

Многопоточное Программирование: Алгоритмы без блокировок: Консенсус

Роман Елизаров, JetBrains, elizarov@gmail.com

Никита Коваль, JetBrains, ndkoval@ya.ru

ИТМО 2020

Какие еще задачи мы можем
решить с помощью регистров?

Консенсус



Задача о консенсусе

```
class Consensus:  
    def decide(val):  
        ...  
        return decision
```

- ✓ Каждый поток использует объект Consensus **один раз**

Задача о консенсусе: Попытка 1

```
class Consensus:  
  def decide(val):  
    return val
```

Задача о консенсусе: Попытка 1

```
class Consensus:  
    def decide(val):  
        return val
```

- **Согласованность**
(consistent): все потоки
должны вернуть одно и то же
значение из метода decide

Задача о консенсусе: Попытка 2

```
class Consensus:  
    def decide(val):  
        return 0
```

- **Согласованность**
(consistent): все потоки
должны вернуть одно и то же
значение из метода decide

Задача о консенсусе: Попытка 2

```
class Consensus:  
    def decide(val):  
        return 0
```

- **Согласованность** (consistent): все потоки должны вернуть одно и то же значение из метода decide
- **Обоснованность** (valid): возвращенное значение было входным значением какого-то из потоков

Задача о консенсусе: Попытка 3 (с блокировкой)

```
shared int decision // init NA
Mutex mutex
```

```
def decide(val):
    mutex.lock()
    if decision == NA:
        decision = val
    mutex.unlock()
    return decision
```

- **Согласованность** (consistent): все потоки должны вернуть одно и то же значение из метода decide
- **Обоснованность** (valid): возвращенное значение было входным значением какого-то из потоков

Тривиальная реализация протокола консенсуса с помощью взаимного исключения для любого количества потоков

Задача о консенсусе: Попытка 3 (с блокировкой)

```
shared int decision // init NA
Mutex mutex
```

```
def decide(val):
    mutex.lock()
    if decision == NA:
        decision = val
    mutex.unlock()
    return decision
```

- **Согласованность** (consistent): все потоки должны вернуть одно и то же значение из метода decide
- **Обоснованность** (valid): возвращенное значение было входным значением какого-то из потоков
- **Без ожидания** (wait-free)

Безусловный прогресс

Задача о консенсусе: все требования

```
class Consensus:  
    def decide(val):  
        ...  
        return decision
```

✓ Каждый поток использует
объект Consensus **один раз**

- **Согласованность** (consistent): все потоки должны вернуть одно и то же значение из метода decide
- **Обоснованность** (valid): возвращенное значение было входным значением какого-то из потоков
- **Без ожидания** (wait-free)

Как же реализовать такой протокол консенсуса
используя регистры?

Консенсусное число

- Если с помощью класса [атомарных] объектов C и атомарных регистров можно реализовать консенсусный протокол **без ожидания** (wait-free) с помощью **детерминированного алгоритма** для N потоков (и не больше), то говорят что у класса C **консенсусное число** равно N .

Консенсусное число

- Если с помощью класса [атомарных] объектов C и атомарных регистров можно реализовать консенсусный протокол **без ожидания** (wait-free) с помощью **детерминированного алгоритма** для N потоков (и не больше), то говорят что у класса C **консенсусное число** равно N .
- **ТЕОРЕМА:** Атомарные регистры имеют консенсусное число 1.
 - Т.е. с помощью атомарных регистров даже 2 потока не могут прийти к консенсусу без ожидания (докажем от противного) даже для 2-х возможных значений

Модель

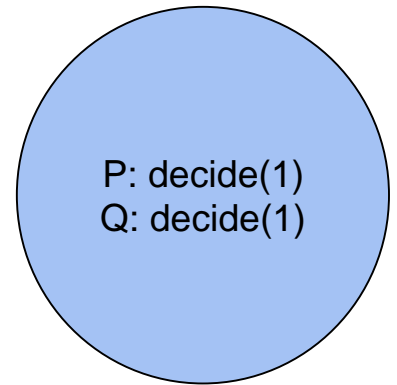
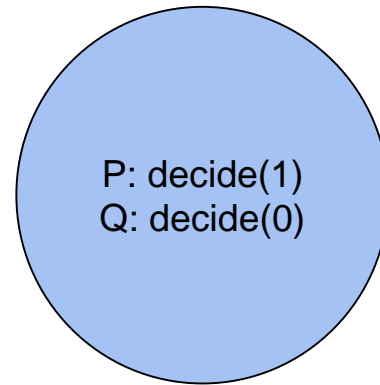
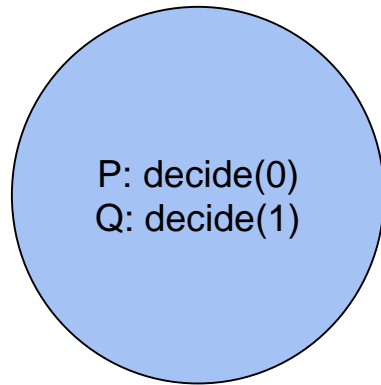
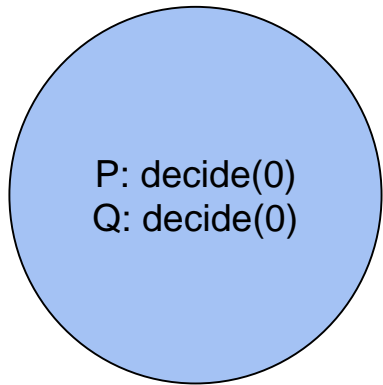
- Исходные объекты **атомарны**. Значит любое исполнения можно рассматривать как последовательное в каком-то порядке

Модель

- Исходные объекты **атомарны**. Значит любое исполнения можно рассматривать как последовательное в каком-то порядке
- Доказываем **от противного**
 - Предполагает что **алгоритм есть**, анализируем его исполнение, находим проблему

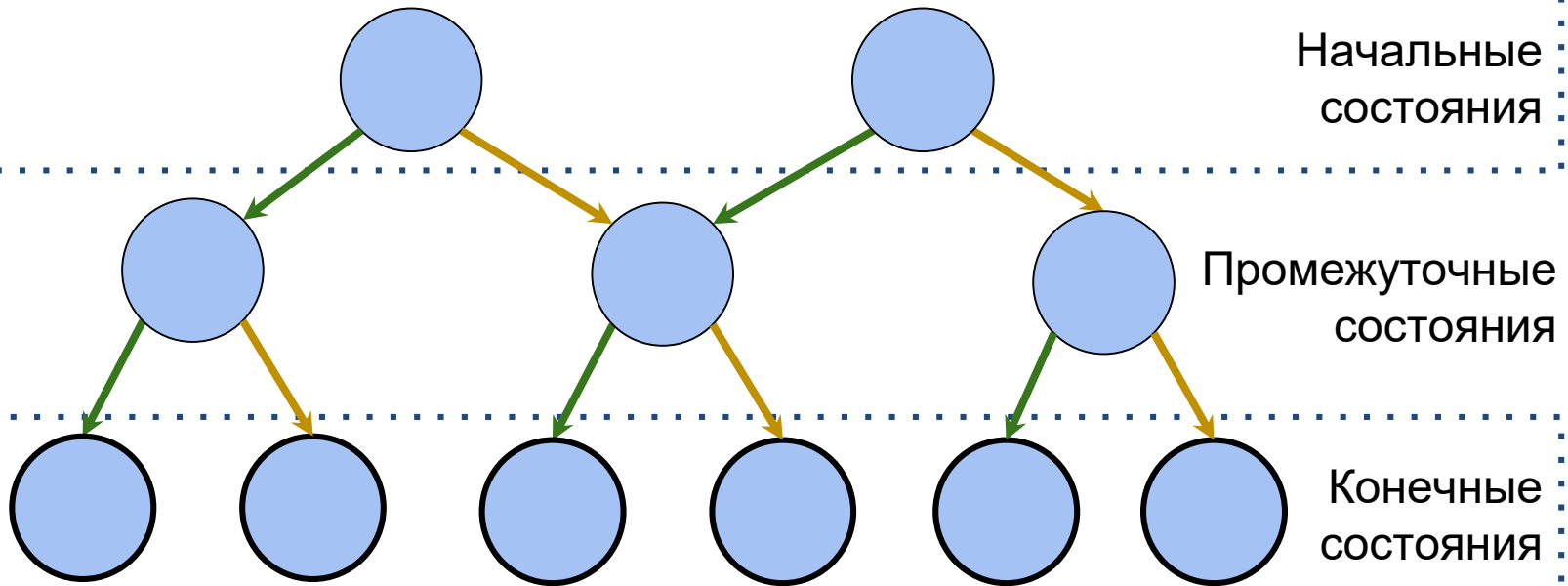
Модель: начальные состояния

- Два потока решают задачу **бинарного** консенсуса



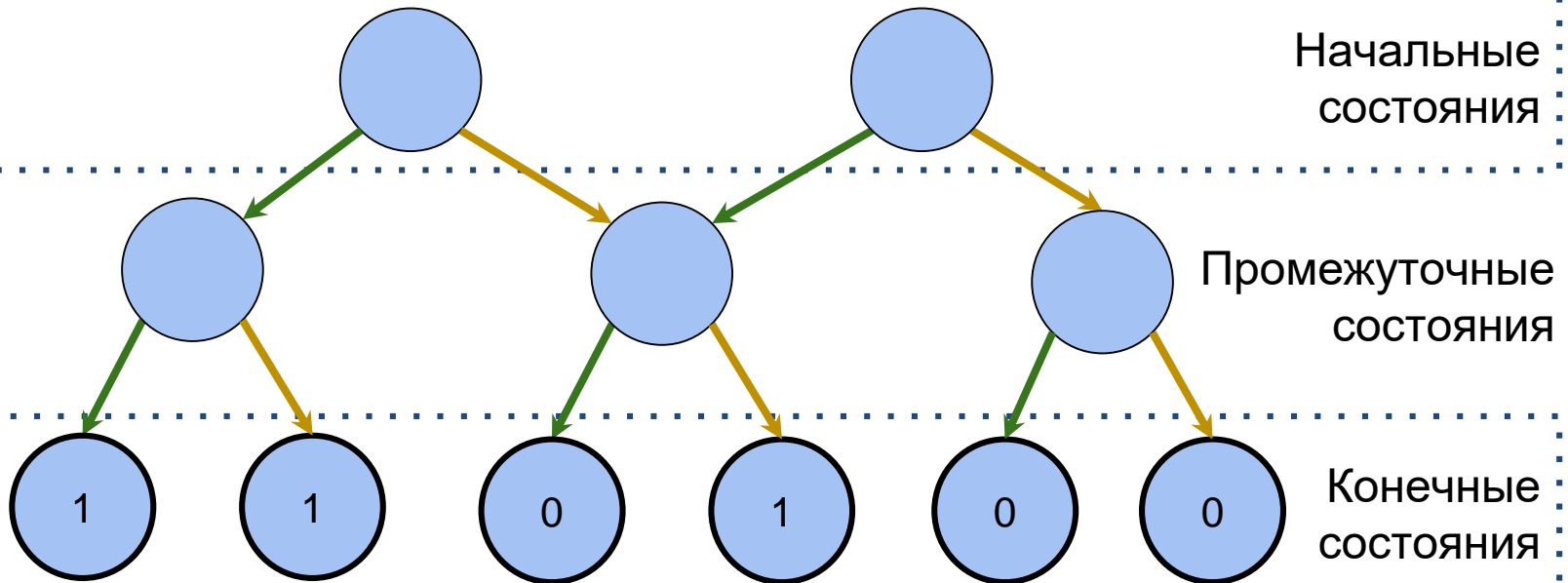
Модель: DAG состояний

- Рассматриваем граф состояний; он **конечный** и **без циклов**, так как алгоритм работает **без ожидания** (wait-free)



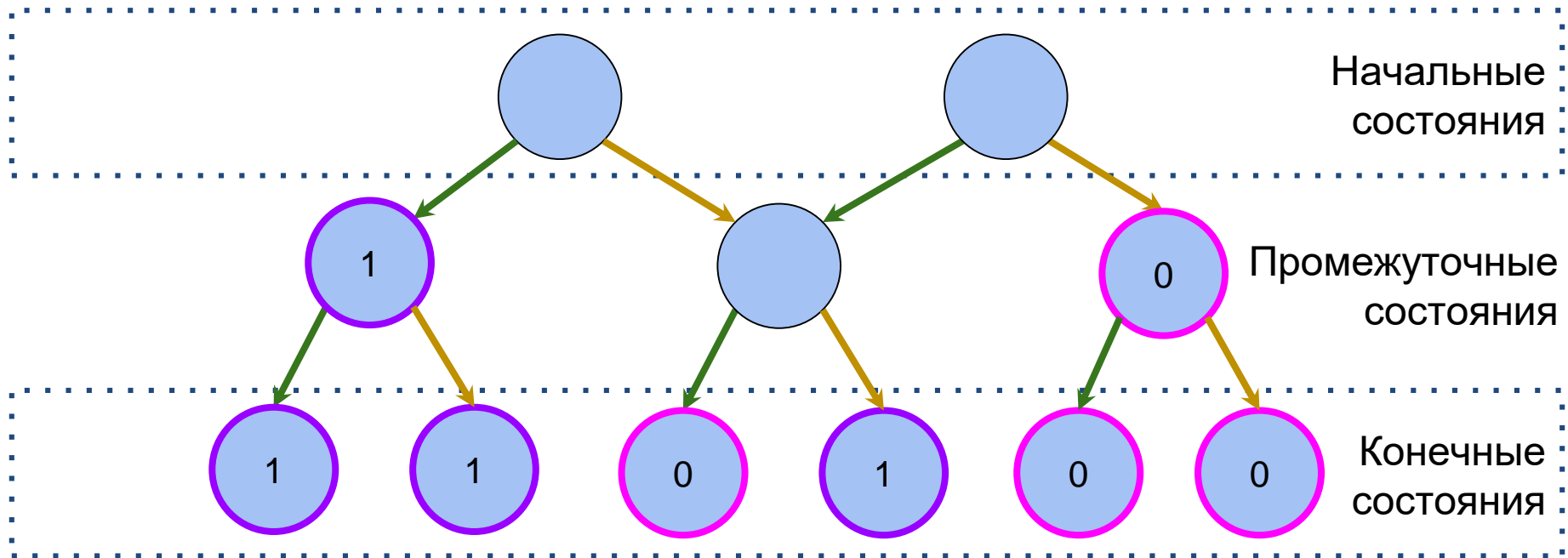
Модель: листья

- Листья – конечные состояния помеченные 0 или 1 (в зависимости от значения консенсуса) так как есть **согласованность**



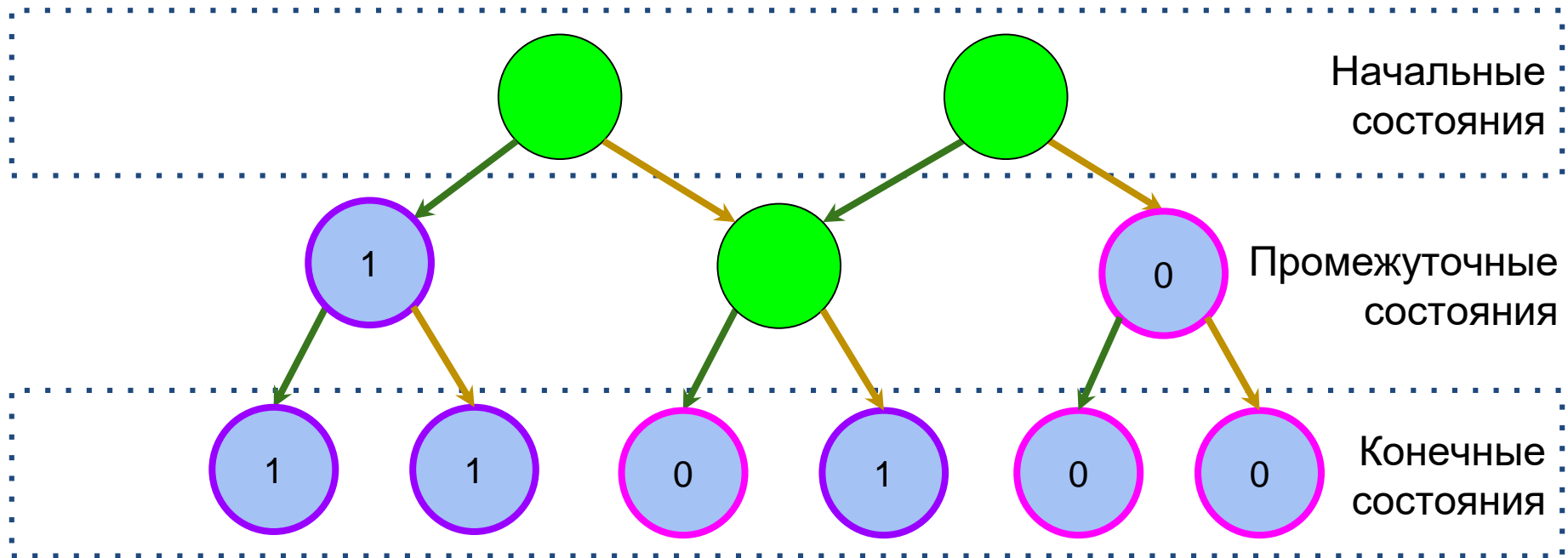
Модель: валентность состояния

- **Определение: x -валентное состояние системы ($x = 0,1$)** – консенсус во всех нижестоящих листьях будет x .



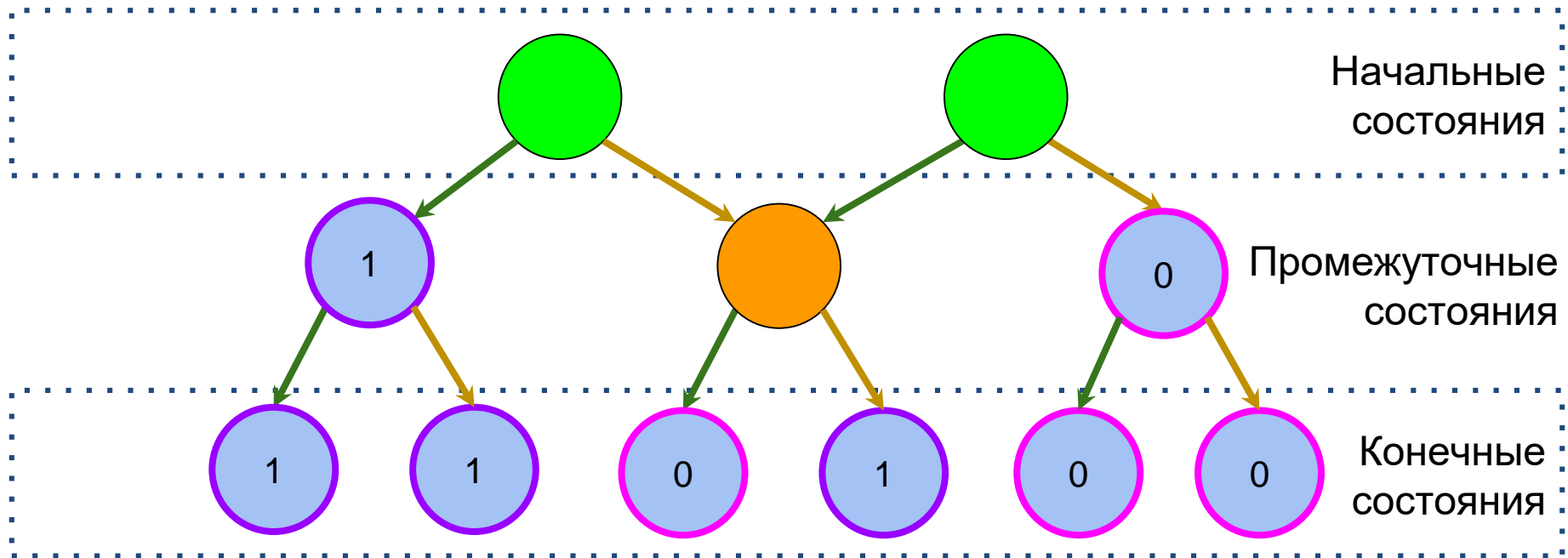
Модель: бивалентность

- **Определение: бивалентное состояние** – возможен консенсус как 0 так и 1.



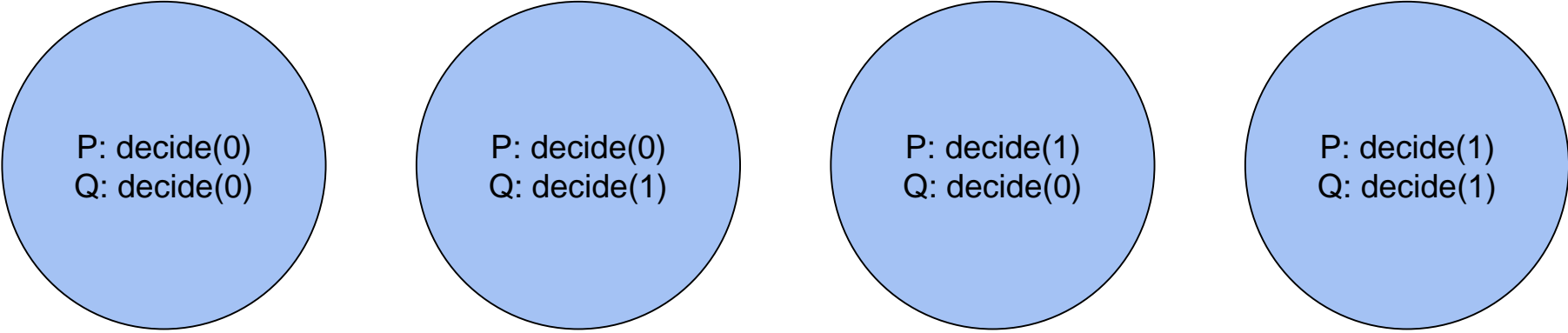
Модель: бивалентность

- **Определение: критическое состояние** – такое бивалентное состояние, все дети которого одновалентны



Лемма о начальном состоянии

- **ЛЕММА 1:** Существует начальное бивалентное состояние.



P: decide(0)
Q: decide(0)

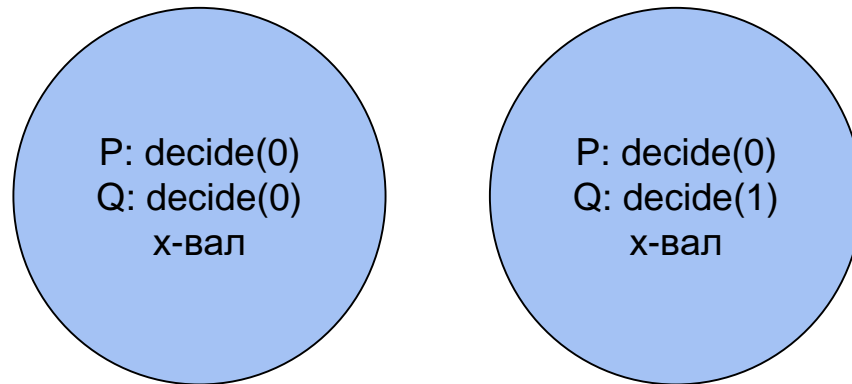
P: decide(0)
Q: decide(1)

P: decide(1)
Q: decide(0)

P: decide(1)
Q: decide(1)

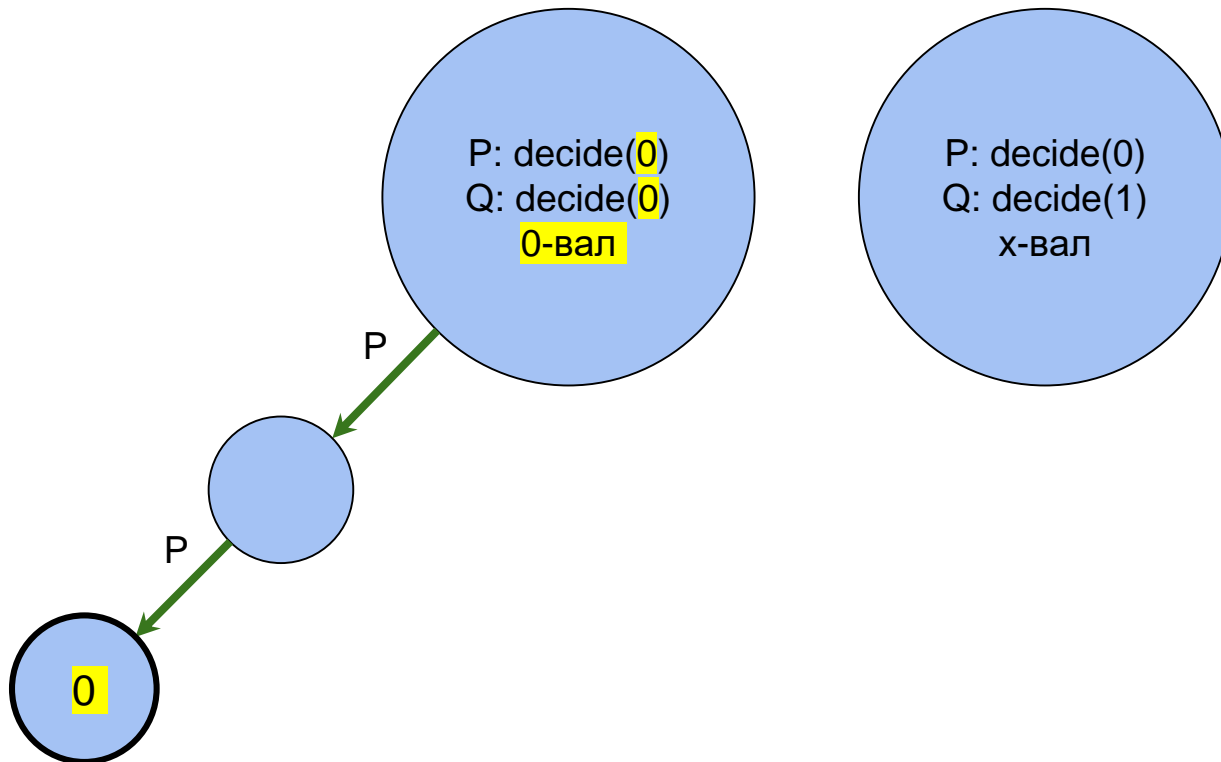
Лемма о начальном состоянии

- **ЛЕММА 1:** Существует начальное бивалентное состояние.



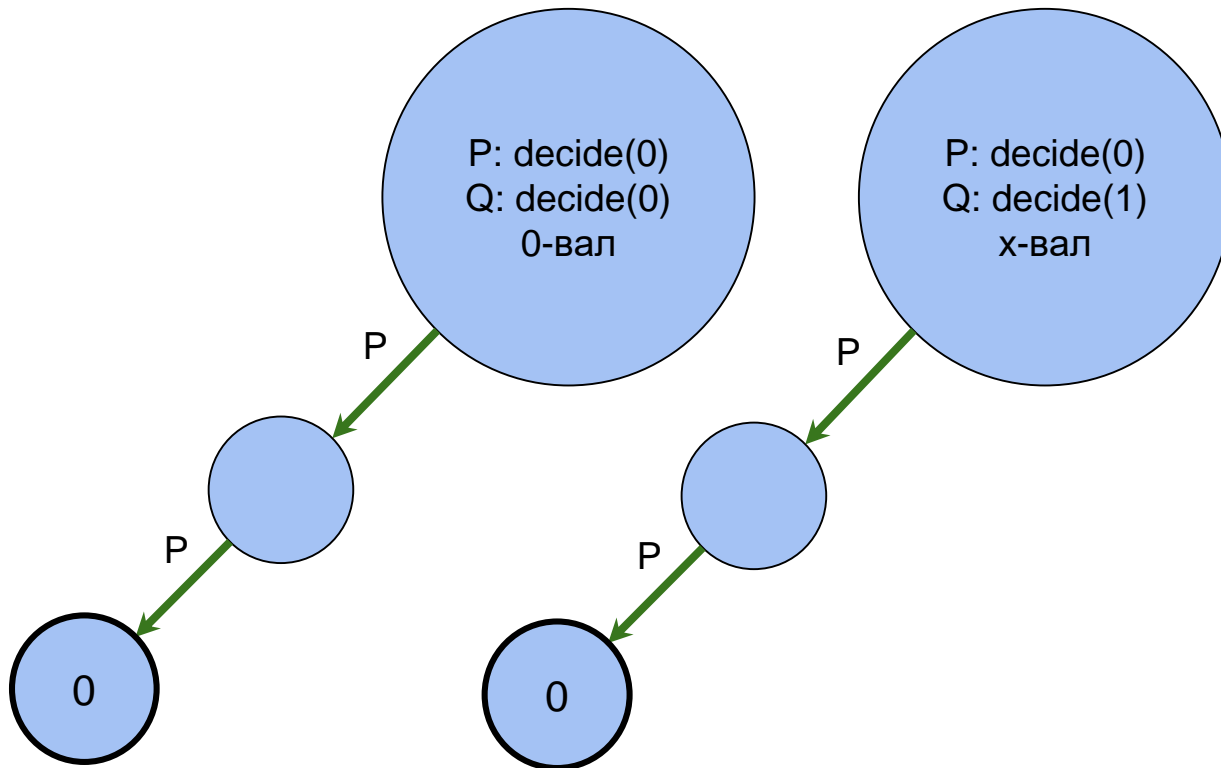
Лемма о начальном состоянии

- **ЛЕММА 1:** Существует начальное бивалентное состояние.
 - Используем **обоснованность**



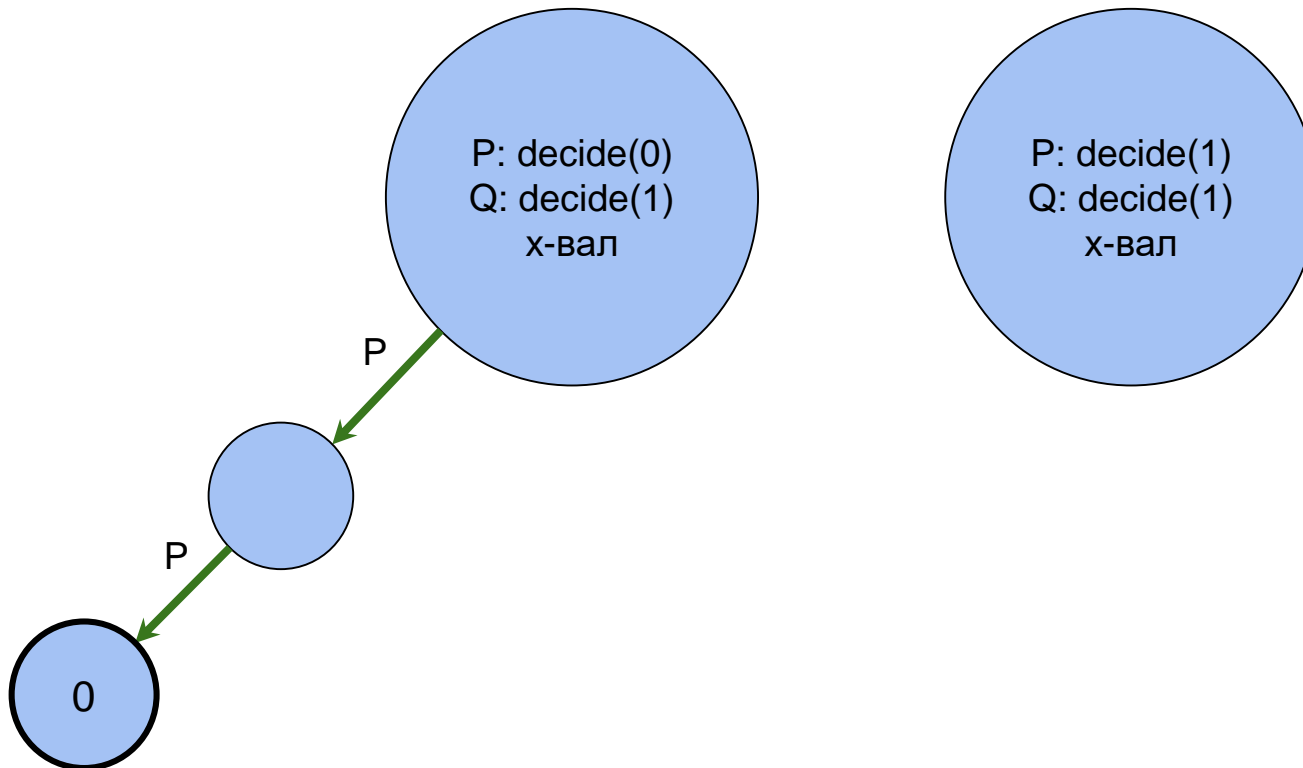
Лемма о начальном состоянии

- **ЛЕММА 1:** Существует начальное бивалентное состояние.
 - Используем **детерминированность**



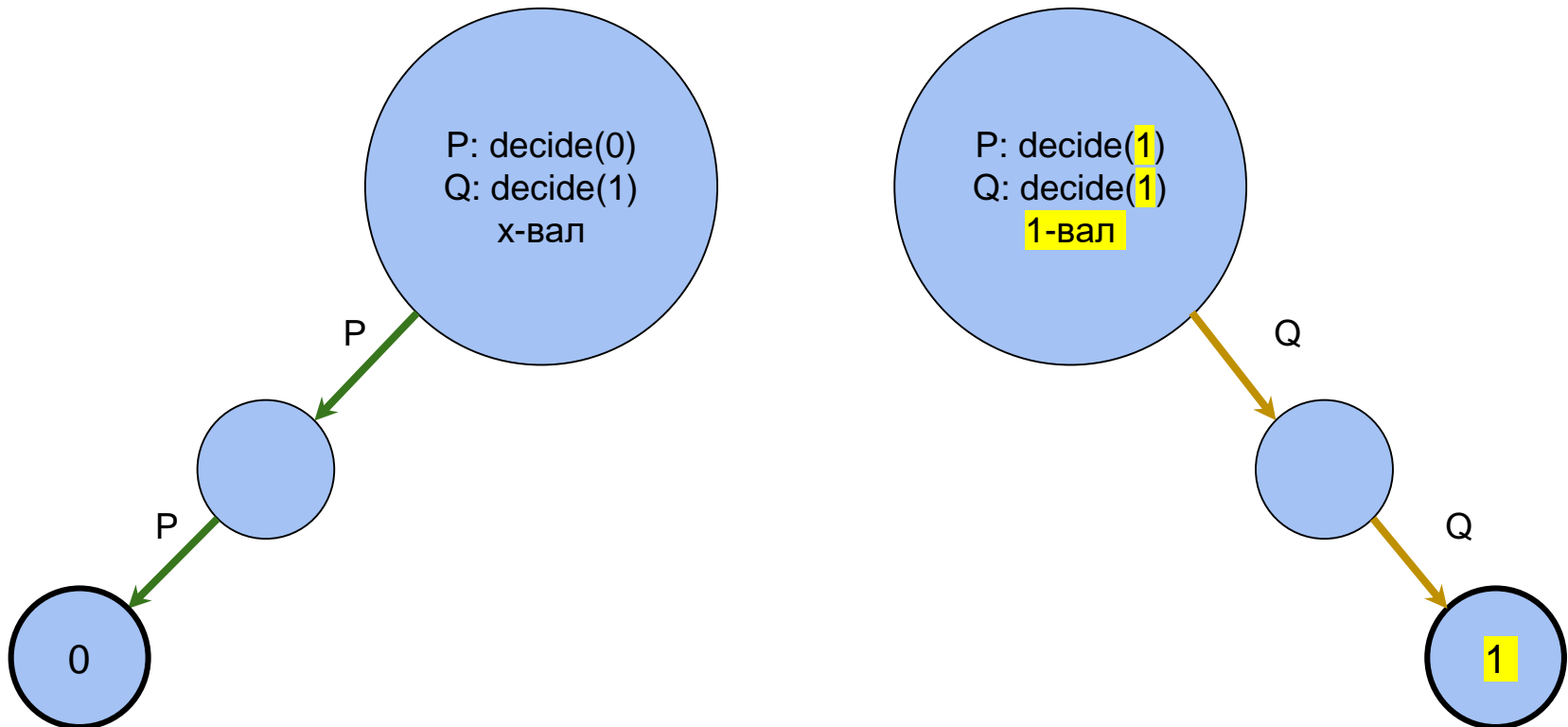
Лемма о начальном состоянии

- ЛЕММА 1:** Существует начальное бивалентное состояние.



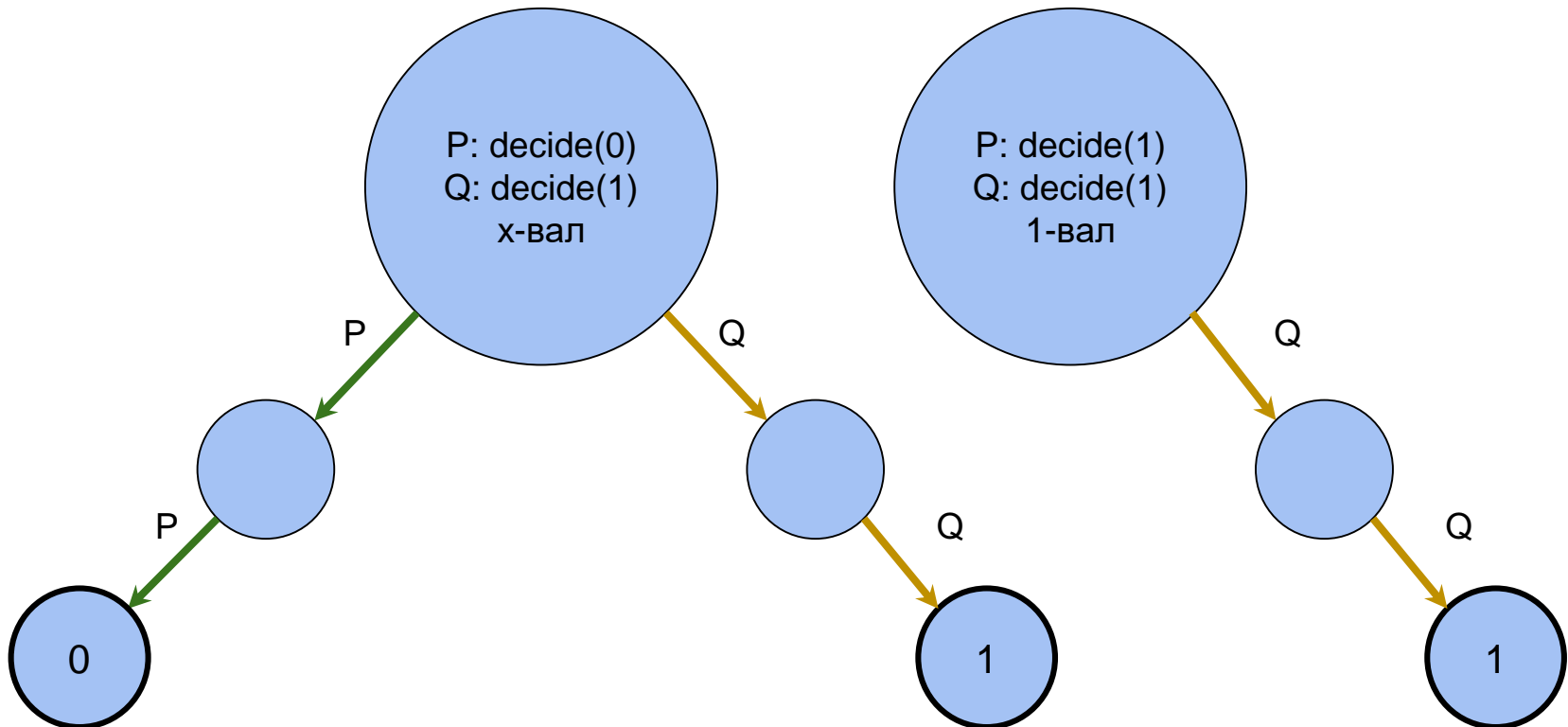
Лемма о начальном состоянии

- ЛЕММА 1:** Существует начальное бивалентное состояние.



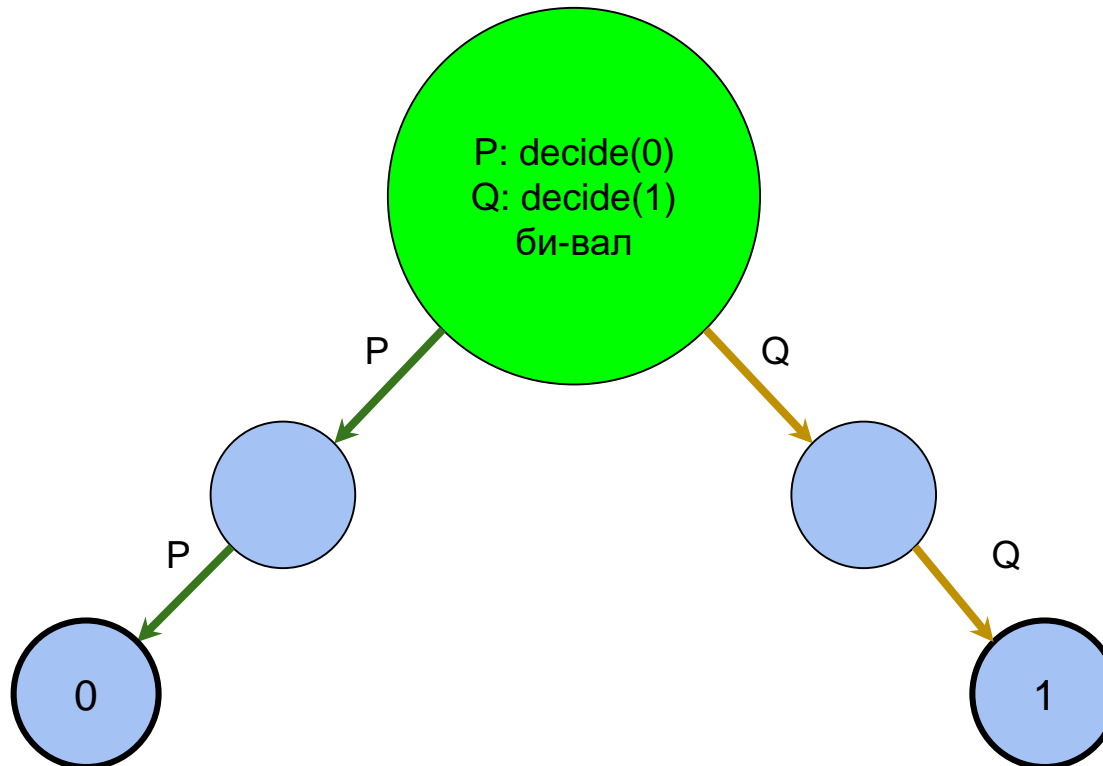
Лемма о начальном состоянии

- ЛЕММА 1:** Существует начальное бивалентное состояние.



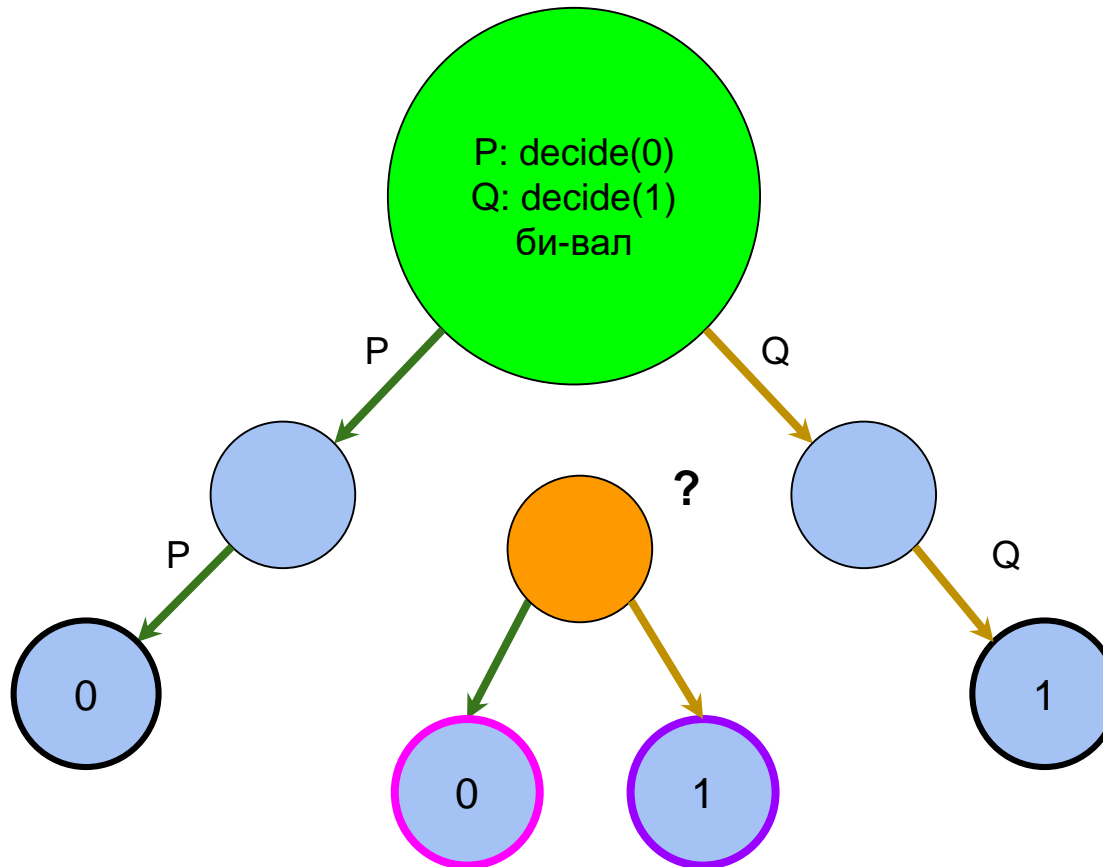
Лемма о начальном состоянии

- **ЛЕММА 1:** Существует начальное бивалентное состояние.



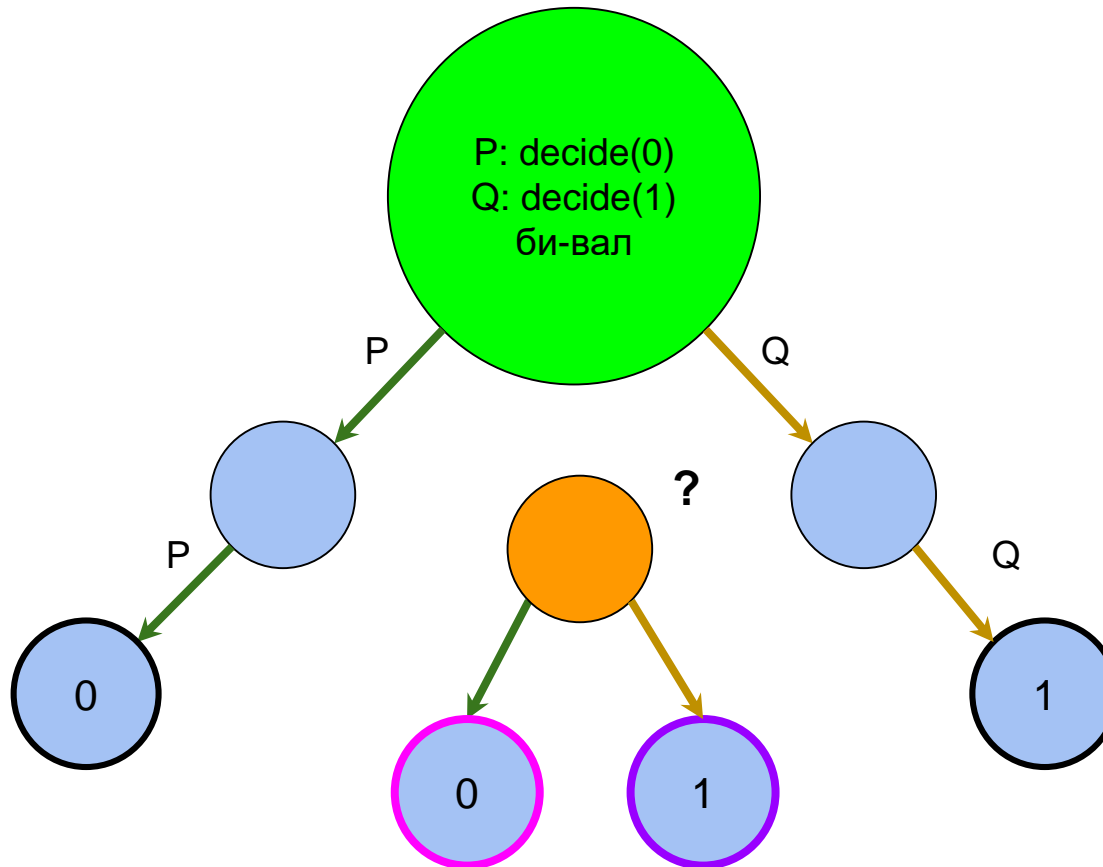
Лемма о критическом состоянии

- **ЛЕММА 2:** Существует критическое состояние.



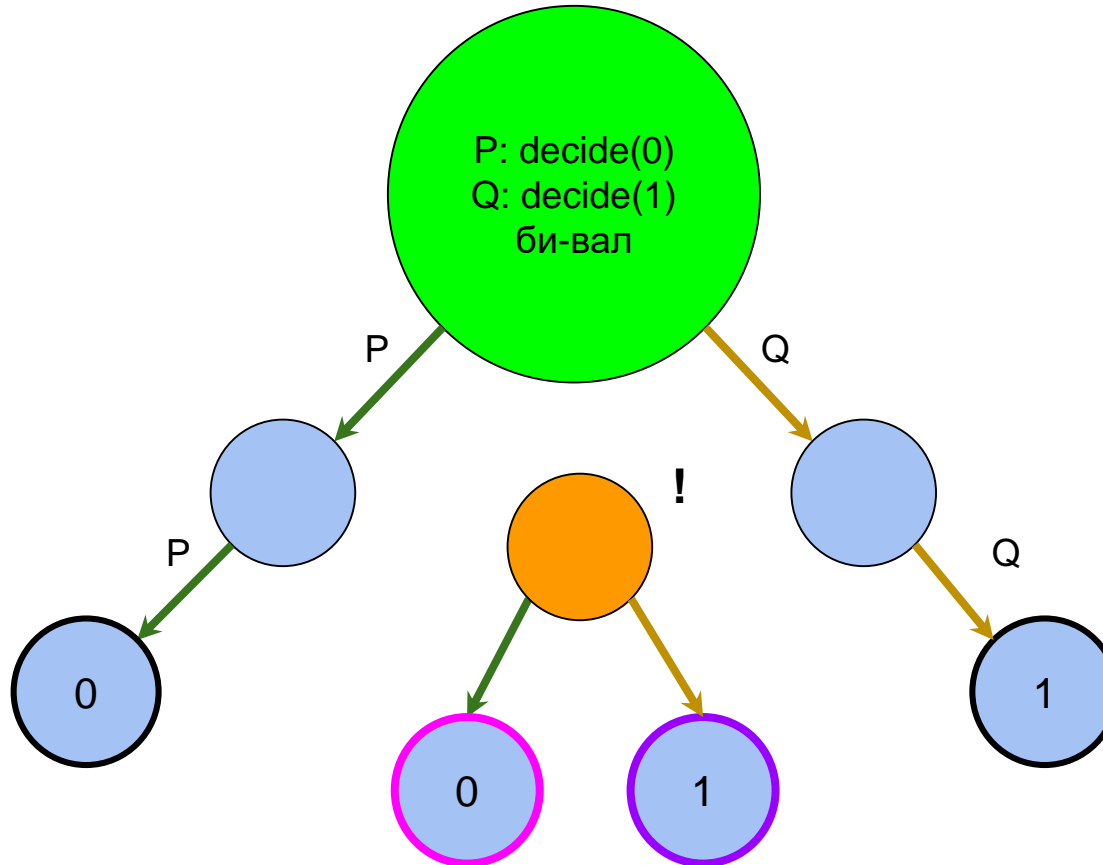
Лемма о критическом состоянии

- **ЛЕММА 2:** Существует критическое состояние.
 - **От противного:** Предположим что нет



Лемма о критическом состоянии

- **ЛЕММА 2:** Существует критическое состояние.
 - **От противного:** Предположим что нет
 - У каждого би-вал есть хотя бы один би-вал ребенок... Конечность!

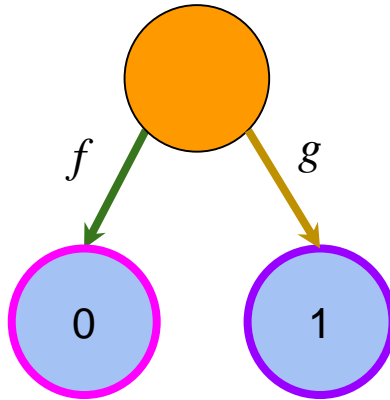


Модель

- Определения и концепции:
 - Исходные объекты **атомарны**. Значит любое исполнения можно рассматривать как последовательное в каком-то порядке
 - Рассматриваем дерево состояния, листья – конечные состояния помеченные 0 или 1 (в зависимости от значения консенсуса).
 - **х-валентное состояние системы** ($x = 0, 1$) – консенсус во всех нижестоящих листьях будет x .
 - **Бивалентное состояние** – возможен консенсус как 0 так и 1.
 - **Критическое состояние** – такое бивалентное состояние, все дети которого одновалентны.
- **ЛЕММА 1:** Существует начальное бивалентное состояние.
- **ЛЕММА 2:** Существует критическое состояние.

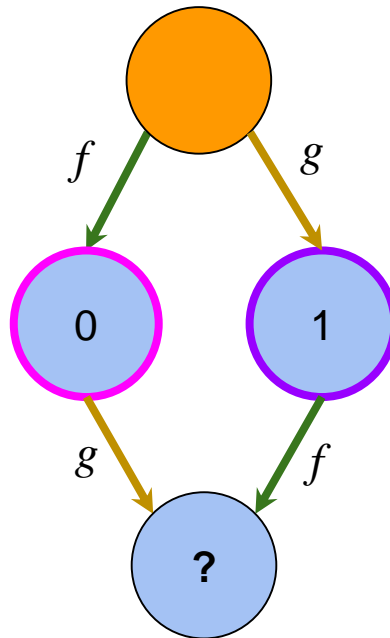
Теорема для атомарных регистров

- Рассмотрим возможные пары операций в критическом состоянии



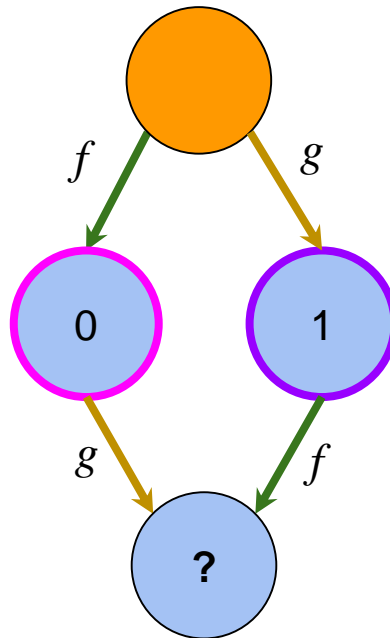
Теорема для атомарных регистров

- Рассмотрим возможные пары операций в критическом состоянии:
 - Операции над разными регистрами – коммутируют.



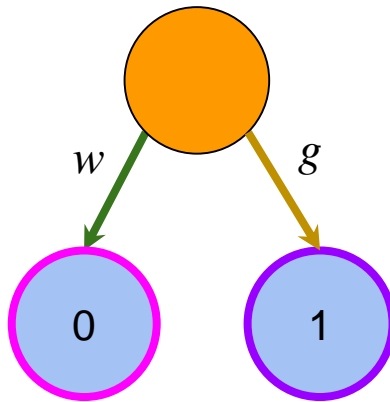
Теорема для атомарных регистров

- Рассмотрим возможные пары операций в критическом состоянии:
 - Операции над разными регистрами – коммутируют.
 - Два чтения – коммутируют.



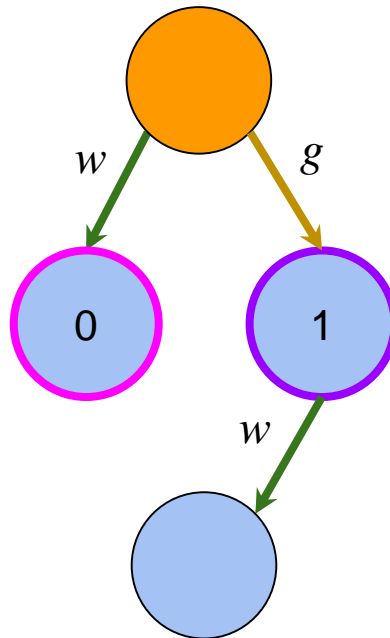
Теорема для атомарных регистров

- Рассмотрим возможные пары операций в критическом состоянии:
 - Операции над разными регистрами – коммутируют.
 - Два чтения – коммутируют.
 - Чтение/Запись + Запись - КОНФЛИКТУЮЩИЕ ОПЕРАЦИИ**



Теорема для атомарных регистров

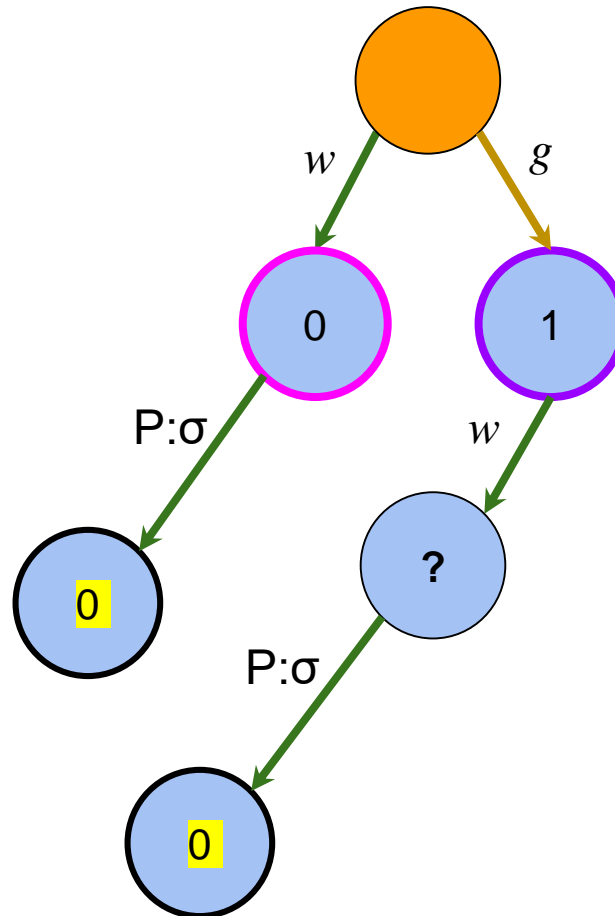
- Рассмотрим возможные пары операций в критическом состоянии:
 - Операции над разными регистрами – коммутируют.
 - Два чтения – коммутируют.
 - Чтение/Запись + Запись - КОНФЛИКТУЮЩИЕ ОПЕРАЦИИ**



Теорема для атомарных регистров

- **Чтение/Запись + Запись**

состояние пишущего потока не зависит от порядка операций.



Консенсус не решить с
помощью регистров!
Нужные более сильные
примитивы

Read-Modify-Write регистры

```
class RMWRegister:
    private shared int reg

    def read():
        return reg

    def getAndF(args):
        do atomically:
            old = reg
            reg = F(args)(reg)
            return old
```

- Для функции или класса функций F(args)
 - getAndSet (exchange), getAndIncrement, getAndAdd и т.п.
 - get (read) это тоже [тривиальная] RMW операция без дополнительных аргументов для F() == id.

Нетривиальные RMW регистры

```
threadlocal int    id // 0 or 1
```

```
shared RMWRegister rmw  
shared int proposed[2]
```

```
def decide(val):  
    proposed[id] = val  
    if rmw.getAndF() == v0:  
        return proposed[id]  
    else:  
        return proposed[1-id]
```

Реализация протокола
для 2-х потоков

- Консенсусное число нетривиального RMW регистра ≥ 2 .
 - Нужно чтобы была хотя бы одна «подвижная» точка функции F , например $F(v0) == v1 \neq v0$.

Common2 RMW регистры

- Определения
 - $F1$ и $F2$ коммутируют если $F1(F2(x)) == F2(F1(x))$
 - $F1$ перезаписывает $F2$ если $F1(F2(x)) == F1(x)$
 - Класс C RMW регистров принадлежит Common2, если любая пара функций либо коммутирует, либо одна из функций перезаписывает другую.
- **ТЕОРЕМА:** Нетривиальный класс Common2 RMW регистров имеет консенсусное число 2.
 - Третий поток не может отличить глобальное состояние при изменении порядка выполнения коммутирующих или перезаписывающих операций в критическом состоянии.

Универсальные объекты

```
class CASRegister:
    private shared int reg

    def CAS(expect, update):
        do atomically:
            old = reg
            if old == expect:
                reg = update
                return true
            return false
```

- Объект с консенсусный числом бесконечность называется **«универсальный объект»**.
 - По определению, с помощью него можно реализовать консенсусный протокол для любого числа потоков.

Примеры:

- compareAndSet (CAS) aka testAndSet (возвращает boolean)
- compareAndExchange aka CMPXCHG (возвращает старое значение как и положено RMW операции)

CAS и консенсус

```
def decide(val):  
    if CAS(NA, val):  
        return val  
    else:  
        return read()
```

**Реализация протокола
через CAS+READ**

```
def decide(val):  
    old = CMPXCHG(NA, val):  
    if old == NA:  
        return val  
    else:  
        return old
```

**Реализация протокола
через CMPXCHG**

Универсальность консенсуса

- **ТЕОРЕМА:** Любой последовательный объект можно реализовать без ожидания (wait-free) для N потоков используя консенсусный протокол для N потоков.
 - Такое построение называется **универсальная конструкция**
 - **Следствие 1:** С помощью любого класса объектов с консенсусным числом N можно реализовать любой объект с консенсусным числом $\leq N$.
 - **Следствие 2:** С помощью универсального объекта можно реализовать вообще любой объект
 - Сначала реализуем консенсус для любого числа потоков (по определению универсального объекта)
 - Потом через консенсус любой другой объект используя универсальную конструкцию.

Универсальная конструкция без блокировки (lock-free) через CAS

shared CASRegister reg

```
def concurrentOperationX(args):  
    loop:  
        old = reg.read()  
        upd = old.deepCopy()  
        res = upd.serialOperationX(args)  
    until reg.CAS(old, upd)  
    return res
```

- ✓ Если reg хранит указатель на данные, то deepCopy должен сделать полную копию

- **Без блокировки**
универсальная конструкция проста и практична, если использовать **CAS** в качестве примитива.
 - Для реализации **через консенсус** надо чтобы каждый объект консенсуса использовался потоком один раз
 - Для реализации **без ожидания** нужно чтобы потоки помогали друг другу

Универсальная конструкция через консенсус

```
class Node:
    val                // readonly
    Consensus next    // init fresh obj

shared Node root      // readonly
threadlocal Node last // init root

def concurrentOperationX(args):
    loop:
        old = last.val
        upd = old.deepCopy()
        res = upd.serialOperationX(args)
        node = new Node(upd)
        last = last.next.decide(node)
    until last == node // until we're in list
    return res
```

- **Идея:** Представим объект в виде односвязного списка состояний.
 - Последний элемент в списке это текущее состояние
 - Объект консенсуса для списка следующий состояний. Каждый поток предлагает свой вариант и они приходят к консенсусу
- Получаем реализацию **без блокировки** очевидным образом

Универсальная конструкция без ожидания (1)

```
class Node:
```

```
    args                // readonly  
    Consensus next // init fresh obj
```

```
threadlocal Node last // init root
```

```
threadlocal my
```

```
def concurrentOperationX(args):
```

```
    Node node = new Node(args)  
    while last != node: // until we're in list  
        last = last.next.decide(node)  
        res = my.serialOperationX(last.args)  
    return res
```

- **Идея 1:** Хранить в узле операцию которую надо выполнить, а не результат её выполнения
 - Каждый поток будет хранить и обновлять свою локальную копию объекта

Универсальная конструкция без ожидания (2)

```
class Node:
```

```
    int seq          // init 0  
    args            // readonly  
    Consensus next  // init fresh obj
```

```
def concurrentOperationX(args):
```

```
    Node node = new Node(args)  
    while last != node: // until we're in list  
        Node prev = last  
        last = prev.next.decide(node)  
        last.seq = prev.seq + 1  
        res = my.serialOperationX(last.args)  
    return res
```

- **Идея 2:** Будет нумеровать выполненные операции последовательными целыми числами, заведя переменную seq:
 - После выполнения будем прописывать номер выполненной операции в Node.seq как только она успешно выполнена

Универсальная конструкция без ожидания (3)

```
shared Node[] know    // init root

def concurrentOperationX(args):
  Node node = new Node(args)
  know[id] = maxSeqFrom(know)
  while know[id] != node: // until we're in
    Node prev = know[id]
    know[id] = prev.next.decide(node)
    know[id].seq = prev.seq + 1
  return updateMyLastTo(node)

def updateMyLastTo(node):
  while last != node:
    res = my.serialOperationX(last.args)
    last = last.next
  return res
```

- **Идея 3:** Каждый поток будет хранить последнее известное ему значение конца списка в элементе массива `know[id]` который виден всем остальным объектам
 - Перед началом работы будем выбирать там элемент с макс. seq
 - Это позволит более медленным потокам догонять более быстрые и конкурировать в decide

Универсальная конструкция без ожидания (4)

```
shared Node[] announce // init root

def concurrentOperationX(args):
  announce[id] = new Node(args)
  know[id] = maxSeqFrom(know)
  // loop until we're in list
  while announce[id].seq == 0:
    Node help =
      announce[know[id].seq % N]
    Node prev = help if help.seq == 0
      else announce[id]
    know[id] = prev.next.decide(node)
    know[id].seq = prev.seq + 1
  know[id] = announce[id]
  return updateMyLastTo(announce[id])
```

- **Идея 4:** Каждый поток будет заранее записывать операция которую он планирует выполнить в массив announce
 - И пользуясь этим массивом помогать потоку $\text{seq} \% N$ (N – число потоков)
 - Предпочитая помощь своему выполнению
 - Тогда за N шагов каждому потоку помогут

Универсальность консенсуса

Примитив А:
Консенсусное число N



Консенсус N потоков



Примитив В:
Консенсусное число N

Иерархия объектов

Объект	Консенсусное число
Атомарные регистры, снимок состояния нескольких регистров	1
getAndSet (атомарный обмен), getAndAdd, очередь, стек	2
Атомарная запись m регистров из $m(m+1)/2$ регистров	m
compareAndSet, LoadLinked/StoreConditional	∞

Универсальная конструкция на практике

Универсальное построение без блокировок с использованием CAS

- Вся структура данных представляется как указатель на объект, **содержимое которого никогда не меняется.**

Универсальное построение без блокировок с использованием CAS

- Вся структура данных представляется как указатель на объект, **содержимое которого никогда не меняется.**
- Любые операции чтения работают без ожидания.

Универсальное построение без блокировок с использованием CAS

- Вся структура данных представляется как указатель на объект, **содержимое которого никогда не меняется.**
- Любые операции чтения работают без ожидания.
- Любые операции модификации создают полную копию структуры, меняют её, из пытаются подменить указать на неё с помощью одного Compare-And-Set (CAS).
 - В случае ошибки CAS – повтор.

Универсальное построение без блокировок с использованием CAS

- Вся структура данных представляется как указатель на объект, **содержимое которого никогда не меняется**.
- Любые операции чтения работают без ожидания.
- Любые операции модификации создают полную копию структуры, меняют её, из пытаются подменить указать на неё с помощью одного Compare-And-Set (CAS).
 - В случае ошибки CAS – повтор.
- Частный случай этого подхода: вся структура данных влезает в одно машинное слово, например счетчик.

Атомарный счетчик

```
val counter: AtomicInt
```

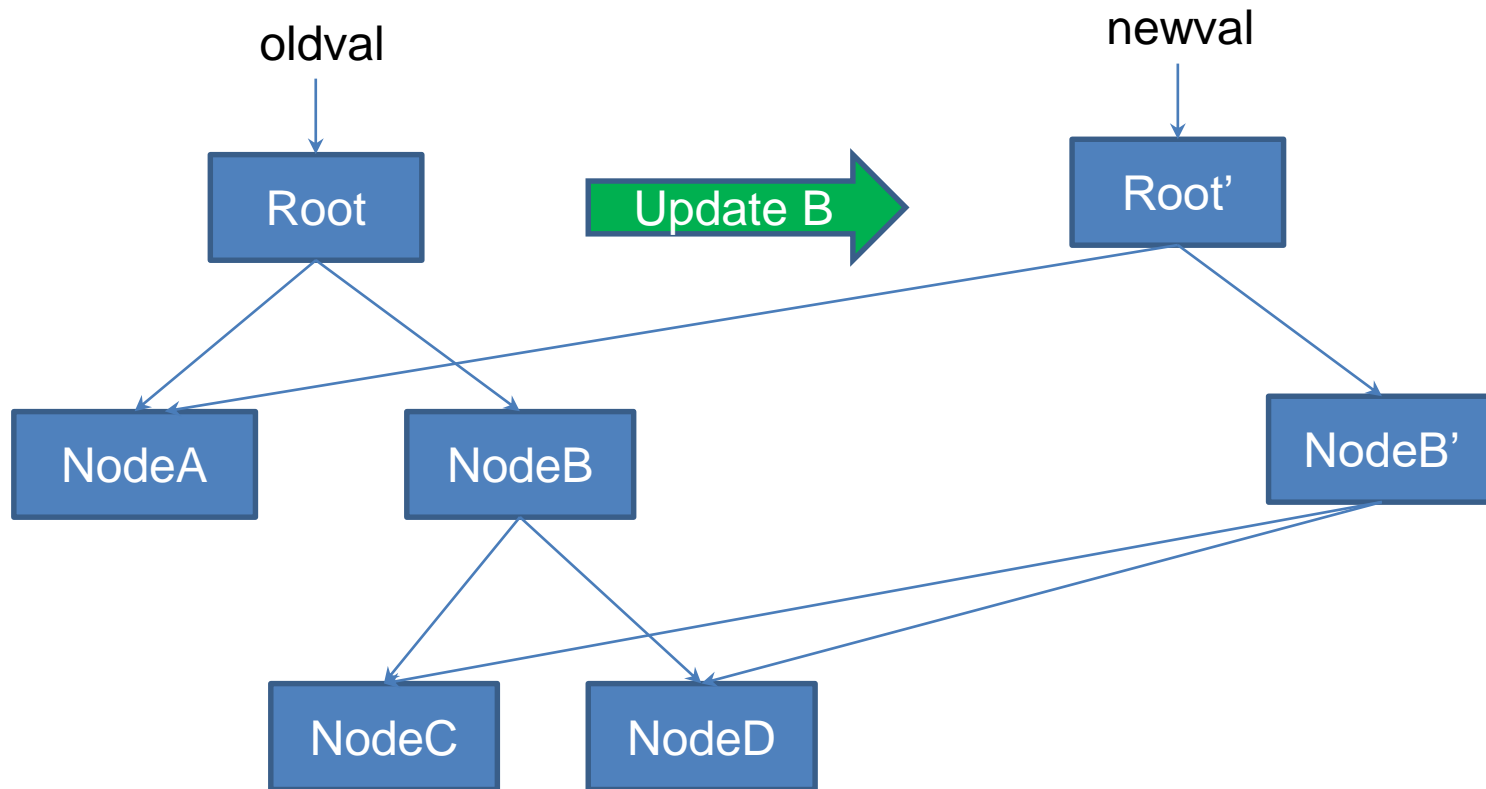
```
fun getAndIncrement(increment: Int) {  
    while (true) {  
        val old = counter  
        val updated = old + increment  
        if (CAS(counter, old, updated))  
            return old  
    }  
}
```

```
// В Java & Kotlin AtomicInt & CAS это  
// java.util.concurrent.atomic.AtomicInteger  
// там метод getAndIncrement уже есть
```

Работа с древовидными структурами без блокировок

- Структура представлена в виде дерева
- Тогда операции изменения можно реализовать в виде **одного** CAS, заменяющего указатель на root дерева.
 - Неизменившуюся часть дерева можно использовать в новой версии дерева, т.е. не нужно копировать всю структуру данных.
 - Это т.н. **персистентные структуры данных**

Персистентная древовидная структура



LIFO стек

- Частный случай вырожденной древовидной структуры это LIFO стек: **Алгоритм Трейбера** (treiber stack)

```
class Node(  
    // Узел никогда не меняется. Все поля final  
    // Меняется только корень (top)  
    val item: T,  
    val next: Node  
)  
  
// Пустой стек это указатель на null  
// AtomicReference чтобы делать CAS на ссылкой  
val top = AtomicReference<Node>(null)
```

Операции с LIFO стеком

```
fun push(item: T) {  
    while (true) {  
        val node = Node(item, top.get())  
        if (top.compareAndSet(node.next, node))  
            return  
    }  
}
```

линеаризация

```
fun pop(): T {  
    while (true) {  
        val node = top.get()  
        if (node == null) throw EmptyStack()  
        if (top.compareAndSet(node, node.next))  
            return node.item  
    }  
}
```

линеаризация