Параллельное программирование

Роман Елизаров, 2009

elizarov@devexperts.com

Взаимное исключение

- Взаимное исключение (Mutual Exclusion)
- Отсутствие блокировки (Freedom from Deadlock)
- Отсутствие голодания (Freedom from Starvation)

```
run() {
  int i = getCurrentThreadID();
  while (true) {
    nonCriticalSection();
    lock(i);
    criticalSection();
    unlock(i);
  }
}
```

Взаимное исключение, попытка 1

```
boolean want[2];

lock(int i) {
  want[i] = true;
  while (want[1 – i]); // wait
}

unlock(int i) {
  want[i] = false;
}
```

Взаимное исключение, попытка 2

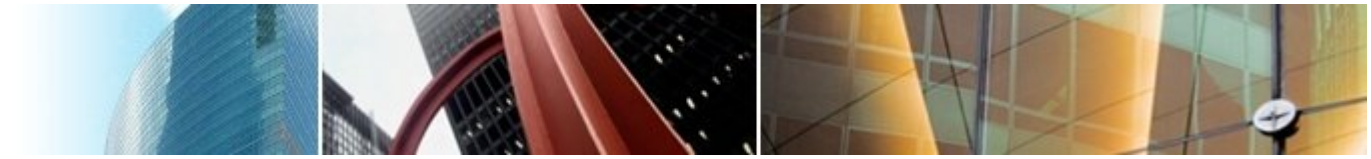
```
int victim;

lock(int i) {
  victim = i;
  while (victim == i); // wait
}

unlock(int i) {
}
```

Взаимное исключение, алгоритм Петерсона

```
boolean want[2];
int victim;

lock(int i) {
  want[i] = true;
  victim = i;
  while (want[1 – i] && victim == i); // wait
}

unlock(int i) {
  want[i] = false;
}
```

Взаимное исключение, алгоритм Петерсона для N потоков

```
int level[N];
int victim[N];

lock(int i) {
  for (int j = 1; j < N; j++) {
    level[i] = j;
    victim[j] = i;
    while exists k != i :
      level[k] >= j && victim[j] == i; // wait
}

unlock(int i) {
  level[i] = 0;
}
```



Честность

- Отсутствие голодания
- Линейное ожидание, квадратичное ожидание и т.п.
- Первым пришел, первым обслужен (FCFS – First Come First Served)

```
lock() {
    DoorwaySection(); // wait-free code
    WaitingSection();
}
```

Взаимное исключение, алгоритм Лампорта (алгоритм булочника – вариант 1)

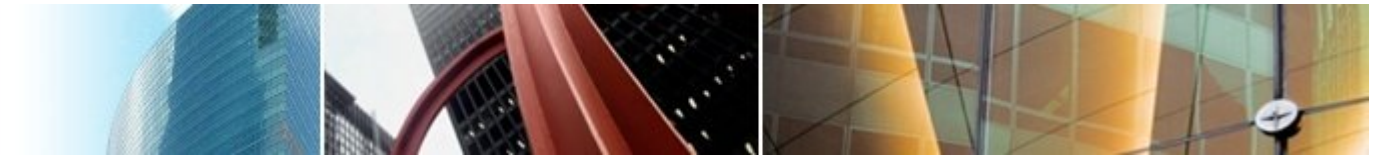
```
boolean want[N]; // init with false
Label label[N];  // init with 0

lock(int i) {
  want[i] = true;
  label[i] = max(label[0], …, label[N-1]) + 1;
  while exists k != i :
    want[k] && (label[k], k) < (label[i], i);
}

unlock(int i) {
  want[i] = false;
}
```

Взаимное исключение, алгоритм Лампорта (алгоритм булочника – вариант 2)

```
boolean choosing[N]; // init with false
Label label[N];      // init with inf

lock(int i) {
  choosing[i] = true;
  label[i] = max(label[0], …, label[N-1]) + 1;
  choosing[i] = false;
  while exists k != i :
    choosing[k] || (label[k], k) < (label[i], i);
}

unlock(int i) {
  label[i] = inf;
}
```

Разделяемые объекты

- Корректность реализации объекта
  - Тихая согласованность (Quiescent consistency)
  - Последовательная согласованность (Sequential consistency)
  - Линеаризуемость (Linearizability)
- Прогресс
  - Без помех (Obstruction-free)
  - Без блокировок (Lock-free)
  - Без ожидания (Wait-free)



Регистры

- Разделяемые регистры – базовый объект для общения потоков между собой

```
interface Register<T> {
  T read();
  void write(T val);
}
```

Классификация регистров

- Безопасные (safe), регулярные (regular), атомарные (atomic)
- Один читатель, много читателей (SR, MR)
- Один писатель, много писателей (SW, MW)
- Булевские значение, множественные значения

- Самый примитивный регистр – Safe SRSW Boolean register
- Самый сложный регистр – Atomic MRMW M-Valued register

Построение регистров

- Будем строить более сложные регистры из более простых без ожиданий (wait-free образом).
  - Safe SRSW Boolean register
  - Regular SRSW Boolean register
  - Regular SRSW M-Valued register
  - Atomic SRSW M-Valued register
  - Atomic MRSW M-Valued register
  - Atomic MRMW M-Valued register

Атомарный снимок состояния N регистров

- Набор SW атомарных регистров (по регистру на поток)
- Любой поток может вызвать scan() чтобы получить снимок состояния всех регистров
- Методы должны быть атомарными (линеаризуемыми)

```
interface Snapshot<T> {
  void update(int i, T val);
  T[] scan();
}
```

Атомарный снимок состояния N регистров, без блокировок (lock free)

```
// каждый регистр хранит версию “version”
(T val, Label version) register[N];

void update(int i, T val) { // wait-free
  register[i] = (val, register[i].version+1);
}

T[] scan() { // obstruction-free
  (T, Label)[] old = copyOf(register); // with loop
  while (true) {
    (T, Label)[] cur = copyOf(register);
    if (equal(old, cur)) return cur.val;
                    else old = cur;
  }
}
```

Атомарный снимок состояния N регистров, без ожидания (wait-free) – update

```
// каждый регистр так же хранит копию снимка “snap”
(T val, Label version, T[] snap) register[N];

void update(int i, T val) { // wait-free
  T[] snap = scan();
  register[i] = (val, register[i].version+1, snap);
}
```

Атомарный снимок состояния N регистров, без ожидания (wait-free) – scan

```
T[] scan() { // wait-free, O(N^2) time
  (T, Label, T[])[] old = copyOf(register);
  boolean updated[N];
  loop: while (true) {
    (T, Label, T[])[] cur = copyOf(register);
    for (int j = 0; j < N; j++)
      if (cur[j].version != old[j].version)
        if (updated[j]) return cur[j].snap;
        else {
          updated[j] = true;
          old = cur; continue loop;
        }
    return cur.val;
  }
}
```

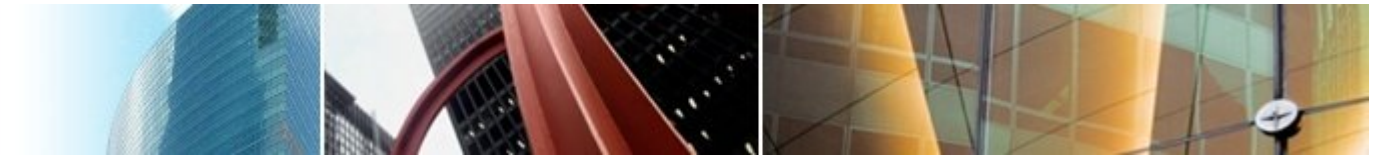
Консенсус

- 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

```
interface Consensus<T> {
  T decide(T val);
}
```

Консенсусное число

- Если с помощью класса объектов C можно реализовать консенсусный протокол без ожидания (wait-free) 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 придти к консенсусу без ожидания (докажем от противного) для для 2-х возможных значений при T = {0, 1}
  - С ожиданием задача решается очевидно (с помощью любого алгоритма взаимного исключения).



Определения и леммы для любых классов объектов

- Определения и концепции:
  - Рассматриваем дерево состояния, листья – конечные состояния помеченные 0 или 1 (в зависимости от значения консенсуса).
  - x-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**: Существует начальное бивалентное состояние.
- **ЛЕММА**: Существует критическое состояние.

Доказательство для атомарных регистров

- Рассмотрим возможные пары операций в критическом состоянии:
  - Операции над разными регистрами – коммутируют.
  - Два чтения – коммутируют.
  - любая операция + Запись – состояние пишущего потока не зависит от порядка операций.

Read-Modify-Write регистры

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

```
class RMWRegister<T> {
  private T val;
  T getAndF(args…) atomic {
    T old = val;
    val = F(T, args);
    return old;
  }
}
```

Нетривиальные RMW регистры

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

```
T proposed[2];
RMWRegister rmw; // начальное значение v0

T decide(T val) {
  int i = myThreadId(); // i = 0,1 – номер потока
  proposed[i] = val;
  if (rmw.getAndF() == v0) return proposed[i];
                      else return proposed[1 – i];
}
```

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 состоянием при изменения порядка выполнения коммутирующих или перезаписывающих операций в критическом состоянии.

Универсальные объекты

- Объект с консенсусный числом бесконечность называется универсальным объектом.
  - По определению, с помощью него можно реализовать консенсусный протокол для любого числа потоков.
- Пример: compareAndSet aka testAndSet (возвращает boolean), compareAndExchange (возвращает старое значение – RMW)

```
private T x;
boolean compareAndSet(T val, T expected) atomic {
  if (x == expected) { x = val; return true; }
                else return false;
}
```

compareAndSet (CAS) и консенсус

```
// реализация консенсусного протокола через CAS+GET
T decide(T val) {
  if (compareAndSet(val, INITIAL))
        return val;
    else return get();
}

// реализация консенсусного протокола через CMPXCHG
T decide(T val) {
  T old = compareAndExchange(val, INITIAL);
  if (old == INITIAL)
        return val;
    else return old;
}
```

Универсальность консенсуса

- **ТЕОРЕМА**: Любой последовательный объект можно реализовать без ожидания (wait-free) для N потоков используя консенсусный протокол для N объектов.
  - Следствие1: С помощью любого класса объектов с консенсусным числом N можно реализовать любой объект с консенсусным числом <= N.
  - Следствие2: С помощью универсального объекта можно реализовать любой объект.

Иерархия объектов

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

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 удается реализовать все операции без ожидания (wait-free). Часто приходится идти на компромиссные решения.
- **ВНИМАНИЕ**: Пока пишем псевдокод. Доведение его до реального кода будем обсуждать отдельно.

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
- Проще всего для списочных структур данных (с них и начнем), хотя на практике массивы работают существенно быстрей

Пример: Множество на основе односвязного списка

- **ИНВАРИАНТ**: node.key < node.next.key

```
class Node {
    int key;
    T item;
    Node next;
}

// Пустой список состоит их 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 Mutex lock.

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(); }
}
```

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 = new Node(key, item);
            node.next = curr; pred.next = node;
            return true;
        }
    } finally { lock.unlock(); }
}
```

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 утерять инвариант pred.next == curr).

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 = new Node(key, item);
        node.next = curr; pred.next = node;
        return true;
    }
} 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
- Имеет смысл только если обход структуры дешев и быстр, а обход с синхронизацией медленный и дорогой

Оптимистичная синхронизация: поиск

```
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;
        return curr.key == key;
    } finally { curr.unlock(); pred.unlock(); }
}
```

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;
    }
    return false;
}
```

Оптимистичная 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 = 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;
    }
    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(); }
}
```

Ленивая синхронизация

- Добавляем в Node поле boolean marked.
- Удаление в 2 фазы:
  - node.marked = true; // Логическое удаление
  - Физическое удаление из списка
- Инвариант: Все непомеченные (неудаленные) 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.
  - Поиск элемента в списке можно делать без ожидания (wait-free)

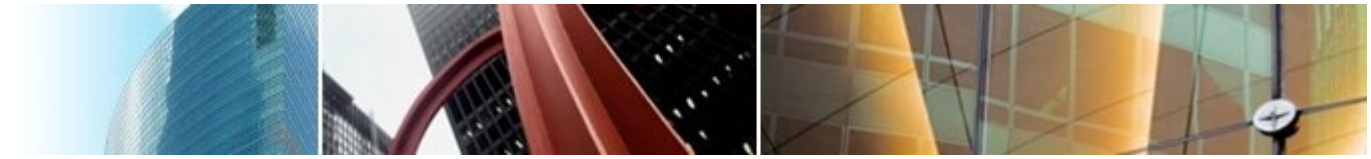
Ленивая синхронизация: удаление

```
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;
            pred.next = curr.next;
            return true;
        } else return false;
    } finally { curr.unlock(); pred.unlock(); }
}
```

точка линеаризации

Ленивая синхронизация: валидация

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



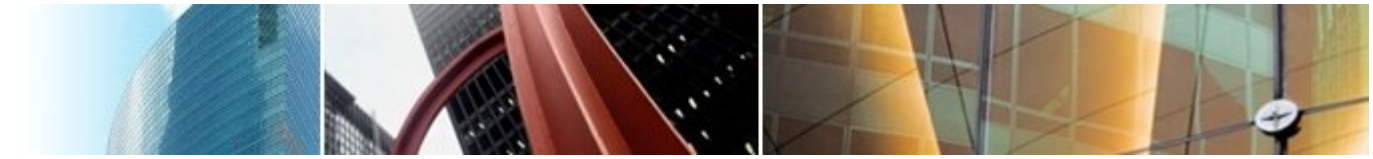
Ленивая синхронизация: поиск (wait-free!)

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

точка линеаризации
успешного поиска

Неблокирующая синхронизация

- Простое использование Compare-And-Set (CAS) не помогает – удаления двух соседних элементов будут конфликтовать.
- Объединим next и marked в одну переменную {next,marked}, которую будем атомарно менять используя CAS
  - Каждая операция модификации будет выполнятся 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)



Неблокирующая синхронизация: добавление

```
retry: while (true) {
    Node pred, curr; {pred,curr} = find(key);
    if (curr.key == key) return false; else {
        Node = new Node(key, item);
        node.{next,marked} = {curr,false};
        if (CAS(pred.{next,marked}, {curr,false},
                                    {next,false})
            return true;
    }
}
```

линеаризация

Неблокирующая синхронизация: поиск окна

```
{Node,Node} find(int key) {
 retry: while(true) {
     Node pred = head, curr = pred.next, succ;
     while (true) {
         {succ,boolean cmk} = curr.{next,marked};
         if (cmk) { // Если curr логически удален
             if (!CAS(pred.{next,marked},
                      {curr,false}, {succ,false}))
                 continue retry;
             curr = succ;
         } else {
             if (curr.key >= key) return {pred,curr};
             pred = curr; curr = succ;
         }
     }
 }
```

Неблокирующая синхронизация: удаление

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

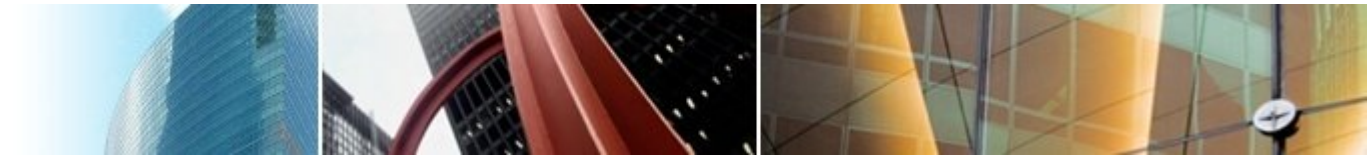
Атомарный счетчик

```
int counter;

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

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.
- Частный случай этого подхода: LIFO стек

```
class Node {
    T item;
    Node next;
}
// Пустой стек это указатель на null
Node top = null;
```

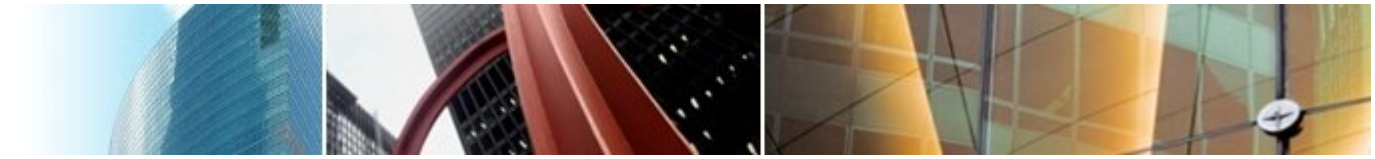


Операции с LIFO стеком

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

линеаризация

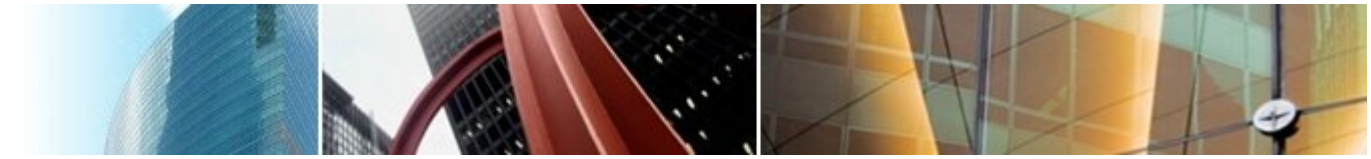
линеаризация



FIFO очередь без блокировок (lock-free)

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

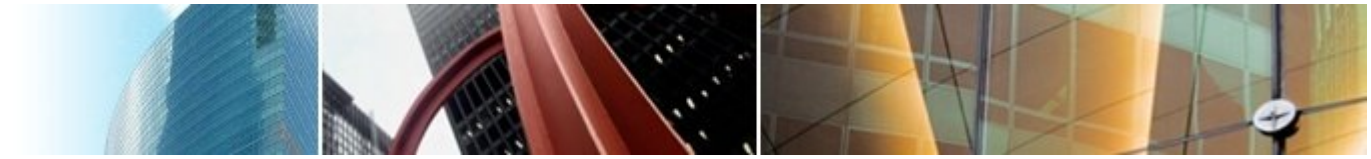
```
class Node {
    T item;
    Node next;
}

// Пустой список состоит их одного элемента-заглушки
Node head = new Node(null);
Node tail = head;
```



Добавление в очередь

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

линеаризация



Удаление из очереди

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

Очереди/стеки на массивах

- Структуры на массивах быстрей 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/CASn (Обновление нескольких слов одновременно)
  - Используя дескрипторы операций (универсальная конструкция)

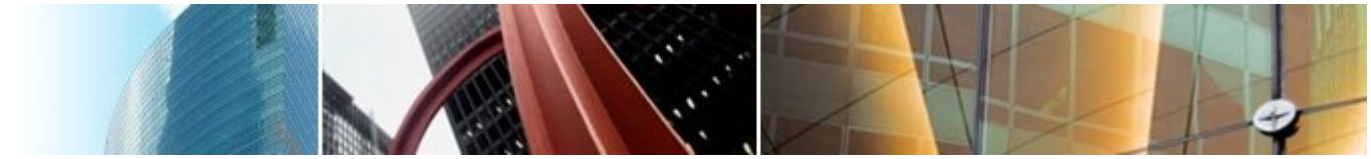
Дек без помех

- 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; // прибл. указатель на LN и RN
```

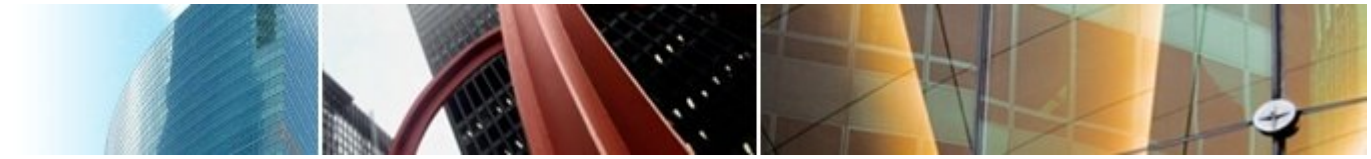
Оракул для поиска правого края

```
// 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;
}
```



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
}
```



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

- Два основных способа разрешения коллизий
  - Прямая адресация: каждая ячейка хранит список элементов
    - Естественный параллелизм, легко делать раздельные блокировки/нарезку блокировок (lock striping)
    - Применяя алгоритмы работы со списками/множествами можно сделать реализацию без блокировок
  - Открытая адресация: ищем в других ячейках
    - На практике быстрей искать в соседних элементах, но требует хэш-функции хорошего качества
    - Так же возможна реализация без блокировок (занимаем ячейку через CAS), но требует специальной техники удаления
- Изменение размера хэш-таблицы (rehash)