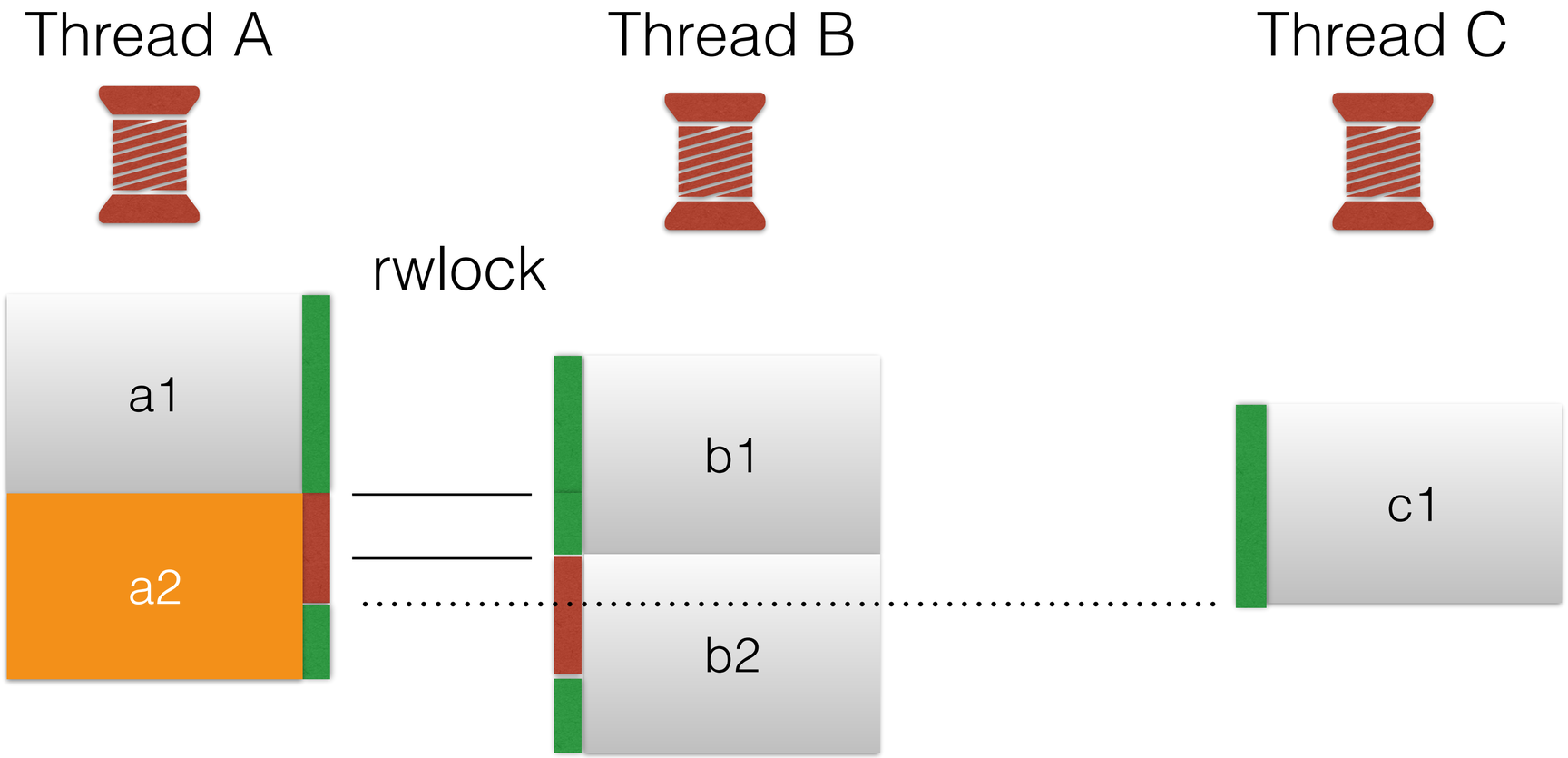
Readers-writer lock



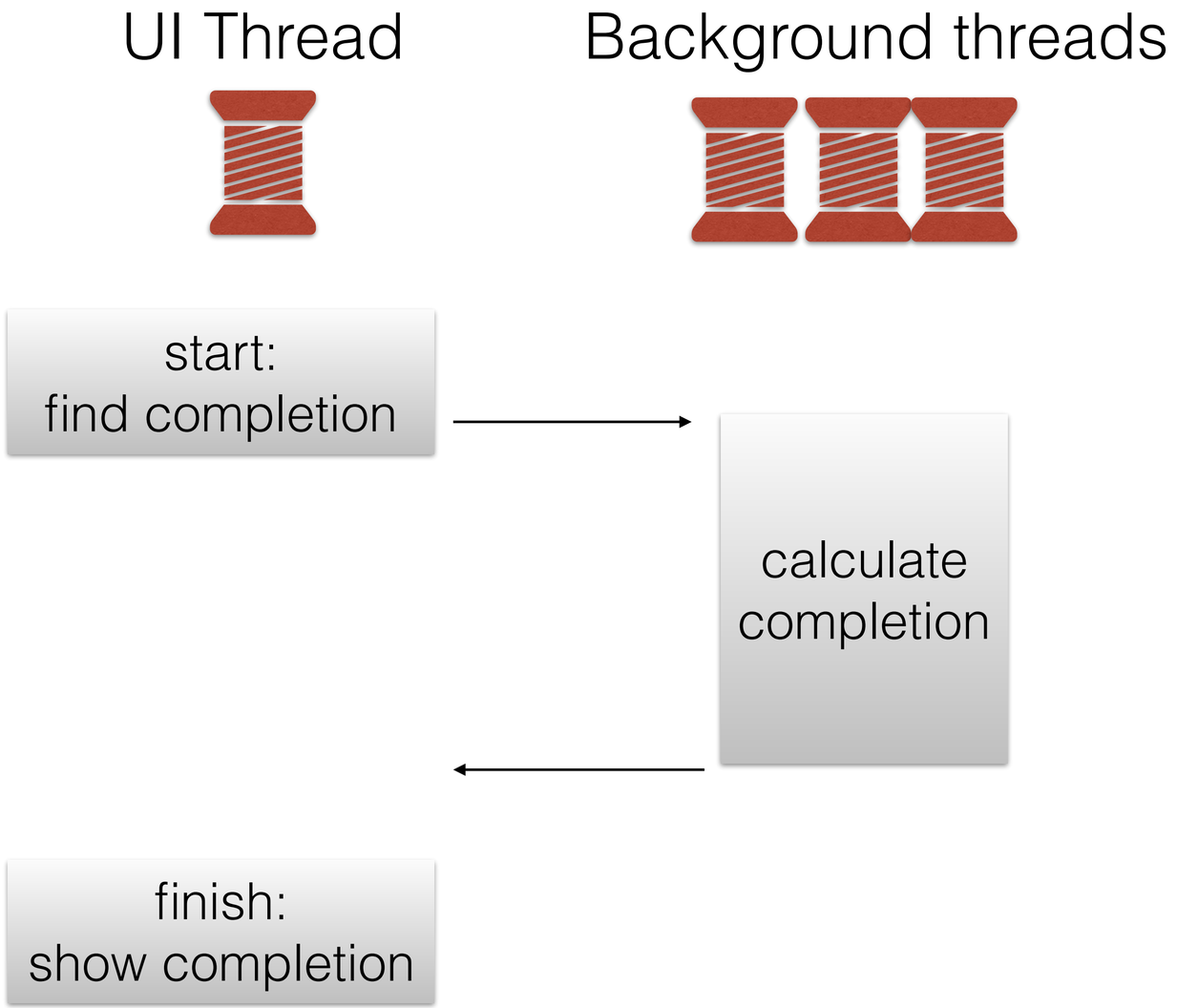
Readers-writer lock



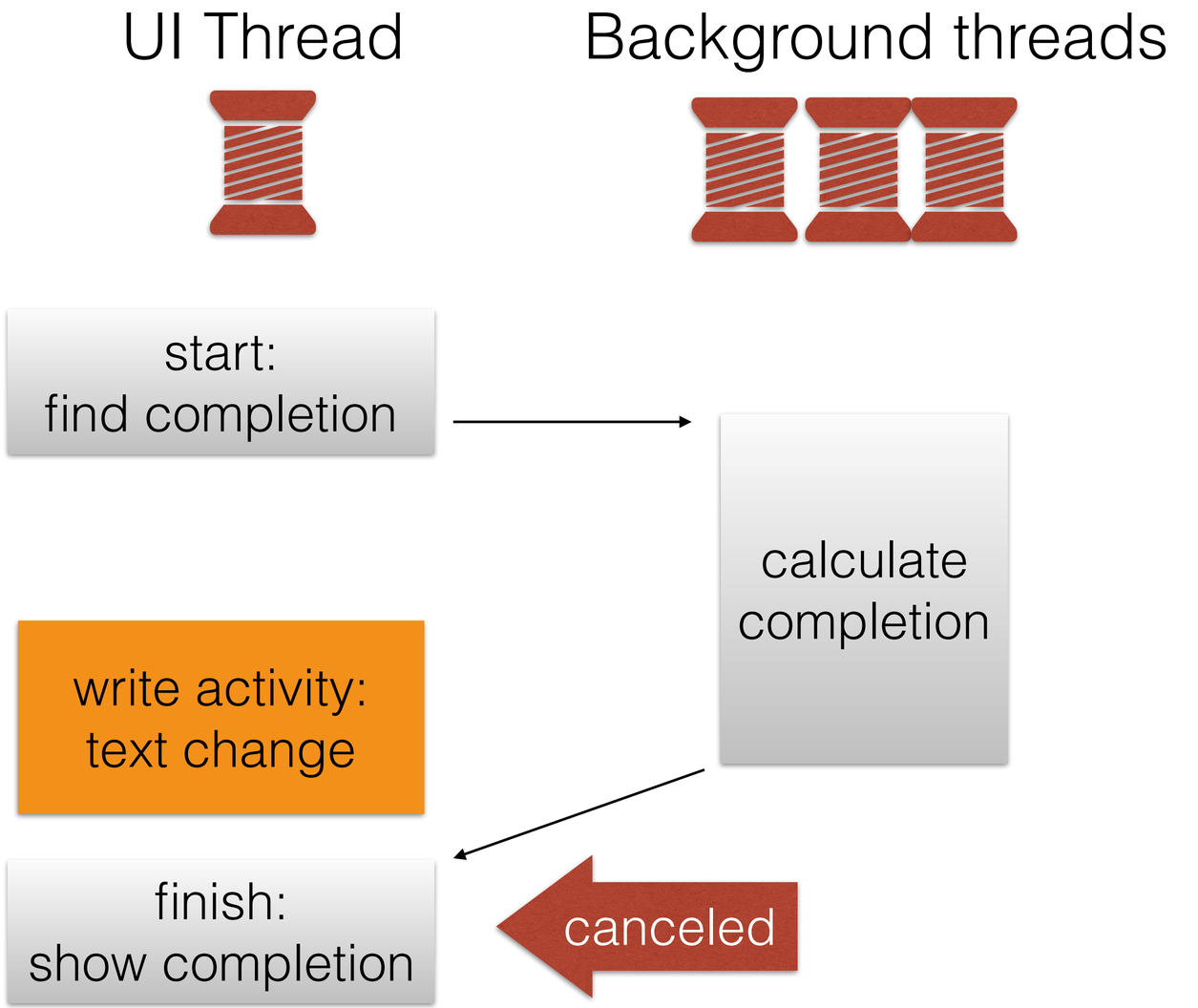
Readers-writer lock



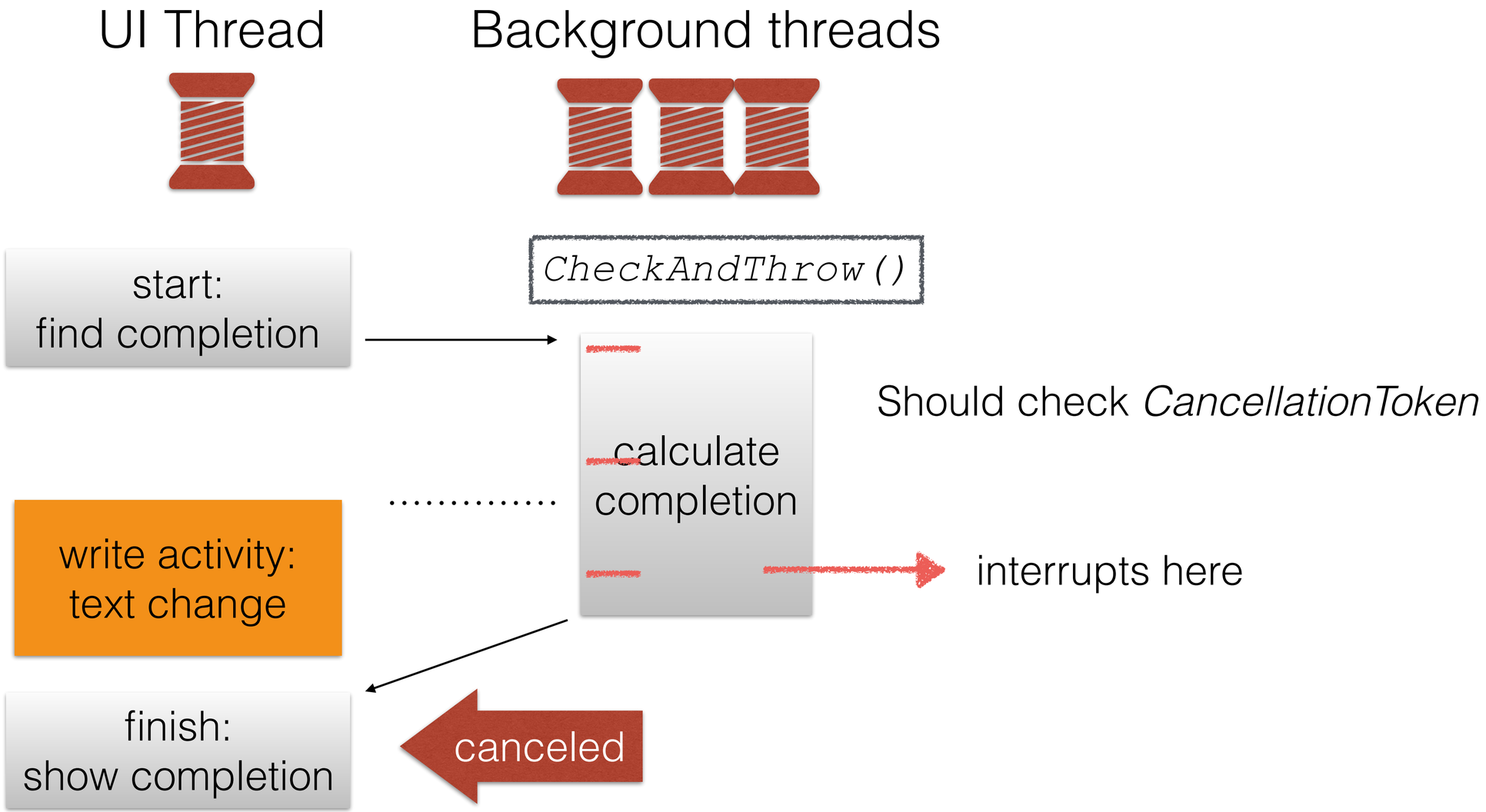
ContentModelReadWriteLock



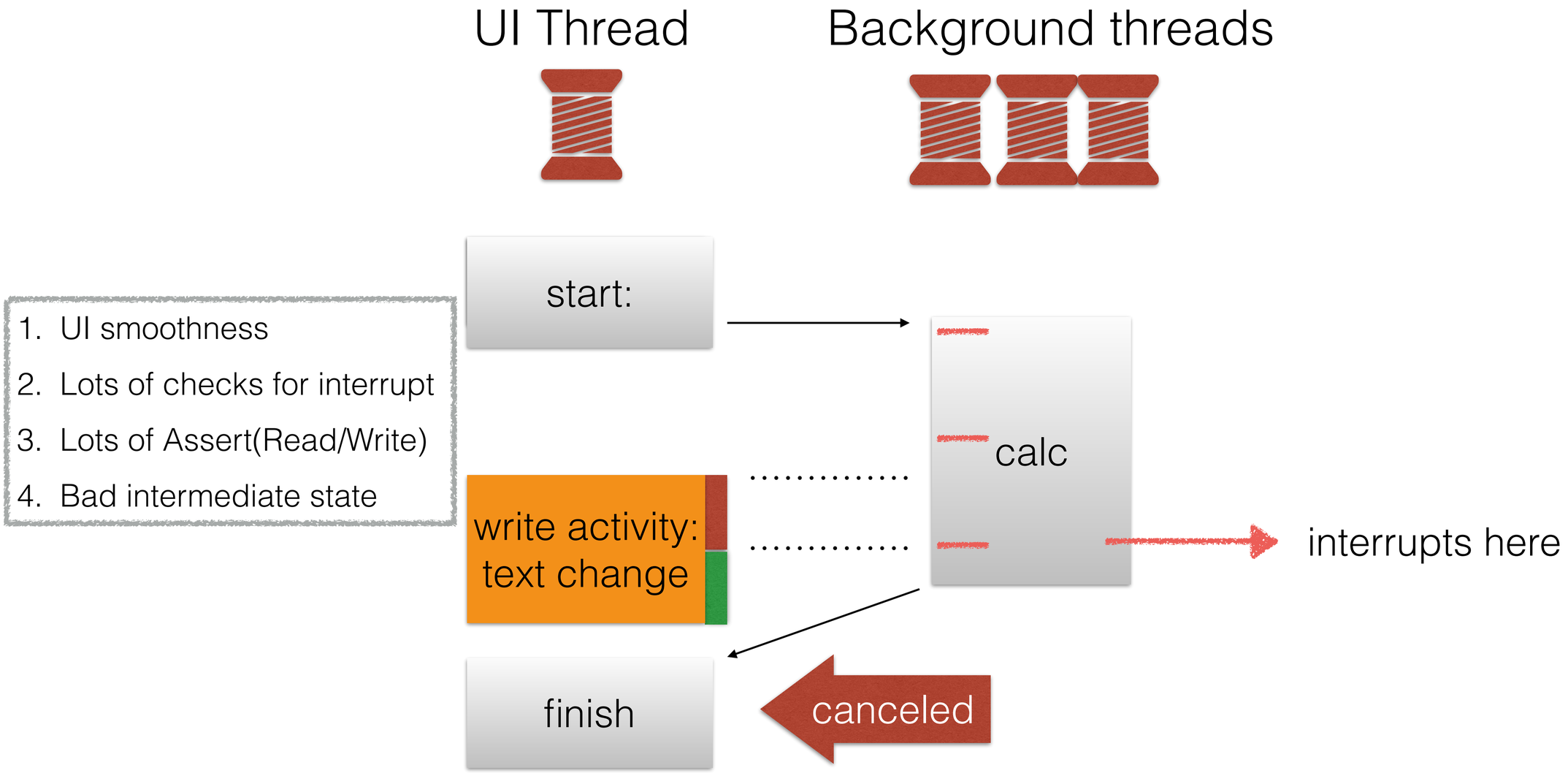
ContentModelReadWriteLock



InterruptableReadActivity

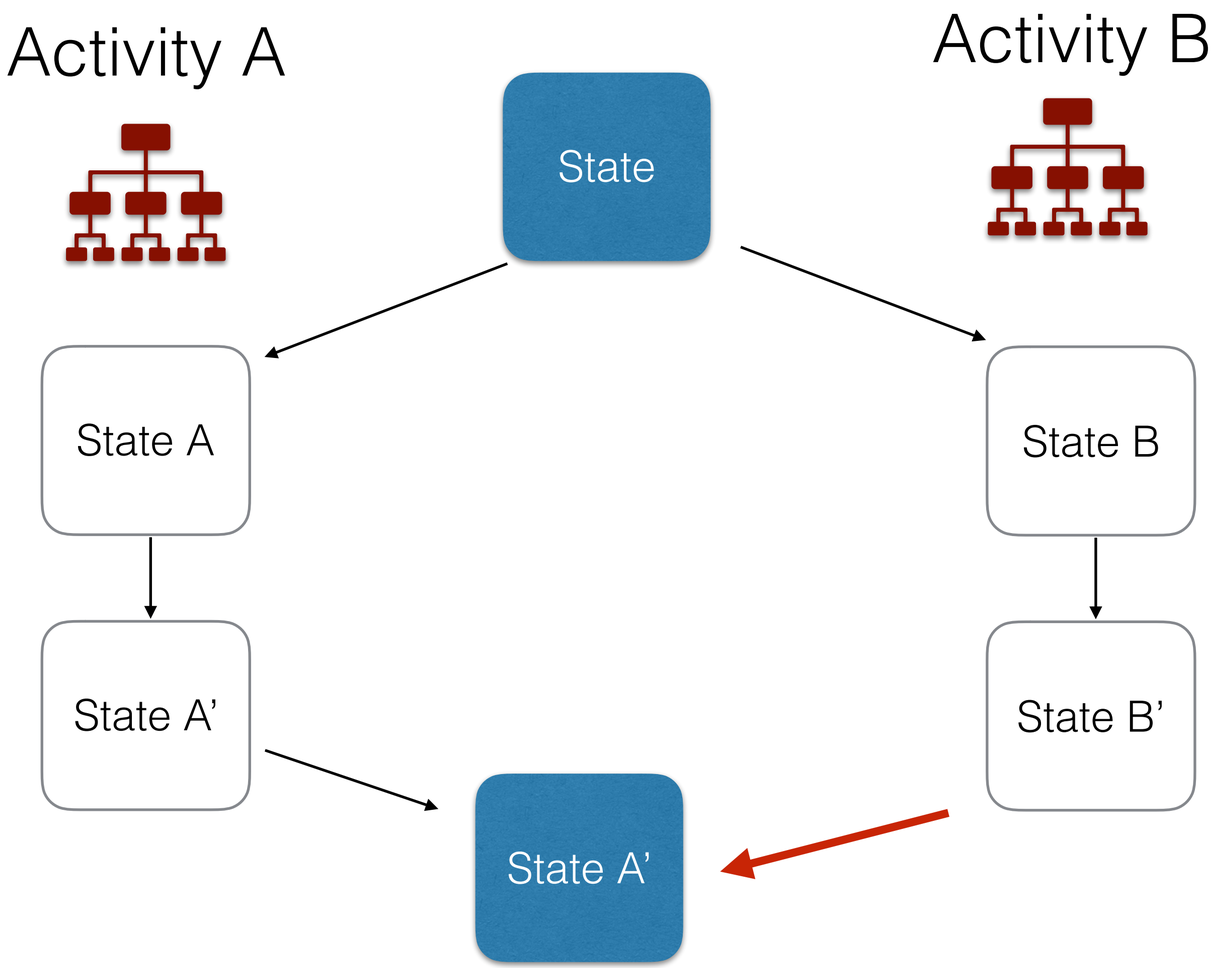


Problems with model

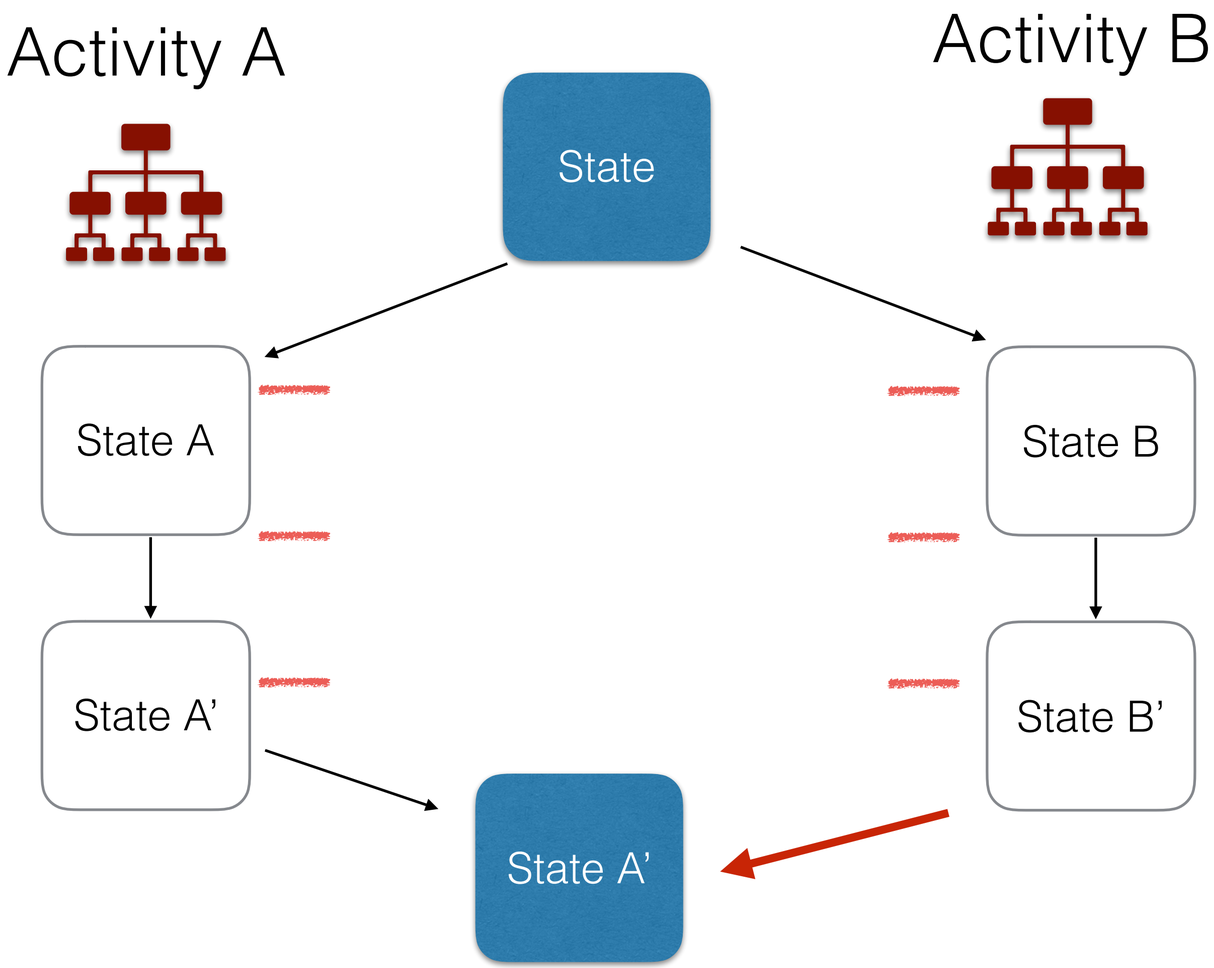


1. UI smoothness
2. Lots of checks for interrupt
3. Lots of Assert(Read/Write)
4. Bad intermediate state

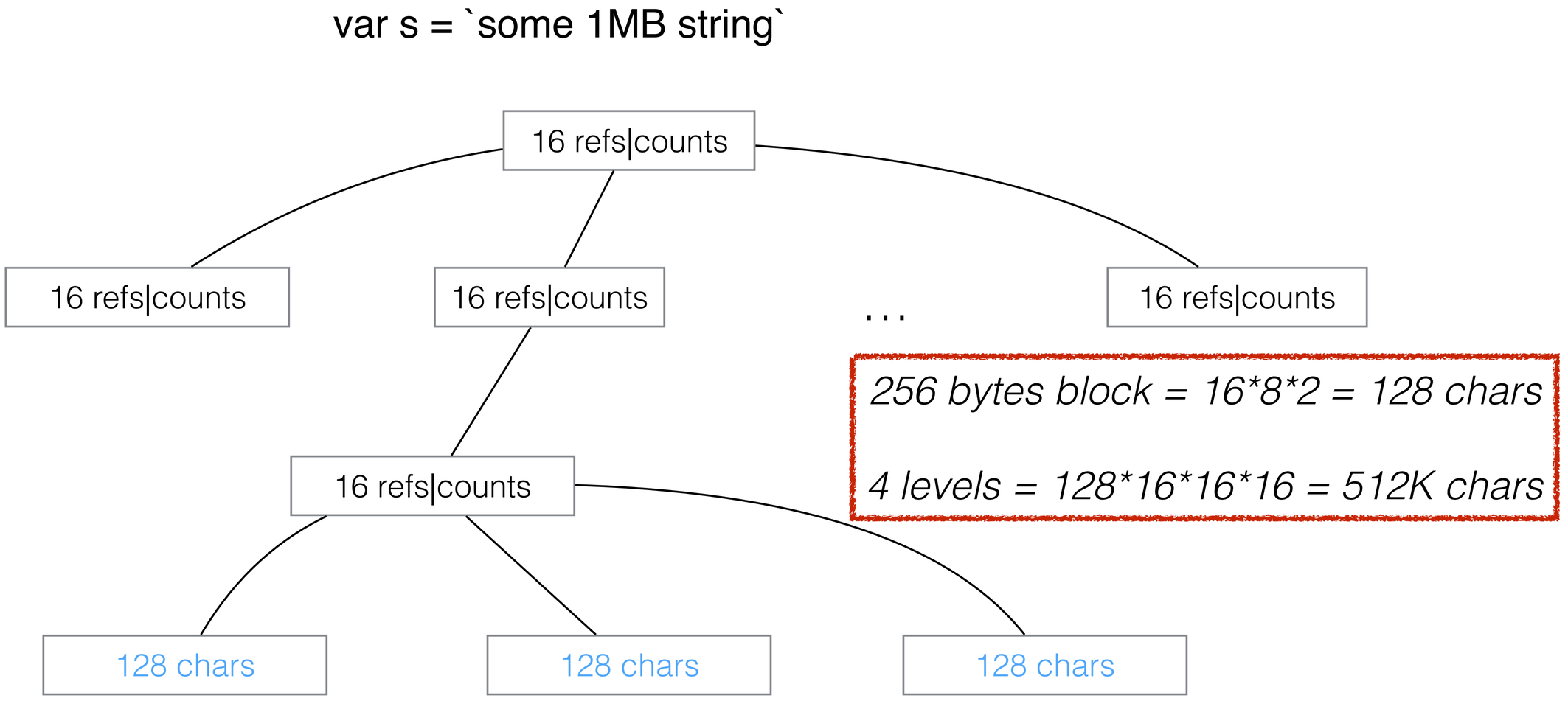
Immutable model



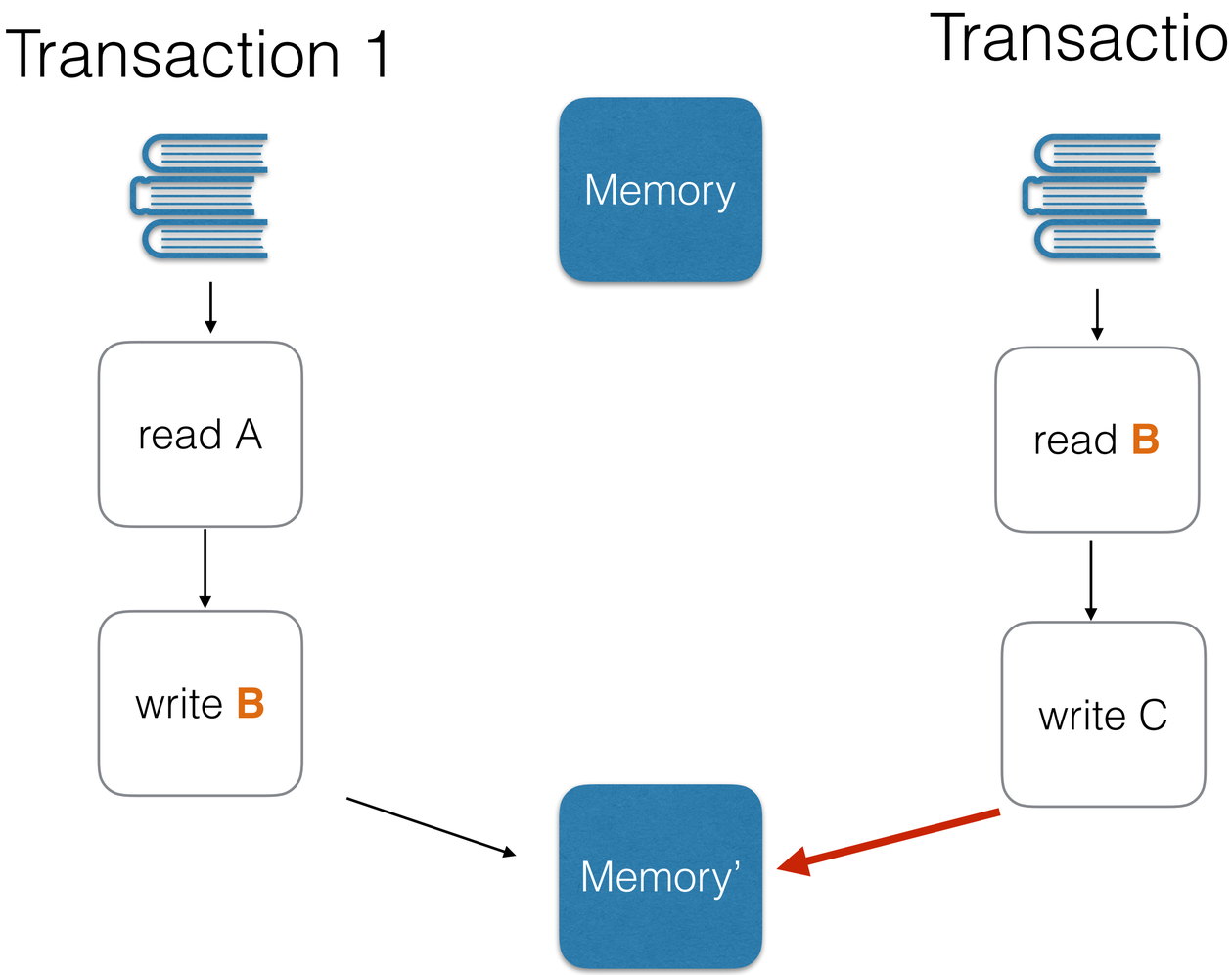
Immutable model



Immutable TextEditor



Software transactional memory



Практика

Задача

Придумать архитектуру поиска использований символа (Find Usages), учитывая нюансы десктопной многопоточности.

Вспомогательные элементы

- Глобальный ReadWriteLock (в компоненте ILocks)
- ILocks.CurrentReadActivityLifetime - лайфтайм, который терминируется, когда UI поток хочет взять WriteLock
- Набор extension методов у лайфтайма
  - lifetime.StartPoolRead(Func<T>)
  - lifetime.RetryWhileOperationCanceled(Func<Task<T>>)

Find Usages on Lifetimes

```
async Task FindUsages(Symbol symbol, ILocks locks) {
    locks.AssertMainThreadReadLock();
    var lf = locks.CurrentReadActivityLifetime();

    DoPrepare();
    var results = await lf.StartPoolRead(() => Find(symbol));
    DoPresent(results);
}
```

```
dialogLifetime.RetryWhileOperationCanceled(symbol, locks)
```

Questions and answers