Параллелизм и транзакции

Денис С. Мигинский

Параллелизм (concurrency)

Параллелизм: способность системы одновременно выполнять несколько вычислений (computations), потенциально (но не обязательно) взаимодействующих друг с другом.

Параллелизм в СП

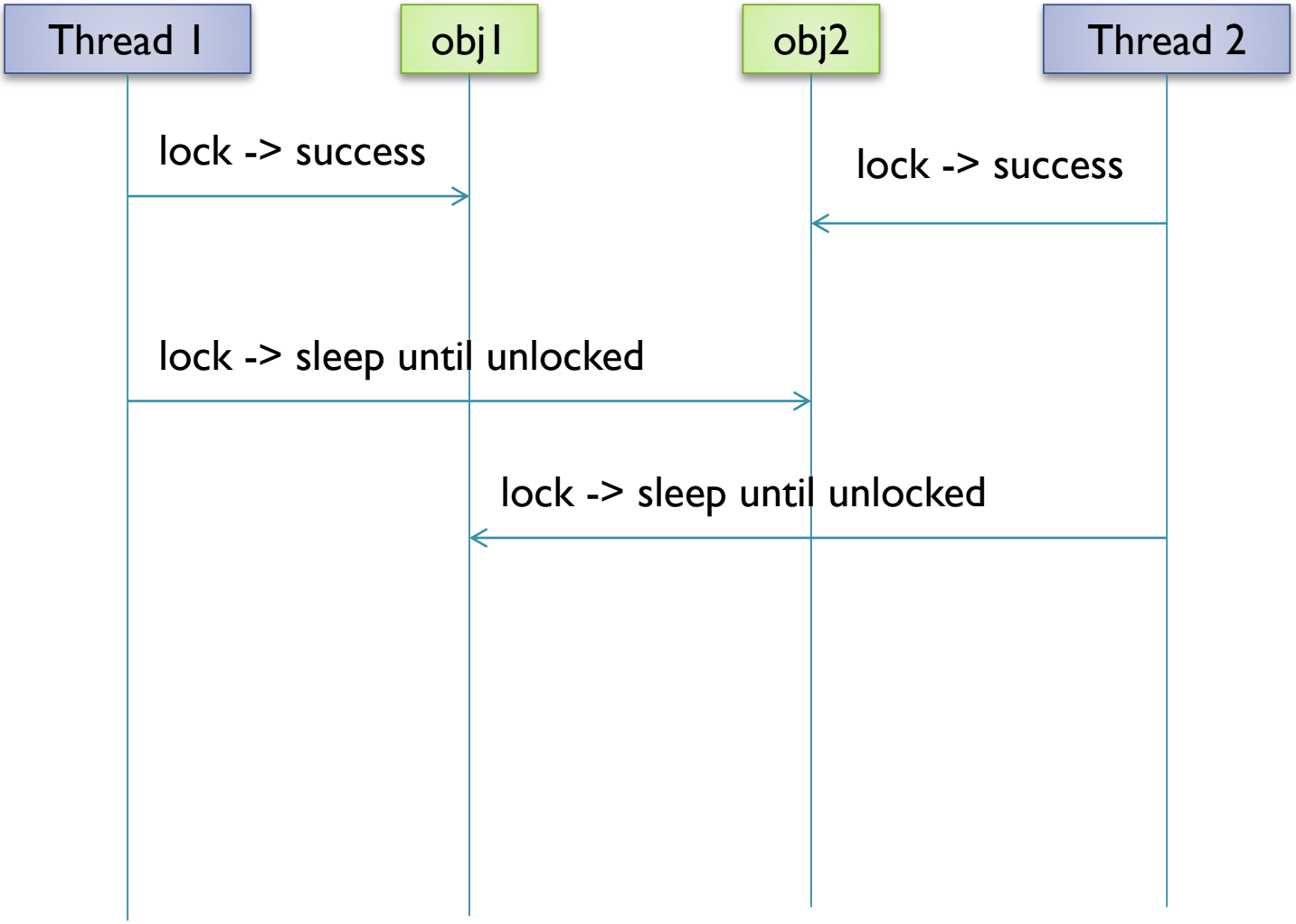
Синхронные примитивы взаимодействия: сообщения, программные каналы, прерывания и т.д.

Асинхронные примитивы взаимодействия: разделяемая память

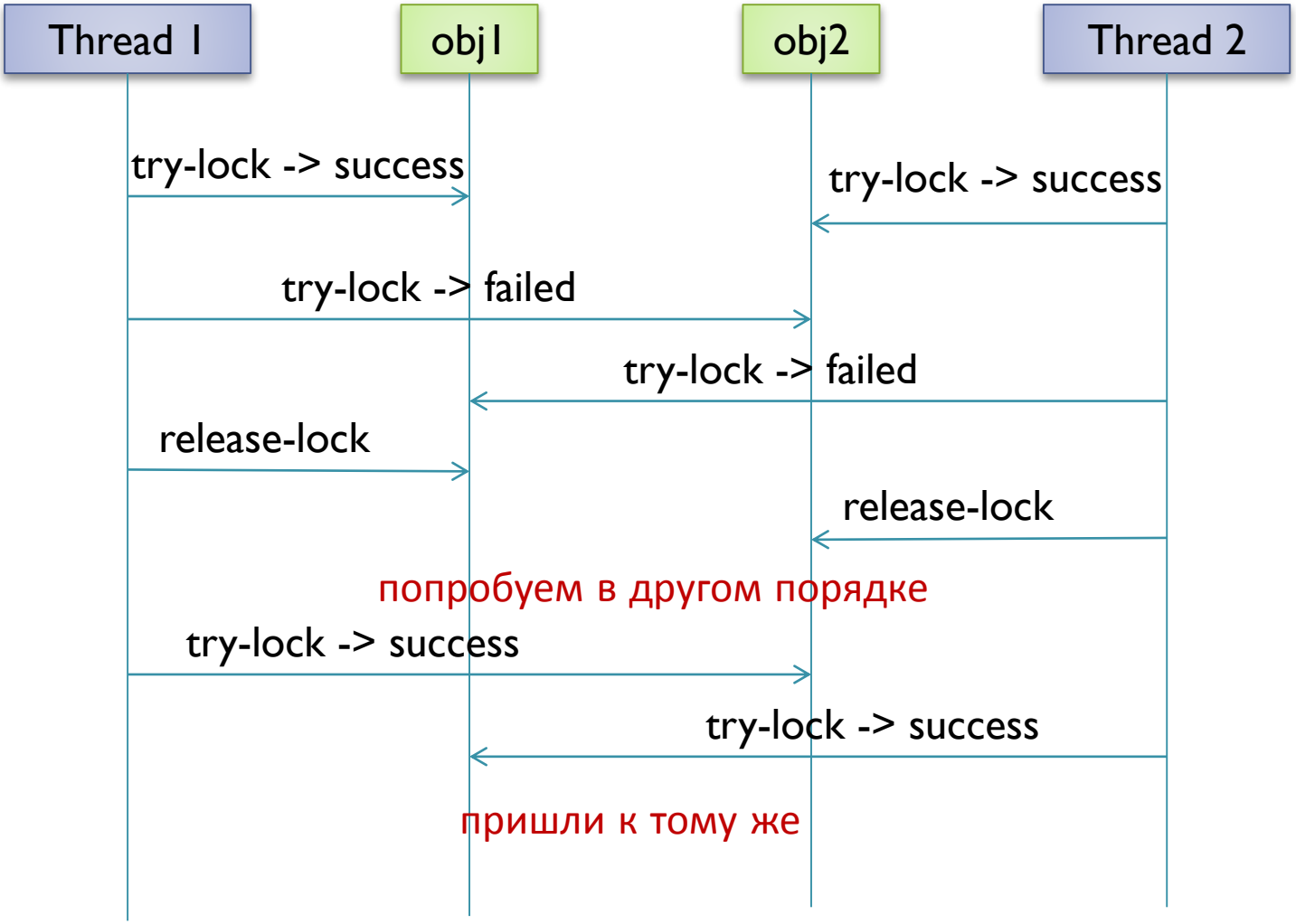
Примитивы синхронизации: семафоры Дейкстры (мьютексы и т.д.), критические секции *(что-то еще?)*

Замечание: любые примитивы синхронизации и взаимодействия обладают побочными эффектами, т.е. операции над ними не являются референциально-прозрачными.

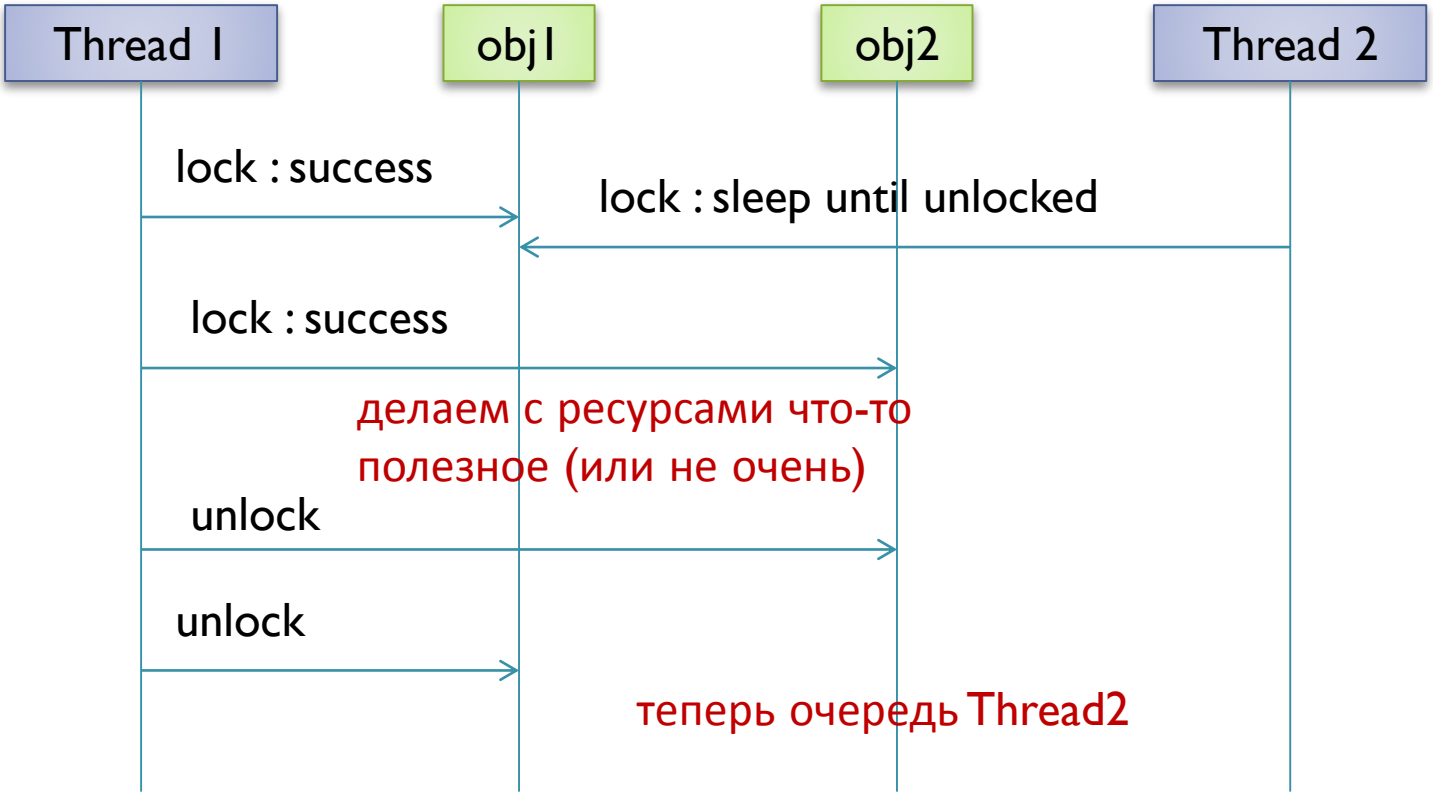
Проблемы синхронизации: dead-lock



Проблемы синхронизации: live-lock



Технически-правильное устранение dead-lock



Проблемы с использованием блокировок

1. Должен быть задан глобальный порядок всех блокируемых ресурсов в системе. Все сущности, участвующие в параллелизме, обязаны про этот порядок знать (нарушение SoC).

2. Нужно знать все ресурсы, которые потенциально будут задействованы в критической секции, даже если эти ресурсы инкапсулированы (следствие предыдущего пункта, также нарушение SoC).

3. Ресурсы блокируются заранее, некоторые могут быть не востребованы в критической секции (потеря эффективности параллелизма).

4. Можно забыть заблокировать или разблокировать какой-то ресурс (сокрытие ошибок)

Вопросы

- Может ли быть параллелизм вообще без блокировок? В каких случаях это возможно?
- Если нельзя избавиться от блокировок, можно ли их изолировать (в смысле SoC)?
- Не возникнет ли новых проблем?

Параллелизм в ФП

Концепция (split-)map-reduce:

1. **split** – разбиваем данные на независимые блоки, которые могут быть обработаны независимо.
2. **map** – преобразуем каждый блок независимо, в т.ч. в параллельных процессах, в т.ч. процессы могут быть на разных узлах (серверах и т.д.)
3. **reduce** – собираем результат из обработанных блоков

Замечания:

1. Концепцию можно применять итеративно, каскадно, и т.д.
2. В некоторых случаях та же концепция может быть применена внутри **reduce**
3. Эффективность распараллеливания зависит от затрат на **split** и **reduce**

Основное преимущество: нет проблем с синхронизацией

Atomic-типы

Свойства atomic-типов:

- ведут себя как обычные типы данных
- обеспечивают потоково-безопасный доступ для операций
- определяют специфичные для данного типа транзакции

Пример: **java.util.concurrent.atomic.AtomicInteger**

Некоторые транзакции:

```
int incrementAndGet()
int getAndIncrement ()
boolean compareAndSet(int expect, int update)
int getAndSet(int newValue)
```

Ссылка

Основные операции над ссылкой:

- конструирование
- разыменование (получение содержимого)
- изменение ссылки (замена объекта, на который ссылка указывает)

Ruby: любой именованный объект является изменяемой ссылкой

Clojure: именованные объекты являются неизменяемыми ссылками (т.е. полноценными ссылками не являются). Для изменяемых ссылок существуют специальные примитивы.

Типичные сценарии использования ссылки

Ссылки в СП (классическом ООП)

- получить объект
- изменить объект (применить к нему метод или набор методов)

Ссылки, как расширение ФП

- получить объект
- создать новый объект на основе старого
- заменить содержимое ссылки

Проблема конкурентного доступа: потерянное обновление

Два потока пытаются одновременно работать с одной ссылкой

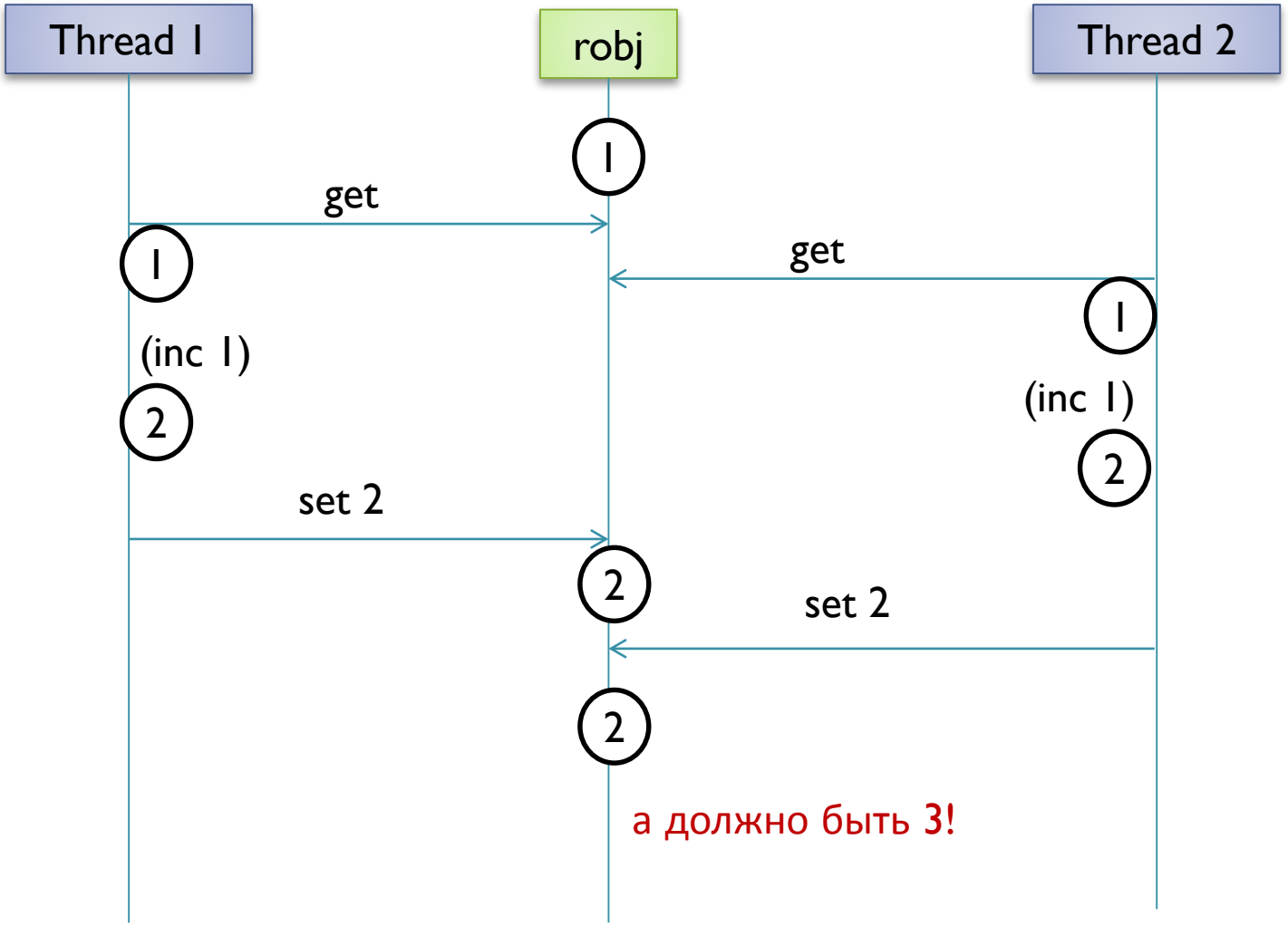
Структурный сценарий

Порядок вызова методов из разных потоков не определен. Результирующее состояние объекта непредсказуемо, в т.ч. не гарантирована целостность объекта.

Функциональный сценарий

Порядок разыменования и присваивания не определен. В результате ссылка будет содержать целостный объект, но некоторые результаты преобразований могут быть забыты.

Иллюстрация проблемы



Atomic-ссылка в Java

Класс: **java.util.concurrent.atomic.AtomicReference<V>**

Операции со ссылкой:

```
V get()
void set(V newValue)
```

Транзакции:

```
V getAndSet(V newValue)

//сравнивает на == и присваивает новое значение
boolean compareAndSet(V expect, V update)

//в отличие от предыдущего метода
//присваивает только при успешном сравнении
boolean weakCompareAndSet(V expect, V update)
```

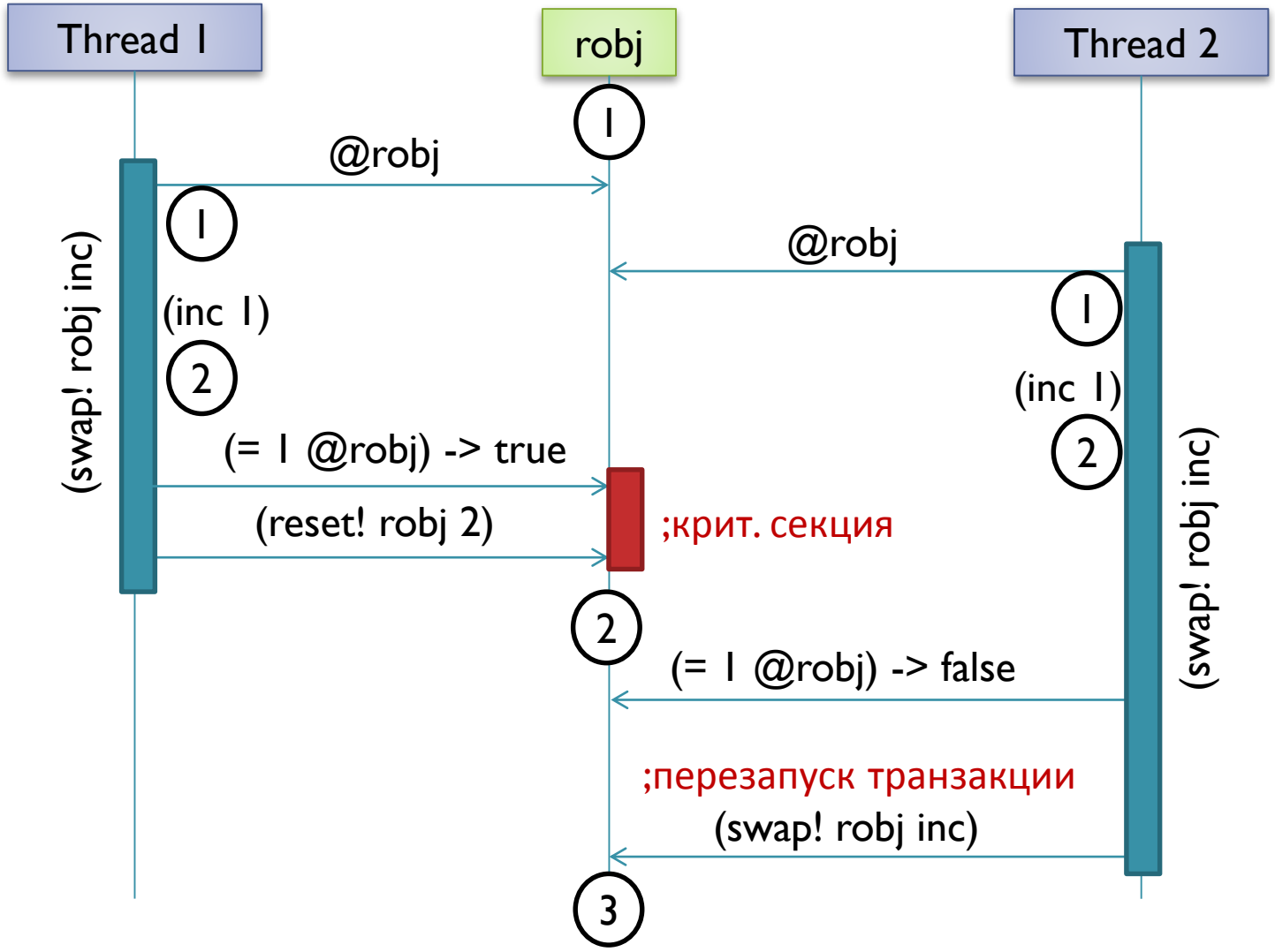
Atomic-ссылка в Clojure

```
;конструирование ссылки
(def mutable-obj (atom 1))

;разыменование ссылки
(println (deref mutable-obj))
;присваивание нового значения
(reset! mutable-obj 2)
;эквивалентно deref
(println @mutable-obj)

;применение функции к ссылке с
;присваиванием результата (транзакция)
(swap! mutable-obj inc)
(println @mutable-obj)
>> 1
   2
   3
```

Решение проблемы потерянного обновления в Clojure

```
(def robj (atom 1))
;java.lang.Thread + syntactic sugar
(let [t1 (new Thread
           (fn [] (swap! robj inc)))
      t2 (new Thread
           (fn [] (swap! robj inc)))]
  (.start t1)
  (.start t2)
  (.join t1)
  (.join t2))

(println @robj)
>> 3
```

Транзакция swap!



Транзакции и критические секции

Транзакция – составная операция, которая обладает свойством атомарности, т.е. должна быть либо выполнена полностью, либо не выполнена вообще.

Типичный примитив для синхронизации потоков:
критическая секция – часть программы, в которой происходит обращение к разделяемым данным, при этом два процесса/потока не могут одновременно находиться в одной критической секции. Может рассматриваться как простейший вид транзакции.

Типичная реализация критической секции – через семафоры/мьютексы.

Крайние случаи:

- один ресурс – один семафор (получаем букет проблем)
- все ресурсы – один семафор (теряем в эффективности)

ACID

ACID – стандартные требования к транзакционной системе (СУБД, транзакционной файловой системе и т.д.):

Atomicity (атомарность) – транзакция должна быть выполнена либо полностью либо не выполнена вообще. В том числе атомарность должна гарантироваться в случае ошибок.

Consistency (целостность) – транзакция переводит систему из одного корректного (valid) состояния в другое. Должна гарантироваться не только техническая целостность (ссылочная и т.д.) структур данных, но и содержательная целостность, включая выполнение всех ограничений (constraints), накладываемых постановкой задачи/предметной областью.

ACID (продолжение)

ACID – стандартные требования к транзакционной системе (СУБД, транзакционной файловой системе и т.д.):

Isolation (изолированность) – параллельное исполнение транзакций должно приводить к тому же результату, что и последовательное (в каком-либо порядке). При это сами транзакции об этом заботиться не должны

Durability (устойчивость) – если транзакция завершена успешно, то внесенные ею изменения должны сохраниться даже в случае потери питания и других аппаратных или программных сбоев (после восстановления работоспособности системы)

Для транзакций с объектами в ОЗУ неприменимо

Операции с транзакциями

Run – запуск транзакции (как и любого другого куска кода)

Commit – подтверждение (и, обычно, публикация) результатов транзакции.

Rollback – откат всех изменений, выполненных транзакцией (в т.ч. если транзакция выполнена не полностью)

Retry – рестарт транзакции (эквивалентно rollback + run)

Отличие полноценной транзакции от критической секции

Критическая секция	Транзакция
Явная блокировка ресурсов (нарушает SoC)	Неявная блокировка, либо отсутствие таковой (соответствует SoC)
Не ограничивает побочные эффекты	Побочные эффекты ограничены рамками ресурсов, управляемых данной транзакцией, другие побочные эффекты запрещены. До некоторой степени может рассматриваться как референциально-прозрачный примитив.
Поддерживаемые операции: run, try-run	Поддерживаемые операции: явно: run явно или неявно: commit, rollback, retry

Проблемы, решаемые транзакциями

По возрастанию сложности решения:

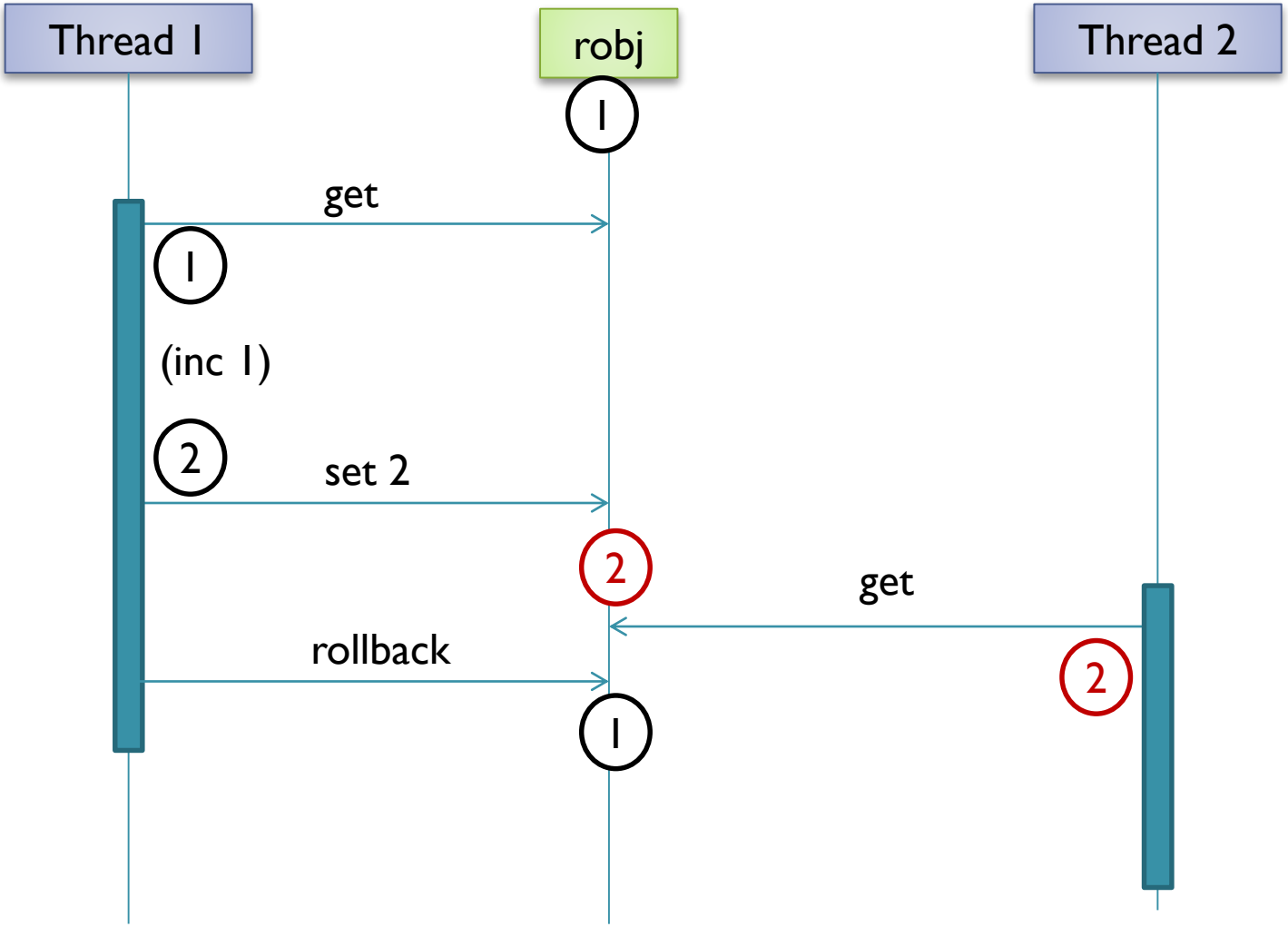
1. Потерянное обновление (lost update) - была рассмотрена выше, решается даже критической секцией (в СУДБ уровень изоляции транзакций read uncommited)

2. Черновое чтение (dirty read)

3. Невоспроизводимое чтение (not-repeatable read)

4. Фантомное чтение (phantom read) – в основном актуальная для БД, может рассматриваться как вариант 3)

Черновое чтение: проблема



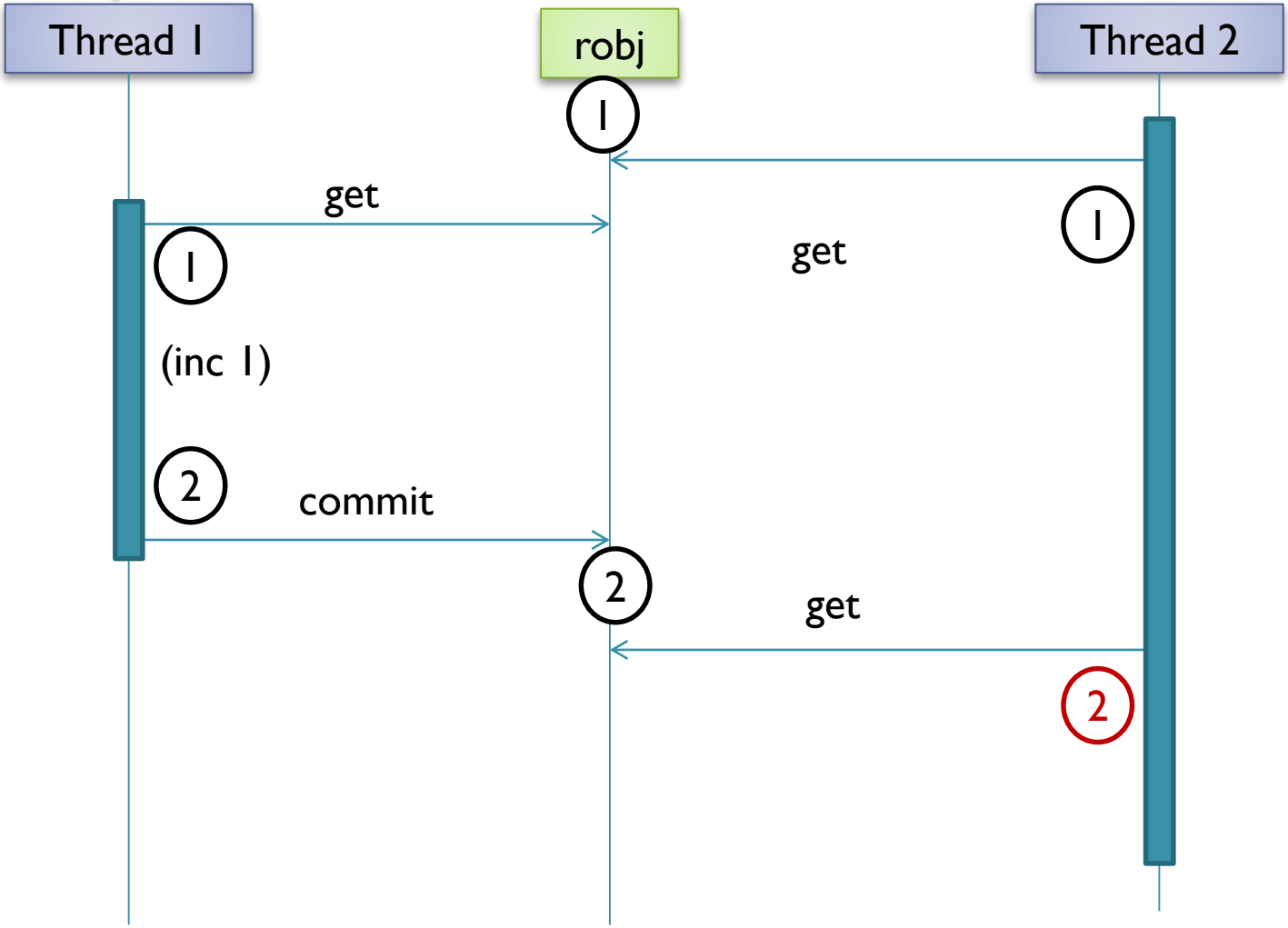
Черновое чтение: решение

Изменения должны быть доступны другим транзакциям только после успешного завершения данной (commit)

Простейшая реализация: блокировка объекта перед записью и до конца транзакции.

В СУБД: уровень изоляции транзакций read commited

Невоспроизводимое чтение: проблема



Невоспроизводимое чтение: решение

Нужно обеспечить воспроизводимость чтения в рамках одной транзакции

Простейшая реализация: блокировка объекта перед чтением и до конца транзакции

Более умная реализация: временная мемоизация get

В СУБД: уровень изоляции транзакций repeatable read

Анализ проблем

Для одного ресурса:

Блокировка решает перечисленные проблемы, но потенциально снижает уровень конкурентности

atom (Clojure) решает перечисленные проблемы без использования блокировок

Вопрос – каким образом это может быть?

Для нескольких ресурсов:

Блокировка решает проблемы, но приходится бороться с **dead-lock**, что приводит к нарушению **SoC**

atom (Clojure) проблемы не решает

Можно ли обобщить механизм atom'а одновременно на несколько ресурсов?