



Лекции «Многопоточное программирование»

© Роман Елизаров, Devexperts, 2015

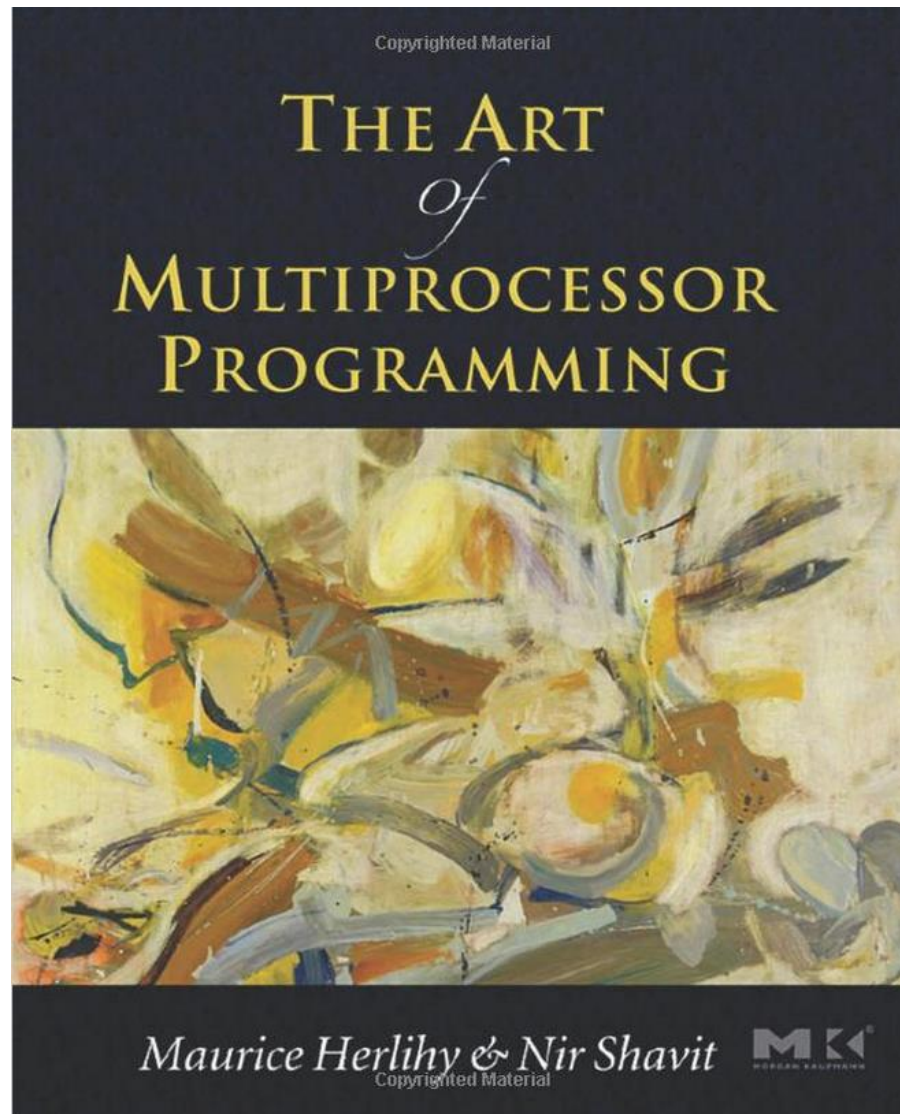


DEVEXPERTS



ЛИТЕРАТУРА

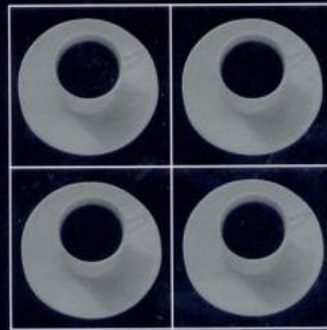
Для самостоятельной подготовки





Copyrighted Material

Concurrent and Distributed Computing in Java



VIJAY K. GARG

Copyrighted Material



DEVEXPERTS



Лекция 5

ПРАКТИЧЕСКИЕ ПОСТРОЕНИЯ НА СПИСКАХ

Детали реализации





Реализация многопоточных алгоритмов

- **Проблемы:**
 - Компилятор оптимизирует код и переставляет операции
 - Среда исполнения (процессов) переставляет операции
- Как реализовать правильный псевдокод на реальном языке программирования?
 - Будем пытаться писать реальные программы на Java



Что нужно сделать, чтобы корректные программы корректно работали?

```
int flag, value; // общие переменные, обе 0 в начале
```

```
void init() {  
    value = 2;  
    flag = 1;  
}
```

```
int take() {  
    while (flag == 0); // ждем  
    return value;  
}
```

```
// какие возможны значения результата take()  
// при параллельной работе с init() ?
```




Проблемы и обходные пути

- Оптимизации в компиляторах
 - изменение порядка операций, устранений общих подвыражений, использование регистров и т.п.
 - Обходной путь: отключить оптимизации, использовать «непрозрачные» для компилятора внешние вызовы
- Оптимизации в процессорах
 - буфера для записи, спекулятивное чтение, кэши, несимметричная память и т.п.
 - Обходной путь: специальные команды «синхронизации» (membar, fence) которые заставляют процессор сделать видимость последовательного исполнения в ущерб производительности



Модель [согласованности] памяти

- Контракт между средой исполнения (компиляторы + процессор)
- Идеал для программиста – последовательная согласованность
 - но слишком «дорого» (нельзя делать никаких оптимизаций)
- Нужно определять более слабую модель
 - Пионером был язык Java
 - Это послужило основной модели памяти для C++11



Java Memory Model

- Зачем Java модель памяти?
 - Трюки используемые C/C++ программистами не работают в «чистом» Java коде без native методов.
 - Java работает на широком классе платформ с разнообразной архитектурой.
 - Два этапа компиляции (Java source -> Byte code и Byte code -> Native Code) усложняют анализ того, что может случиться с программой и у программиста нет прямого контроля над конечным нативным кодом
 - Нет возможности написать `#ifdef SOME_ARCH` и т.п.



Основы JMM

- Межпоточные действия (vs внутрипоточные действия)
 - Обычные: чтение и запись разделяемых переменных
 - Операции синхронизации
 - Чтение и запись переменных **volatile**
 - Блокировка и разблокировка (вход и выход в **synchronized**)
 - Запуск/останов потоков и прочее
- Отношение синхронизации (synchronizes-with) и отношение произошло-до (happens-before)
- Понятие конфликтующего доступа (conflicting access) и гонки за данными (data race, race condition)
 - Понятие «корректно синхронизированной программы»



Гарантии JMM

- **Выполнение корректно синхронизированной программы будет выглядеть последовательно согласовано.**
- Гонки за данными не могут нарушить базовые гарантии безопасности платформы:
 - Система типов (**instanceof** и т.п.), длины массивов
 - Все типы кроме **long** и **double** пишутся и читаются атомарно даже в отсутствии синхронизации
 - Все поля гарантировано инициализированы нулями (нельзя увидеть там «мусор»)
 - Дополнительные гарантии для неизменяемых объектов (при использовании **final** полей)



DEVEXPERTS

Рабочий вариант #1

```
volatile int flag; // всего один volatile
// ... решает проблемы видимости и упорядоченности
int value;

void init() {
    value = 2;
    flag = 1;
}

int take() {
    while (flag == 0); // ждем... кушаем CPU
    return value;
}
```




DEVEXPERTS

Рабочий вариант #2

```
public class Value {  
    int flag, value;  
  
    void synchronized init() {  
        value = 2; // здесь порядок уже не важен  
        flag = 1;  
        notifyAll(); // разбудить ожидания  
    }  
  
    int synchronized take() throws InterruptedException {  
        while (flag == 0) wait();  
        return value;  
    }  
}
```



Многопоточные (Thread-Safe) объекты (алгоритмы и структуры данных) на практике

- Многопоточный объект включает в себя синхронизацию потоков (блокирующую или не блокирующую), которая позволяет его использовать из нескольких потоков одновременно без дополнительной внешней синхронизации
 - Специфицируется через последовательное поведение
 - По умолчанию требуется **линеаризуемость** операций (более слабые формы согласованности – редко)
 - Редко удастся реализовать все операции без ожидания (wait-free). Часто приходится идти на компромиссные решения.
 - Проще всего реализовать с блокировками
 - Lock-free это наиболее частая гарантия с независимым прогрессом, которая как раз удачно имеет название «без блокировок»

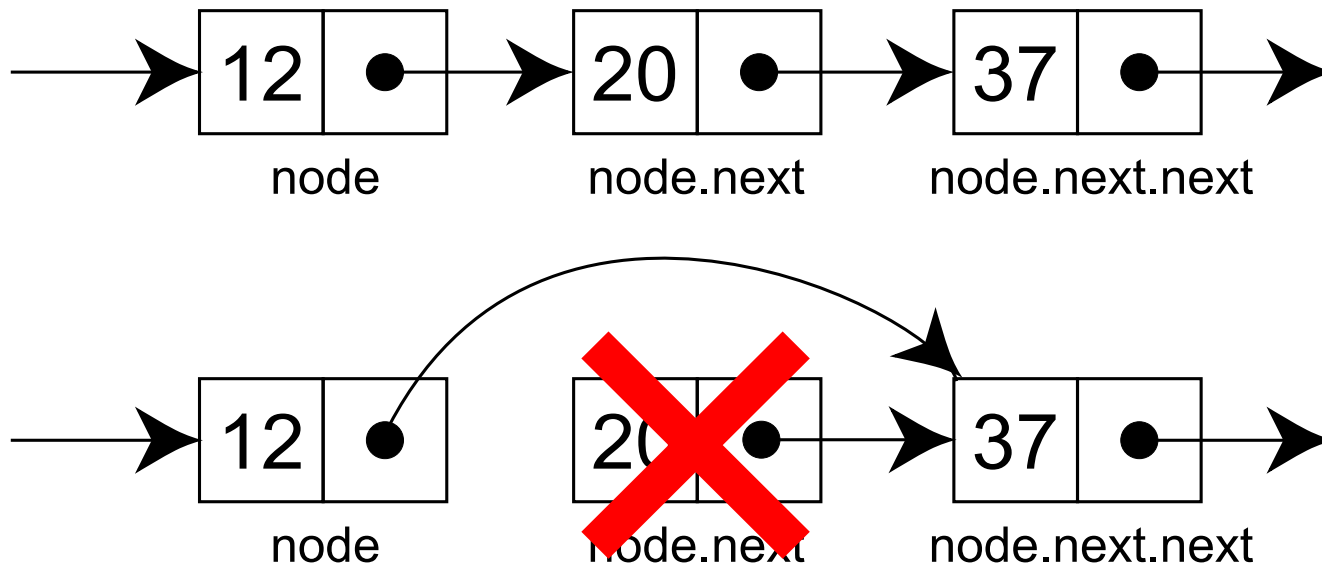


Разные подходы к синхронизации потоков при работе с общей структурой данных

- Типы синхронизации:
 - Грубая (Coarse-grained) синхронизация
 - Тонкая (Fine-grained) синхронизация
 - Оптимистичная (Optimistic) синхронизация
 - Ленивая (Lazy) синхронизация
 - Неблокирующая (Nonblocking) синхронизация (lock-free, wait-free и т.п.)
- Проще всего для списочных структур данных (с них и начнем), хотя на практике массивы работают существенно быстрее



Многопоточные связанные списки





Множество на основе односвязного списка

- **ИНВАРИАНТ:** `node.key < node.next.key`
 - Будем хранить ключи в порядке возрастания

```
class Node {  
    final int key; // никогда не меняем  
    final T item; // никогда не меняем  
    Node next;  
}
```

```
// Пустой список состоит из 2-х граничных элементов  
Node head = new Node(Integer.MIN_VALUE, null);  
head.next = new Node(Integer.MAX_VALUE, null);
```



Грубая синхронизация

- Обеспечиваем взаимное исключение всех операций через общий **java.util.concurrent.locks.ReentrantLock** lock.
 - Это реализация объекта взаимного исключения с операциями `lock()` и `unlock()`.
 - Дает немного больше функционала чем секции **synchronized**.

```
class LinkedSet {  
    final Node head;  
    final Lock lock; // свой mutex  
  
    // методы здесь  
}
```




DEVEXPERTS

Грубая синхронизация: поиск

```
boolean contains(int key) {  
    lock.lock();  
    try {  
        Node curr = head;  
        while (curr.key < key) {  
            curr = curr.next;  
        }  
        return key == curr.key;  
    } finally { lock.unlock(); }  
}
```



DEVEXPERTS

Грубая синхронизация: добавление

```
boolean add(int key, T item) {  
    lock.lock();  
    try {  
        Node pred = head, curr = pred.next;  
        while (curr.key < key) {  
            pred = curr; curr = curr.next;  
        }  
        if (key == curr.key) return false; else {  
            Node node = new Node(key, item);  
            node.next = curr; pred.next = node;  
            return true;  
        }  
    } finally { lock.unlock(); }  
}
```



DEVEXPERTS

Грубая синхронизация: удаление

```
boolean remove(int key, T item) {  
    lock.lock();  
    try {  
        Node pred = head, curr = pred.next;  
        while (curr.key < key) {  
            pred = curr; curr = curr.next;  
        }  
        if (key == curr.key) {  
            pred.next = curr.next;  
            return true;  
        } else {  
            return false;  
        }  
    } finally { lock.unlock(); }  
}
```



Тонкая синхронизация

- Обеспечиваем синхронизацию через взаимное исключение на каждом элементе.
- При любых операциях одновременно удерживаем блокировку текущего и предыдущего элемента (чтобы не потерять инвариант `pred.next == curr`).

```
class Node {  
    final int key;  
    final T item;  
    final Lock lock; // на каждом узле свой mutex  
    Node next;  
  
    void lock() { lock.lock(); }  
    void unlock() { lock.unlock(); }  
}
```



DEVEXPERTS

Тонкая синхронизация: поиск

```
Node pred = head; pred.lock();  
Node curr = pred.next; curr.lock();  
try {  
    while (curr.key < key) {  
        pred.unlock(); pred = curr;  
        curr = curr.next; curr.lock();  
    }  
    return key == curr.key;  
} finally { curr.unlock(); pred.unlock(); }
```



DEVEXPERTS

Тонкая синхронизация: добавление

```
Node pred = head; pred.lock();
Node curr = pred.next; curr.lock();
try {
    while (curr.key < key) {
        pred.unlock(); pred = curr;
        curr = curr.next; curr.lock();
    }
    if (key == curr.key) return false; else {
        Node node = new Node(key, item);
        node.next = curr; pred.next = node;
        return true;
    }
} finally { curr.unlock(); pred.unlock(); }
```




DEVEXPERTS

Тонкая синхронизация: удаление

```
Node pred = head; pred.lock();
Node curr = pred.next; curr.lock();
try {
    while (curr.key < key) {
        pred.unlock(); pred = curr;
        curr = curr.next; curr.lock();
    }
    if (key == curr.key) {
        pred.next = curr.next;
        return true;
    } else {
        return false;
    }
} finally { curr.unlock(); pred.unlock(); }
```



Оптимистичная синхронизация

- Ищем элемент без синхронизации (оптимистично предполагая что никто не помешает), но перепроверяем с синхронизацией
 - Если перепроверка обломалась, то начинаем операцию заново
 - Поиск не заикнется, ибо ключи упорядочены, никогда не меняются внутри Node и значения next не могут возникнуть «с потолка» даже при чтении без синхронизации
- Имеет смысл только если обход структуры дешев и быстр, а обход с синхронизацией медленный и дорогой
- Потоки всегда синхронизируются между собой (“synchronizes with”) через критические секции, поэтому никаких дополнительных механизмов синхронизации не нужно
 - Линеаризация происходит благодаря критическим секциям (операции упорядочены в том порядке, в котором они входят в критическую секцию)



DEVEXPERTS

Оптимистичная синхронизация: поиск

```
retry: while (true) {  
    Node pred = head, curr = pred.next;  
    while (curr.key < key) {  
        pred = curr; curr = curr.next;  
        if (curr == null) continue retry; // может!!!  
    }  
    pred.lock(); curr.lock();  
    try {  
        if (!validate(pred, curr)) continue retry;  
        return curr.key == key;  
    } finally { curr.unlock(); pred.unlock(); }  
}
```



DEVEXPERTS



Оптимистичная синхронизация: валидация

```
boolean validate(Node pred, Node curr) {  
    Node node = head;  
    while (node.key <= pred.key) {  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
        if (node == null) return false;  
    }  
    return false;  
}
```



DEVEXPERTS

Оптимистичная синхронизация: добавление

```
retry: while (true) {  
    Node pred = head, curr = pred.next;  
    while (curr.key < key) {  
        pred = curr; curr = curr.next;  
        if (curr == null) continue retry; // может!!!  
    }  
    pred.lock(); curr.lock();  
    try {  
        if (!validate(pred, curr)) continue retry;  
        if (curr.key == key) return false; else {  
            Node node = new Node(key, item);  
            node.next = curr; pred.next = node;  
            return true;  
        }  
    }  
    finally { curr.unlock(); pred.unlock(); }  
}
```



Оптимистичная синхронизация: удаление

```
retry: while (true) {  
    Node pred = head, curr = pred.next;  
    while (curr.key < key) {  
        pred = curr; curr = curr.next;  
        if (curr == null) continue retry; // может!!!  
    }  
    pred.lock(); curr.lock();  
    try {  
        if (!validate(pred, curr)) continue retry;  
        if (curr.key == key) {  
            pred.next = curr.next;  
            return true;  
        } else return false;  
    } finally { curr.unlock(); pred.unlock(); }  
}
```




DEVEXPERTS

Ленивая синхронизация

- Добавляем в Node поле `boolean marked`.
- Удаление в 2 фазы:
 - `node.marked = true;` // Логическое удаление
 - Физическое удаление из списка
- Инвариант: Все непомеченные (не удаленные) элементы всегда в списке
- Результат:
 - Для валидации не надо просматривать список (только проверить что элементы не удалены логически и `pred.next == curr`). В остальном, код добавление идентичен оптимистичному.



DEVEXPERTS

Ленивая синхронизация: валидация

```
boolean validate(Node pred, Node curr) {  
    return !pred.marked &&  
           !curr.marked &&  
           pred.next == curr;  
}
```



Ленивая синхронизация и поиск без ожидания

- Добавим wait-free поиск
- Какая точка линеаризации будет у поиска, если он делается wait-free и нет критической секции, по входу в которую он будет упорядочен с операциями изменения?
 - С операциями изменения успешный поиск будет линеаризоваться по изменению поля `next` и последующего его чтения при поиске
 - Для этого **next** надо объявить как **volatile**
 - Заодно гарантируем, что `next` у добавленного в список Node не может получиться при чтении **null**
 - Линеаризация *неуспешного* поиска происходит более сложно и зависит от того какие операции выполнялись параллельно и в каком порядке



Ленивая синхронизация: узел

- ВАЖНОСТЬ `volatile` для линейризуемости!!!
 - Без **`volatile`** next поиск, незащищенный критической секцией, может вообще не увидеть элемент, который был добавлен до *начала* этого поиска (=> не линейризуем)

```
class Node {  
    final int key;  
    final T item;  
    final Lock lock;  
    boolean marked;  
    volatile Node next;  
}
```



Ленивая синхронизация: добавление

```
retry: while (true) {  
    Node pred = head, curr = pred.next;  
    while (curr.key < key) {  
        pred = curr; curr = curr.next;  
    }  
    pred.lock(); curr.lock();  
    try {  
        if (!validate(pred, curr)) continue retry;  
        if (curr.key == key) return false; else {  
            Node node = new Node(key, item);  
            node.next = curr;  
            pred.next = node;  
            return true;  
        }  
    }  
    finally { curr.unlock(); pred.unlock(); }  
}
```

точно != null

сначала!!!

точка линеаризации



Ленивая синхронизация: удаление

```
retry: while (true) {  
    Node pred = head, curr = pred.next;  
    while (curr.key < key) {  
        pred = curr; curr = curr.next;  
    }  
    pred.lock(); curr.lock();  
    try {  
        if (!validate(pred, curr)) continue retry;  
        if (curr.key == key) {  
            curr.marked = true; // для validate  
            pred.next = curr.next; ← точка линеаризации  
            return true;  
        } else return false;  
    } finally { curr.unlock(); pred.unlock(); }  
}
```



Ленивая синхронизация: поиск (wait-free!)

```
boolean contains(int key) {  
    Node curr = head;  
    while (curr.key < key) {  
        curr = curr.next;  
    }  
    return key == curr.key;  
}
```

точка линеаризации
успешного поиска

Однако, если множество ключей не ограничено сверху, то поиск не будет wait-free, ибо после его curr позиции могут постоянно добавляться элементы. Но он, в любом случае, без блокировок (lock-free)



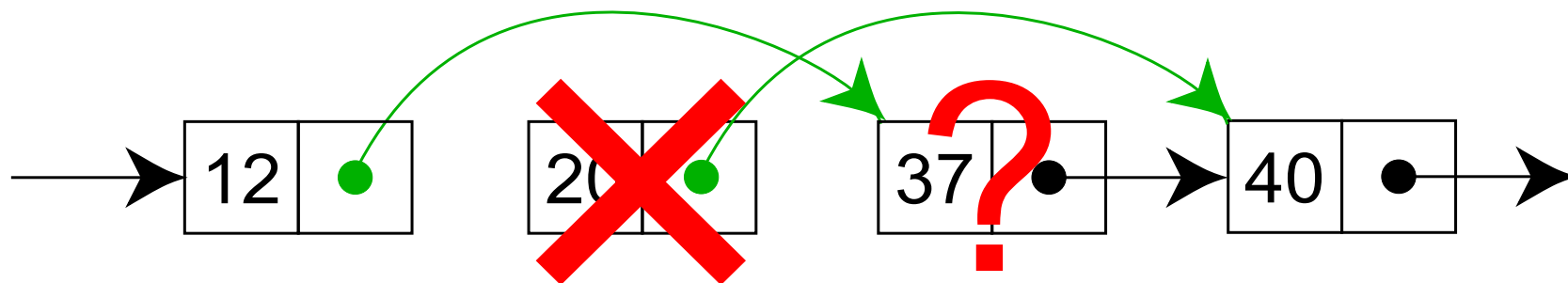
Вариации на тему поиска без блокировок

- Если изменения в структуре данных защищены любой синхронизацией (не только ленивой), то можно сделать поиск без блокировок
 - Например, изменения могут использовать **грубую** синхронизацию (один lock на всё)
 - Тогда не нужна валидация, а значит не нужно поле `marked`
 - Всё что нужно для линеаризуемого поиска без синхронизации это объявить поле `next` как **volatile**.
 - Здесь мы фундаментальным образом полагаемся на GC и на тот факт, что один и тот же Node не может быть использован два раза, и что `key` и `value` в нем никогда не меняются



Неблокирующая синхронизация (1)

- Простое использование Compare-And-Set (CAS) не помогает – удаления двух соседних элементов будут конфликтовать.
 - Удаляем одновременно элементы 20 и 37
 - Конфликта по изменяемым указателям (зеленые) нет
 - Но элемент 37 оказывается в списке после удаления!





Неблокирующая синхронизация (2)

- Объединим `next` и `marked` в одну переменную, пару `(next,marked)`, которую будем атомарно менять используя CAS
 - Тогда одновременное удаление двух соседних элементов будет конфликтовать (элемент 20 в примере удаляется и меняет `next`)
 - Каждая операция модификации будет выполняться одним успешным CAS-ом.
 - Успешное выполнение CAS-а является **точкой линеаризации**.
- При выполнении операции удаления или добавления будем пытаться произвести физическое удаление
 - Добавление и удаление будут работать без блокировки (lock-free)
 - Поиск элемента будет работать без ожидания (wait-free)



DEVEXPERTS

Неблокирующая синхронизация: добавление

// пока это псевдокод... сейчас доведем до Java

```
retry: while (true) {
```

```
    // метод find возвращает пару (pred, curr)
```

```
    (Node pred, Node curr) = find(key);
```

```
    if (curr.key == key) return false; else {
```

```
        Node node = new Node(key, item);
```

```
        node.(next, marked) = (curr, false);
```

линеаризация →

```
        if (CAS(pred.(next, marked), (curr, false),  
                (node, false))
```

```
        return true;
```

```
    }
```

```
}
```

Пара (next, marked) с поддержкой операции CAS реализуется в Java с помощью класса **java.util.concurrent.atomic.AtomicMarkableReference**



DEVEXPERTS

Неблокирующая синхронизация: узел

- Напишем настоящую Java реализацию
 - Больше не нужен lock.
 - **AtomicMarkableReference** работает как **volatile**

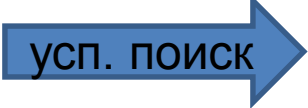
```
class Node {  
    final int key;  
    final T item;  
    final AtomicMarkableReference<Node> next;  
}
```

// Так будем возвращать пару узлов из find

```
class Window {  
    final Node pred;  
    final Node curr;  
}
```



Неблокирующая синхронизация: поиск окна

```
Window find(int key) {  
    retry: while (true) {  
        Node pred = head, curr = pred.next.getReference();  
        boolean[] marked = new boolean[1];  
        while (true) {  
             Node succ = curr.next.get(marked);  
            if (marked[0]) { // Если curr удален  
                if (!pred.next.compareAndSet(  
                    curr, succ, false, false))  
                    continue retry;  
                curr = succ;  
            } else {  
                if (curr.key >= key)  
                    return new Window(pred, curr);  
                pred = curr; curr = succ;  
            }  
        }  
    }  
}
```



Неблокирующая синхронизация: поиск

- Здесь уже всё просто
- Поиск фактически полностью выполняется методом `find`
 - Точка линеаризации успешного поиска это чтение пары `(next,marked)` в узле `curr`

```
boolean contains(int key) {  
    Window w = find(key);  
    return w.curr.key == key;  
}
```



DEVEXPERTS

Неблокирующая синхронизация: добавление'

```
// вот так выглядит реальный Java код добавления
retry: while (true) {
    Window w = find(key);
    Node pred = w.pred, curr = w.curr;
    if (curr.key == key) return false; else {
        Node node = new Node(key, item);
        node.next.set(curr, false);
        if (pred.next.compareAndSet(
            curr, node, false, false))
            return true;
    }
}
```



Неблокирующая синхронизация: удаление

```
retry: while (true) {  
    Window w = find(key);  
    Node pred = w.pred, curr = w.curr;  
    if (curr.key != key) return false; else {  
        Node succ = curr.next.getReference();  
        if (!curr.next.attemptMark(succ, true))  
            continue retry;  
        // оптимизация – попытаемся физ. удалить  
        pred.next.compareAndSet(  
            curr, succ, false, false);  
        return true;  
    }  
}
```

линеаризация



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.



DEVEXPERTS

Атомарный счетчик

```
int counter;
```

```
int getAndIncrement(int increment) {  
    while (true) {  
        int old = counter;  
        int updated = old + increment;  
        if (CAS(counter, old, updated))  
            return old;  
    }  
}
```

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



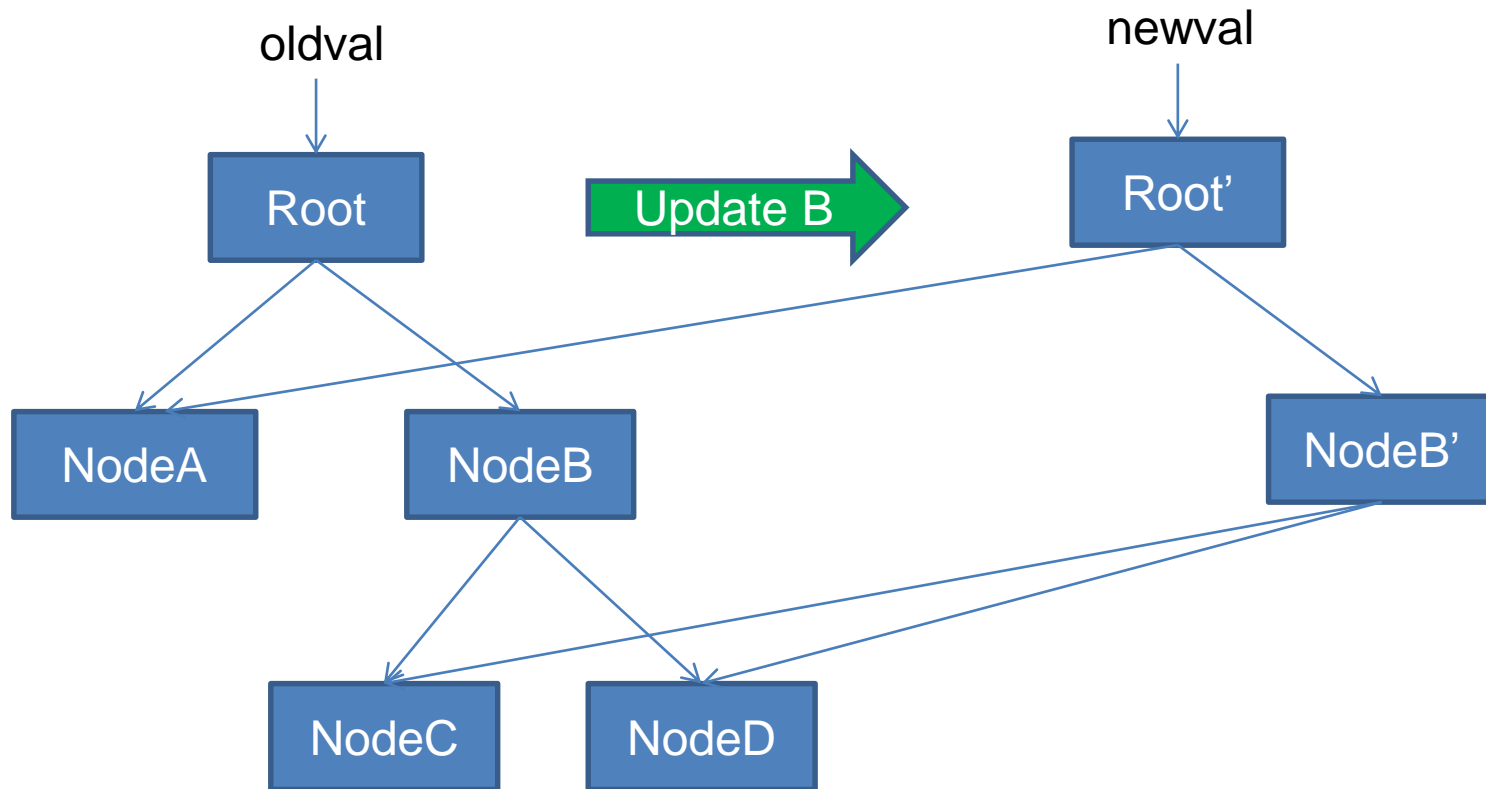
DEVEXPERTS

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

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



Операции с LIFO стеком

```
void push(T item) {  
    while (true) {  
        Node node = new Node(item, top.get());  
        if (top.compareAndSet(node.next, node))  
            return;  
    }  
}  
T pop() {  
    while (true) {  
        Node node = top.get();  
        if (node == null) throw new EmptyStack();  
        if (top.compareAndSet(node, node.next))  
            return node.item;  
    }  
}
```

линеаризация

линеаризация



FIFO очередь без блокировок (lock-free)

- Так же односвязный список, но два указателя: head и tail.
- Алгоритм придумали Michael & Scott в 1996 году.

```
class Node {  
    T item;  
    final AtomicReference<Node> next;  
}
```

```
// Пустой список состоит из одного элемента-заглушки  
AtomicReference<Node> head =  
    new AtomicReference<Node>(new Node(null));  
AtomicReference<Node> tail =  
    new AtomicReference<Node>(head.get()) ;
```



Добавление в очередь

```
void enqueue(T item) {  
    Node node = new Node(item);  
    retry: while (true) {  
        Node last = tail.get(),  
            next = last.next.get();  
        if (next == null) {  
            if (!last.next.compareAndSet(null, node))  
                continue retry;  
            // оптимизация – сами переставляем tail  
            tail.compareAndSet(last, node);  
            return;  
        }  
        // Помогаем другим операциям enqueue с tail  
        tail.compareAndSet(last, next);  
    }  
}
```

линеаризация



Удаление из очереди

```
T dequeue() {  
    retry: while (true) {  
        Node first = head.get(),  
            last = tail.get(),  
            next = first.next();  
        if (first == last) {  
            if (next == null) throw new EmptyQueue();  
            // Помогаем операциям enqueue с tail  
            tail.compareAndSet(last, next);  
        } else {  
            if (head.compareAndSet(first, next))  
                return next.item;  
        }  
    }  
}
```

линеаризация →



Работа без GC, проблема ABA

- Память освобождается явно через “free” с добавлением в список свободной памяти:
 - Содержимое ячейки может меняться произвольным образом, пока на неё удерживается указатель
 - решение – дополнительные перепроверки
 - CAS может ошибочно отработать из-за проблемы ABA
 - решение – добавление версии к указателю
- Альтернативное решение – свой GC
 - Hazard Pointers как одна из специальных техник выявления указателей, которые можно безопасно освободить
 - Либо любой GC типа mark & sweep, копирующий и т.п.



DEVEXPERTS



Лекция 6

АЛГОРИТМЫ НА МАССИВАХ



Очереди/стеки на массивах

- Структуры на массивах быстрее работают на практике из-за локальности доступа к данным
- Очевидные решения не работают
 - Стек на массиве не работает
 - Очередь работает только при проходе по памяти один раз (можно развернуть очередь со списками для увеличения быстродействия)
- Неочевидные решения
 - Дек без помех (Obstruction-free Deque)
 - DCAS/CAS_n (Обновление нескольких слов одновременно)
 - Используя дескрипторы операций (универсальная конструкция)



Дек без помех

- Каждый элемента массива должен содержать элемент и версию, которые мы будем атомарно обновлять CAS-ом
- Пустые элементы будут заполнены правыми и левыми нулями (RN и LN)
- Указатели на правый и левый край будут храниться «приблизительно» и подбираться перед выполнением операций с помощью оракула (rightOracle и leftOracle)

```
// массив на MAX элементов (0..MAX-1)
{T item, int ver} a[MAX];
int left, right; // пригл. указатель на LN и RN
```



DEVEXPERTS

Оракул для поиска правого края

```
// Должен находить такое место что:  
//      a[k] == RN && a[k - 1] != RN  
// Должен корректно работать «без помех»
```

```
int rightOracle() {  
    int k = right; // только для оптимизации  
    while (a[k] != RN) k++;  
    while (a[k-1] == RN) k--;  
    right = k; // запомнили для оптимизации  
    return k;  
}
```



DEVEXPERTS

Добавление справа

```
void rightPush(T item) {  
    retry: while(true) {  
        int k = rightOracle();  
        {T item,int ver} prev = a[k-1], cur = a[k];  
        if (prev.item == RN || cur.item != RN) continue;  
        if (k == MAX-1) throw new FullDeque();  
        if (CAS(a[k-1], prev, {prev.item,prev.ver+1}) &&  
            CAS(a[k], cur, {item,cur.ver+1}))  
            return; // успешно закончили  
    }  
}
```



DEVEXPERTS

Удаление справа

```
T rightPop() {  
    retry: while(true) {  
        int k = oracleRight();  
        {T item,int ver} cur = a[k-1], next = a[k];  
        if (cur.item == RN || next.item != RN) continue;  
        if (cur.item == LN) throw new EmptyDeque();  
        if (CAS(a[k], next, {RN,next.ver+1} &&  
            CAS(a[k-1], cur, {RN,cur.ver+1}))  
            return cur.item; // успешно закончили  
    }  
}
```



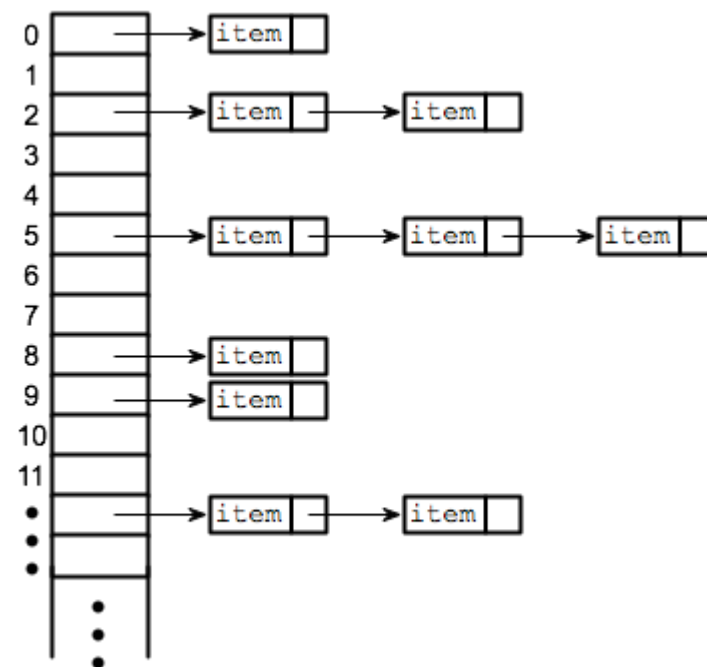

Хэш-таблицы

- Самая полезная на практике структура данных
 - $O(1)$ ожидаемое время доступа к элементу по ключу
- Два основных способа разрешения коллизий
 - Прямая адресация:
 - каждая ячейка хранит список элементов
 - Открытая адресация:
 - Быстрее на практике (нет списков, меньше ожиданий памяти)
 - Ищем в других ячейках
 - ... используя вторую хэш-функцию (double hashing)
 - ... однако на практике быстрее искать в соседних элементах (linear probing). Требуется одна хэш-функция хорошего качества



Хэш-таблицы с прямой адресацией (1)

- Наивный подход
 - Естественный параллелизм, легко делать отдельные блокировки/нарезку блокировок (lock striping) по каждой ячейке
 - Применяя алгоритмы работы со списками/множествами можно сделать реализацию без блокировок
- Изменение размера хэш-таблицы (rehash)
 - Проблема. Требуется блокировка

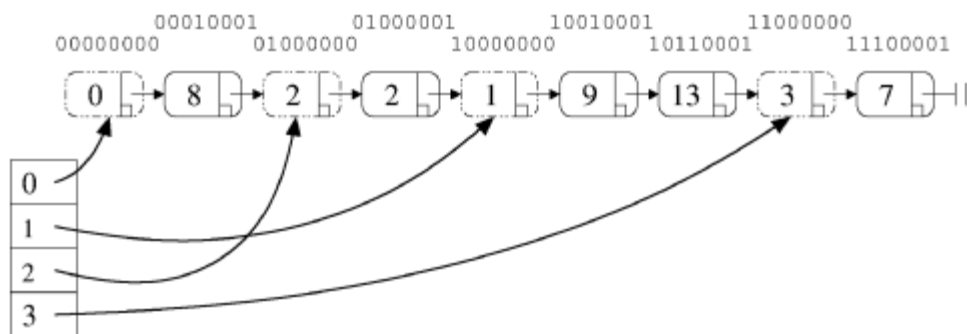


Но можно блокировать только запись, читая без ожидания



Хэш-таблицы с прямой адресацией (2)

- Упорядоченные по хэш-коду таблицы Шалева и Шавита
 - Использует один отсортированный список без блокировок
 - Таблицу ссылок (которая дает $O(1)$ поиск) можно перестраивать не теряя возможности вносить изменения в основную структуру данных (список)





Хэш-таблицы с открытой адресацией (1)

- Чтобы читать без блокировок, надо чтобы каждый ключ заняв какую-то позицию в таблице больше никогда её не освобождал и не перемещался
 - Тогда найдя эту позицию можно читать последнее значение из ячейки без ожидания
- Удаление через пометку (сброс значения)
- Чтобы добавить элементы, почистить удаленные элементы или увеличить размер нужна блокировка
 - Для всех «структурных изменений» (что влечет блокировку для всех изменений)

К	V
e	T
e	T
k_1	V_1
k_2	T
e	T
e	T
e	T

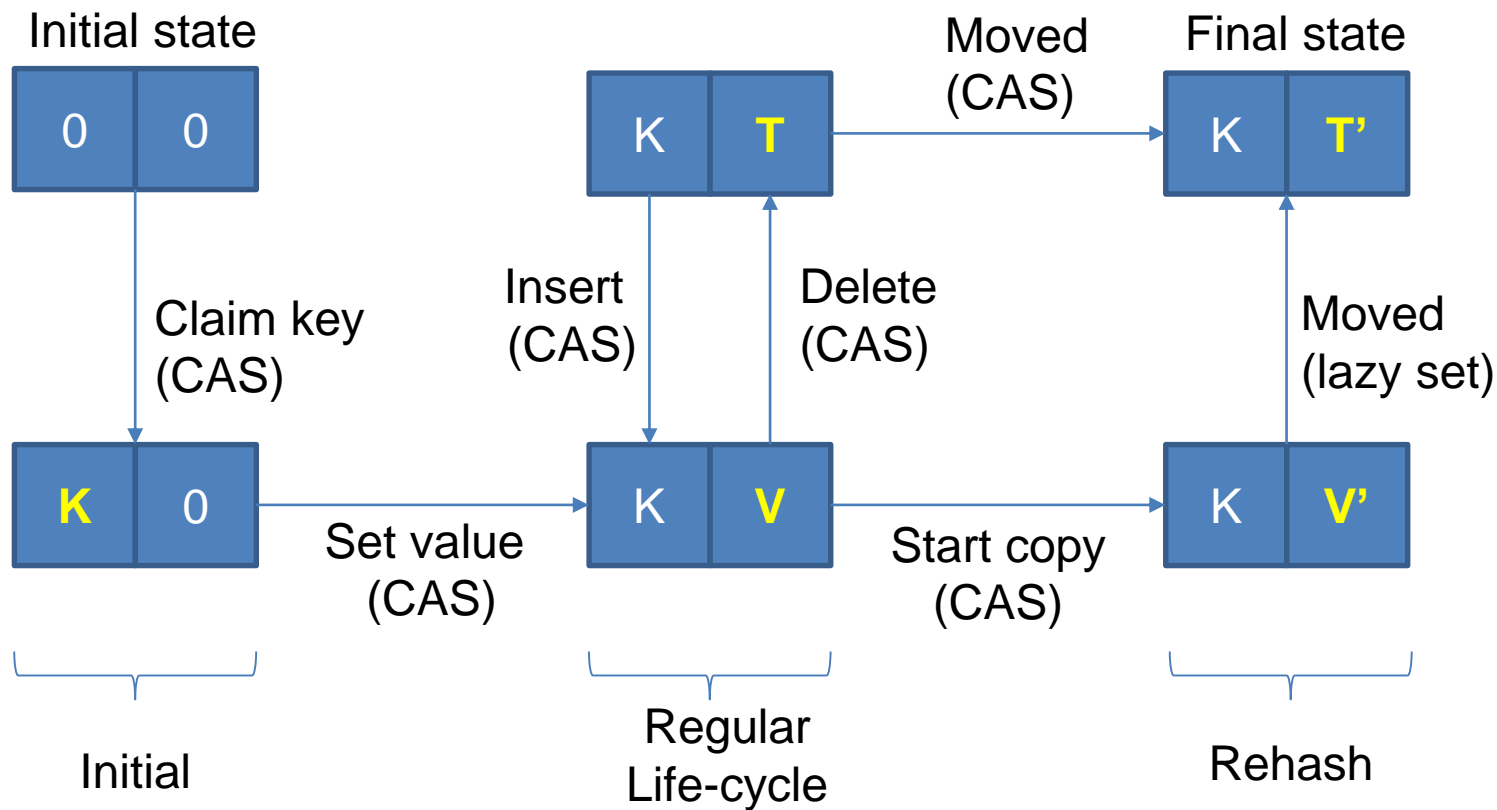


Хэш-таблицы с открытой адресацией (2)

- Но можно научиться выделять новую таблицу и отслеживать процедуру переноса ключа и значения из старой таблицы в новую таблицу
 - У каждого элемента будет свой автомат состояний и переходов между ними
 - Каждый элемент можно будет переносить по-отдельности
 - В том числе, разные элементы параллельно
 - В процессе переноса нужно помнить указатель и на новую таблицу и на старую
 - Указатель на старую таблицу можно сбросить, когда перенос завершен



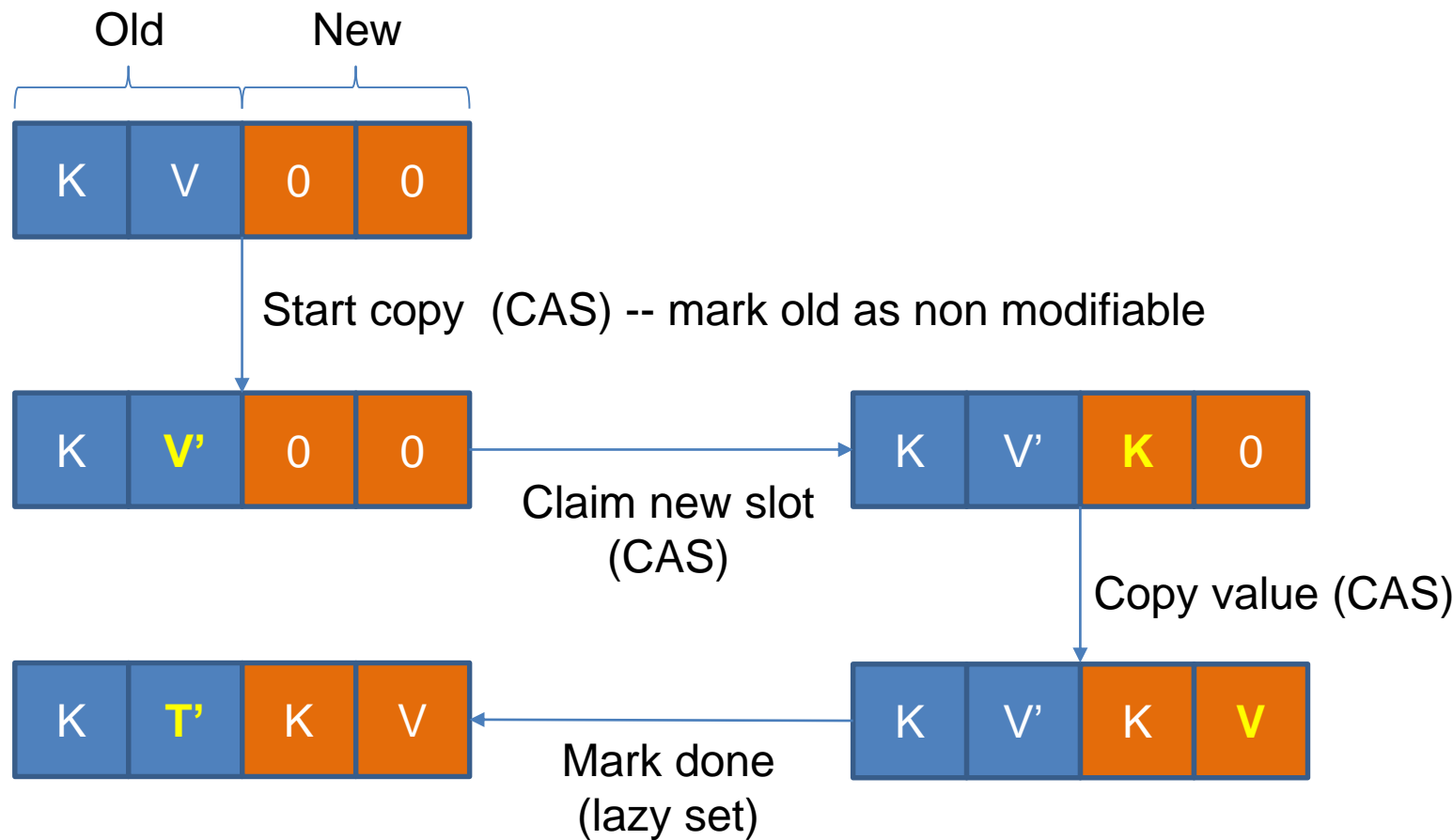
Хэш-таблицы с открытой адресацией: Жизнь ячейки



(*) Алгоритм “Efficient Almost Wait-free Parallel Accessible Dynamic Hashtables” by Gao et al
0 == null; T == del; V' == old(V)



Хэш-таблицы с открытой адресацией: Перенос в новую таблицу





Лекция 7

CASN



DEVEXPERTS

CASN: Задача

```
// ХОТИМ СОЗДАТЬ КЛАСС (на Java)
public class DataReference<T> {
    // внутреннее значение
    volatile Object value;
    // public API методы
    public T get();
    public static boolean CASN(CASEntry... entries);
}
```

```
// где
public class CASEntry<T> {
    final DataReference<T> a; // что поменять
    final T expect; // что ожидаем
    final T update; // на что заменить
    // и простой конструктор для всех 3-х полей ...
}
```



CASN: Логика работы (псевдокод)

- Хотим чтобы работало корректно (линеаризуемо) и:
 - Lock-free (без блокировок)
 - Disjoint-Access Parallel (непересекающиеся доступы ||-ны)

```
boolean CASN(CASEntry... entries) atomic {  
    for (CASEntry entry: entries)  
        if (entry.a.value != entry.expect)  
            return false;  
    for (CASEntry entry: entries)  
        entry.a.value = entry.update;  
    return true;  
}
```

(*) Алгоритм “A Practical Multi-Word Compare-And-Swap Operation” by Timothy Harris et al



DEVEXPERTS

CASN: Что есть в Java чтобы начать?

```
import java.util.concurrent.atomic.  
    AtomicReferenceFieldUpdater;  
  
// пишем в классе DataReference  
private static final  
    AtomicReferenceFieldUpdater<DataReference, Object>  
        VALUE_UPDATER =  
        AtomicReferenceFieldUpdater.newUpdater(  
            DataReference.class, Object.class, "value");  
  
boolean CAS(Object expect, Object update) {  
    return VALUE_UPDATER.compareAndSet(  
        this, expect, update);  
}
```



DEVEXPERTS

CASN: Теперь напишем getAndCAS

// псевдокод

```
Object getAndCAS(DataReference a,  
                  Object expect, Object update) atomic {  
    Object curval = a.value;  
    if (a.value == expect) a.value = update;  
    return curval;  
}
```

// реализация в классе DataReference

```
Object getAndCAS(Object expect, Object update) {  
    do {  
        Object curval = value;  
        if (curval != expect) return curval;  
    } while (!CAS(expect, update));  
    return expect;  
}
```



DEVEXPERTS

CASN: Промежуточная операция DCSS (Double-Compare Single-Set)

// псевдокод

```
Object DCSS(DataReference a1, Object expect1,  
            DataReference a2,  
            Object expect2, Object update2) atomic {  
    Object curval2 = a2.value;  
    if (a1.value == expect1 &&  
        a2.value == expect2) a2.value = update2;  
    return curval2;  
}
```

// Реализуем ограниченную (Restricted) версию – RDCSS
// a1 – только из специальной «контрольной» секции
// a2 – произвольные данные



DEVEXPERTS

CASN: RDCSSDescriptor

```
class RDCSSDescriptor {  
    private final DataReference a1;  
    private final Object expect1;  
    private final DataReference a2;  
    private final Object expect2;  
    private final Object update2;  
    // и простой конструктор для всех 5 полей ...  
}  
  
// будем вызывать операцию RDCSS так:  
new RDCSSDescriptor(  
    a1, expect1, a2, expect2, update2).invoke();  
// сейчас напомним метод invoke
```



DEVEXPERTS

CASN: Реализация RDCSS

```
// в классе RDCSSDescriptor
Object invoke() {
    Object r;
    do {
        r = a2.getAndCAS(expect2, this);
        if (r instanceof RDCSSDescriptor)
            ((RDCSSDescriptor)r).complete();
    } while (r instanceof RDCSSDescriptor);
    if (r == expect2) complete();
    return r;
}

void complete() {
    if (a1.value == expect1) a2.CAS(this, update2);
    else a2.CAS(this, expect2);
}
```



DEVEXPERTS

CASN: Как прочитать RDCSS значение?

```
// в классе DataReference
public T get() {
    while (true) {
        Object curval = value;
        if (curval instanceof RDCSSDescriptor) {
            ((RDCSSDescriptor)curval).complete();
            continue; // retry
        }
        return (T)curval; // not a descriptor
    }
}
```

```
// Этого не нужно для «контрольной» секции RDCSS (a1)
// там никогда не может оказаться RDCSSDescriptor
```




CASN: CASNDescriptor

```
class CASNDescriptor {  
    private final DataReference status =  
        new DataReference(Status.UNDECIDED);  
    private final CASEntry[] entries;  
    // и простой конструктор для entries  
}  
  
enum Status {  
    UNDECIDED,  
    SUCCEEDED,  
    FAILED  
}
```



CASN: 1-ая фаза

// в классе CASNDescriptor

```
boolean complete() {  
    if (status.value == Status.UNDECIDED) {  
        Status newStatus = Status.SUCCEEDED;  
        for (int i = 0; i < entries.length;) {  
            CASEntry entry = entries[i];  
            Object val = new RDCSSDescriptor(  
                this.status, Status.UNDECIDED,  
                entry.a, entry.expect, this).invoke();  
            if (val instanceof CASNDescriptor) {  
                if (val != this) {  
                    ((CASNDescriptor)val).complete();  
                    continue; // retry this entry }  
                } else if (val != entry.expect) {  
                    newStatus = Status.FAILED; break; }  
                i++; // go to next entry } // end for  
            this.status.CAS(Status.UNDECIDED, newStatus);  
        }  
    }  
}
```

Acquire entry {



DEVEXPERTS

CASN: 2-ая фаза

```
// в классе CASNDescriptor  
// ... продолжаем метод complete
```

```
    boolean succeeded =  
        status.value == Status.SUCCEEDED;  
    for (CASEntry entry : entries)  
Release {    entry.a.CAS(this,  
                succeeded ? entry.update : entry.expect);  
    return succeeded;  
} // конец метода complete
```



DEVEXPERTS

CASN: Как прочитать значение?

```
// в классе DataReference обновим метод
public T get() {
    while (true) {
        Object curval = value;
        if (curval instanceof RDCSSDescriptor) {
            ((RDCSSDescriptor)curval).complete();
            continue; // retry
        }
        if (curval instanceof CASNDescriptor) {
            ((CASNDescriptor)curval).complete();
            continue; // retry
        }
        return (T)curval; // not a descriptor
    }
}
```



CASN: Как гарантировать прогресс?

- Надо гарантировать одинаковый порядок обработки DataReference каждым CASN... Их надо как-то упорядочить

```
// конструктор в классе CASNDescriptor  
CASNDescriptor(CASEntry[] entries) {  
    this.entries = entries;  
    Arrays.sort(this.entries);  
}
```



DEVEXPERTS

Оптимизации CASN

- RDCSS такая сложная (с дескриптором) только если значения в ячейка могут страдать от проблемы ABA.
- Если нет, то все делается проще



Лекция 8

СЛОЖНЫЕ БЛОКИРОВКИ И STM



Анализ конфликтов

- Гонка (конфликт) данных (data race): два [несинхронизированных] доступа к одной ячейке данных, один из которых запись.
- *Матрица конфликтов* (X – конфликт):

	R	W
R		X
W	X	X



DEVEXPERTS

Пример: стек на массиве (однопоточный)

```
public class ArrayStack<T> {  
    int top;  
    T[] data;  
    // конструктор выделяет массив data  
    public int size() {  
        return top;  
    }  
  
    public void push(T item) {  
        data[top++] = item;  
    }  
  
    public T pop() {  
        return data[--top];  
    }  
}
```



Анализ конфликтов стека

	size	push	pop
size		X	X
push	X	X	X
pop	X	X	X



DEVEXPERTS

Пример: стек с грубой синхронизацией

```
public class ArrayStack<T> {  
    private int top;  
    private T[] data;  
    // конструктор выделяет массив data  
    public synchronized int size() {  
        return top;  
    }  
  
    public synchronized void push(T item) {  
        data[top++] = item;  
    }  
  
    public synchronized T pop() {  
        return data[--top];  
    }  
}
```



Пример: стек с грубой синхронизацией

- Теперь нет гонок по данным. *Почему?*

```
public synchronized int size() {  
    return top;  
}
```

```
// в Java полностью эквивалентно:
```

```
public int size() {
    synchronized (this) { // bytecode -- monitorenter
        return top;
    }                       // bytecode -- monitorexit
}
```



Пример: стек с грубой синхронизацией 2

```
import java.util.concurrent.locks.*;
```

```
// В классе ArrayStack в дополнение к данным:  
private final Lock lock = new ReentrantLock();
```

```
public int size() {  
    try {  
        lock.lock();  
        return top;  
    } finally {  
        lock.unlock();  
    }  
}
```



Пример: стек с грубой синхронизацией 2

// В классе ArrayStack продолжаем:

```
public void push(T item) {  
    lock.lock();  
    try {  
        data[top++] = item;  
    } finally {  
        lock.unlock();  
    }  
}
```

// аналогично pop



Матрица совместимости блокировок с грубой синхронизацией

	size	push	pop
size	X лишнее	X	X
push	X	X	X
pop	X	X	X



Read-write locks (блокировка чтения-записи)

- Это специальные локи, со следующей *матрицей совместимости* (X – несовместимые блокировки)
 - Read aka *Shared* Lock
 - Write aka *Exclusive* Lock
- Они идеально подходят для грубой защиты структур данных в которых есть конфликты (гонки) по данным.

	R	W
R		X
W	X	X



DEVEXPERTS

Пример: стек с грубой синхронизацией 3

```
// В классе ArrayStack в дополнение к данным:  
private final ReadWriteLock lock =  
    new ReentrantReadWriteLock();  
  
public int size() {  
    lock.readLock().lock();  
    try {  
        return top;  
    } finally {  
        lock.readLock().unlock();  
    }  
}
```



DEVEXPERTS

Пример: стек с грубой синхронизацией 3

// В классе ArrayStack продолжаем:

```
Public void push(T item) {  
    lock.writeLock().lock();  
    try {  
        data[top++] = item;  
    } finally {  
        lock.writeLock().unlock();  
    }  
}
```

// аналогично pop



Матрица совместимости с блокировками чтения-записи с грубой синхронизацией

	size	push	pop
size		X	X
push	X	X	X
pop	X	X	X

Полностью повторяет матрицу конфликтов. Лучше сделать нельзя.



DEVEXPERTS

Стек на массивах используя CASN

// в классе ArrayStack:

```
private final DataReference<Integer> top =  
    new DataReference<Integer>(0);  
private final DataReference<T>[] data;
```

// конструктор создает и инициализирует data

```
public int size() {  
    return top.get();  
}
```



DEVEXPERTS

Стек на массивах используя CASN

// в классе ArrayStack:

```
public void push(T newItem) {  
    Integer curtop;  
    do {  
        curtop = top.get();  
    } while (!DataReference.CASN(  
        new CASEntry<Integer>(top, curtop, curtop + 1),  
        new CASEntry<T>(data[curtop], null, newItem)));  
}
```

// похожим образом pop

// есть ли здесь проблема АВА?



DEVEXPERTS

Обзор методов создания многопоточных объектов (уже рассмотренных)

Или как сделать *линеаризуемый* многопоточный объект?

- Блокировка (aka синхронизация)
 - Грубая, тонкая, оптимистичная, ленивая
 - Read-Write
- Без блокировки
 - Универсальная конструкция (copy-on-write + CAS, частичное копирование + CAS)
 - CASN
 - Алгоритмы специфичные для структуры данных (пример: список-множество, очередь без блокировки Майкла-Скота, дэк без помех и т.п.)



Недостатки блокировки

- В системе нет прогресса, пока объект заблокирован
 - Инверсия приоритетов
- Требуются дополнительные переключения контекста чтобы дать закончить работу блокирующему потоку
 - Может съесть существенную долю CPU времени системы
- Минимальный параллелизм работы
 - Но чем меньше блокировки, тем больше параллелизм
- Взаимные блокировки (deadlocks)



DEVEXPERTS

Общая проблема: построение составных объектов (composability)

// Небезопасный объект!

```
public class Employees {  
    Set working = new ConcurrentSet(); // thread-safe  
    Set vacating = new ConcurrentSet(); // thread-safe  
  
    public boolean contains(Employee e) {  
        return working.contains(e) ||  
            vacating.contains(e);  
    }  
  
    public void startVacation(Employee e) {  
        working.remove(e);  
        vacating.add(e);  
    }  
}
```




Решение: Software Transactional Memory (STM)

- Классические транзакции:
 - **A**tomicity
 - **C**onsistency
 - **I**solation
 - ~~**D**~~urability



DEVEXPERTS

Поддержка на уровне языка

// если бы можно было бы написать так...

```
public boolean contains(Employee e) {  
    atomic {  
        return working.contains(e) ||  
            vacating.contains(e);  
    }  
}
```

```
public void startVacation(Employee e) {  
    atomic {  
        working.remove(e);  
        vacating.add(e);  
    }  
}
```



DEVEXPERTS



Без поддержки на уровне языка

// придется написать так...

```
public boolean contains(final Employee e) {  
    return Transaction.atomic(() -> {  
        return working.contains(e) ||  
            vacating.contains(e);  
    });  
}
```

// и т.п.



DEVEXPERTS



Transaction

```
public class Transaction {  
    public static <R> R atomic(AtomicBlock<R> call) {  
        for (;;) {  
            Transaction t = beginTransaction();  
            try {  
                R result = call.call();  
                if (t.commit())  
                    return result;  
            } catch (RuntimeException|Error e) {  
                t.rollback();  
                throw e;  
            }  
        }  
    }  
}  
// далее идет метод begin и т.п.
```



DEVEXPERTS

Реализация транзакций

```
public class TVar<T> {  
    private T value;  
  
    public T get() {  
        // что-то здесь...  
    }  
  
    public void set(T value) {  
        // что-то тут...  
    }  
}
```

// Хотим чтобы в рамках транзакции при использовании операций чтения-записи всегда получалось линейризуемое исполнение.



Реализация транзакций с блокировками

- Двухфазная блокировка (2PL = 2 Phase Locking)
 - Все конфликтующие операции защищаются локами, исключающими конфликты.
 - В начале транзакции (первая фаза) локи накапливаются.
 - В конце транзакции (вторая фаза) локи освобождаются
- **Основная теорема:** Любое допустимое исполнение такой системы будет линеаризуемо.



DEVEXPERTS

Реализация транзакций с блокировками

```
// В классе TVar добавляем...
private final ReadWriteLock lock =
    new ReentrantReadWriteLock();

public T get() {
    lock.readLock().lock();
    Transaction.currentTransaction().
        addLock(lock.readLock());
    return value;
}

public void set(T value) {
    lock.writeLock().lock();
    Transaction.currentTransaction().
        addLock(lock.writeLock());
    this.value = value;
}
```



Реализация транзакций с блокировками

// В классе Transaction добавляем...

```
private static final ThreadLocal<Transaction> CURRENT=  
    new ThreadLocal<Transaction>();  
private final List<Lock> locks =  
    new ArrayList<Lock>();  
public static Transaction beginTransaction() {  
    Transaction t = new Transaction();  
    CURRENT.set(t);  
    return t;  
}  
  
public static Transaction currentTransaction() {  
    return CURRENT.get();  
}  
  
void addLock(Lock lock) { locks.add(lock); }
```




DEVEXPERTS

commit с блокировками

// В классе Transaction добавляем...

```
public boolean commit() {  
    for (Lock lock : locks)  
        lock.unlock();  
    return true;  
}
```

// а вот rollback сделать не так очевидно...

// В классе TVar добавляем

```
private static final Object UNDEFINED = new Object();  
private Object oldValue = UNDEFINED;
```



DEVEXPERTS

rollback с блокировками 1

```
// В классе TVar добавляем
public void set(T value) {
    if (oldValue == UNDEFINED) {
        lock.writeLock().lock();
        this.oldValue = this.value;
        Transaction.currentTransaction().
            addWrite(this);
    }
    this.value = value;
}

void rollback() {
    value = (T)oldValue;
    oldValue = UNDEFINED;
    lock.writeLock().unlock();
}
```



DEVEXPERTS

rollback с блокировками 2

// В классе Transaction добавляем...

```
private final Set<TVar<?>> writes =  
                                new HashSet<TVar<?>>();
```

```
public void addWrite(TVar<?> var) {  
    writes.add(var);  
}
```

```
public void rollback() {  
    for (TVar<?> var : writes)  
        var.rollback();  
    for (Lock lock : locks)  
        lock.unlock();  
}
```



Реализация транзакций с блокировками

- Проблемы – прогресс
 - Взаимные блокировки (deadlocks)
- Нужно написать
 - Предотвращение взаимных блокировок (deadlock avoidance)
 - Определение взаимных блокировок (deadlock detection)
 - Например, по истечению времени ожидания лока
 - Устранение взаимных блокировок (deadlock resolution)
 - Откатывая и начиная снова одну из заблокированных транзакций



DEVEXPERTS

Реализация транзакций без блокировок

```
public class Transaction {  
    private static final int ACTIVE = 0;  
    private static final int COMMITTED = 1;  
    private static final int ABORTED = -1;  
    private final AtomicInteger state =  
        new AtomicInteger(ACTIVE);  
    public boolean isCommitted() {  
        return state.get() == COMMITTED;  
    }  
  
    public boolean commit() {  
        return state.compareAndSet(ACTIVE, COMMITTED);  
    }  
  
    public void rollback() {  
        state.compareAndSet(ACTIVE, ABORTED);  
    }  
}
```



DEVEXPERTS

Хранение значений без блокировок

// Реализуем вспомогательный объект

```
class VarHolder<T> {  
    final Transaction owner;  
    final Object value;  
    Object newValue; // updated by owner
```

```
    VarHolder(Transaction owner, Object value) {  
        this.owner = owner;  
        this.value = value;  
        this.newValue = value;  
    }
```

// Текущее значение зависит от состояния транзакции

```
T current() {  
    return owner.isCommitted() ?  
        (T)newValue : (T)value;  
}
```



DEVEXPERTS

«Открытие» переменной без блокировок

```
public class TVar<T> {  
    private AtomicReference<VarHolder<T>> holder = ...  
  
    // Будем «открывать» переменную перед любым доступом  
  
    VarHolder<T> open() {  
        Transaction tx = Transaction.current();  
        VarHolder<T> old, upd;  
        do {  
            old = holder.get();  
            if (old.owner == tx) return old;  
            old.owner.rollback(); // если активен!  
            upd = new VarHolder<T>(tx, old.current());  
        } while (!holder.compareAndSet(old, upd));  
        return upd;  
    }  
}
```



Доступ к переменной без блокировок

```
// В классе TVar<T>
```

```
public T get() {  
    return (T)open().newValue;  
}
```

```
public void set(T value) {  
    open().newValue = value;  
}
```




Реализация транзакций без блокировок

- Это реализация без помех (obstruction-free)
 - Разные потоки могут бесконечно долго друг другу мешать закончить транзакцию без прогресса
 - Но если активен только один поток, то прогресс гарантирован
 - Сравните с версией с блокировками
 - На практике, нужно управлять конфликтами (contention manager), чтобы отслеживать конфликты и увеличивать прогресс
- Проблема
 - Даже читающие транзакции мешают друг другу
 - Предыдущий алгоритм с блокировками был здесь лучше



Параллельное чтение без блокировок

```
// В классе TVar не будем открывать при чтении  
public T get() {  
    return holder.get().current();  
}
```

- Проблема: Значение может поменяться в процессе работы транзакции (non-repeatable read)
 - А значит, нет линейризуемости транзакций
- Решения:
 - Перепроверка корректности во время завершения транзакции
 - Многоверсионный контроль корректности (MVCC – Multiversion Concurrency Control)



DEVEXPERTS

Перепроверка корректности во время завершения транзакции - read

```
// В классе Transaction запомним прочитанные значения
private final Map<TVar<?>, Object> reads =
    new HashMap<TVar<?>, Object>();

// Метод для чтения значения в транзакции
<T> T read(TVar<T> var) {
    VarHolder<T> holder = var.holder.get();
    T cur = holder.current();
    if (holder.owner == this) return cur;
    if (reads.containsKey(var) && reads.get(var) != cur)
        rollback(); // кто-то поменял в процессе...
    if (state.get() == ABORTED)
        throw new Rollback();
    reads.put(var, cur);
    return cur;
}
```



DEVEXPERTS

Перепроверка корректности во время завершения транзакции - commit

// В классе Transaction будем открывать прочитанные
// переменные перед самым завершением транзакции

```
public boolean commit() {  
    for (Map.Entry<TVar<?>, Object> entry :  
                                                reads.entrySet()) {  
        VarHolder<?> cur = entry.getKey().open();  
        Object expect = entry.getValue();  
        if (cur.value != expect) {  
            rollback();  
            return false;  
        }  
    }  
    return state.compareAndSet(ACTIVE, COMMITTED);  
}
```



DEVEXPERTS

Перепроверка корректности во время завершения транзакции - get

```
// В классе TVar будем использовать read  
public T get() {  
    return Transaction.current().read(this);  
}
```

- Конфликт по чтению остался только на время commit
 - Можно использовать глобальную блокировку на короткое время операции commit
 - Алгоритм теряет свойство «без блокировок»
 - Алгоритм перестанет быть DAP (disjoint access parallel)
 - Можно не открывать прочитанные переменные в commit, а написать алгоритм завершения без ожидания, аналогичный DCSS (double-compare single-swap), чтобы commit происходил только если все прочитанные и не измененные переменные имеют свои старые значения