

# Конспект по курсу Параллельное программирование<sup>1</sup>

Александра Лисицына<sup>2</sup>

12 января 2019 г.

<sup>1</sup>Читаемый Романом Елизаровым Никитой Ковалем в 2018-2019 годах  
<sup>2</sup>Студентка группы М3334

# Содержание

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Закон Мура . . . . .	4
1.2	Закон Амдала . . . . .	5
1.3	Разные виды параллелизма . . . . .	5
1.3.1	Параллелизм на уровне инструкций (ILP) . . . . .	5
1.4	Операционные системы . . . . .	7
1.5	Основные понятия в современных ОС . . . . .	7
1.6	Формализм . . . . .	8
1.6.1	Модели программирования . . . . .	8
1.6.2	Общие объекты . . . . .	8
1.6.3	Общие переменные . . . . .	8
1.6.4	Свойства многопоточных программ . . . . .	9
1.6.5	Моделирование многопоточного исполнения . . . . .	9
<b>2</b>	<b>Lock-free Treiber Stack and Michael-Scott Queue</b>	<b>10</b>
<b>3</b>	<b>Определения и формализм</b>	<b>10</b>
3.1	Физическая реальность . . . . .	10
3.2	Модель «произошло до» (happens before) . . . . .	11
3.3	Модель глобального времени . . . . .	11
3.4	Обсуждение глобального времени . . . . .	11
3.5	«Произошло до» на практике . . . . .	12
3.6	Свойства исполнений над общими объектами . . . . .	12
3.6.1	Операции над общими объектами . . . . .	12
3.6.2	Последовательное исполнение . . . . .	12
3.6.3	Конфликты и гонки данных (data race) . . . . .	12
3.6.4	Правильное исполнение . . . . .	13
3.6.5	Правильное исполнение и нотация . . . . .	13
3.6.6	Последовательная спецификация объекта . . . . .	13
3.6.7	Допустимое последовательное исполнение . . . . .	13
3.6.8	Условия согласованности (корректности) . . . . .	14
3.7	Лианеризуемость . . . . .	15
3.7.1	Применительно к Java . . . . .	15
<b>4</b>	<b>Построение атомарных объектов и блокировки</b>	<b>15</b>
4.1	Сложные и составные операции . . . . .	15
4.1.1	Формализация сложных операций . . . . .	15
<b>5</b>	<b>Consensus</b>	<b>16</b>
5.1	Задача о консенсусе . . . . .	16
5.2	Консенсусное число . . . . .	16
5.3	Модель . . . . .	16
5.4	Read–Modify–Write регистры . . . . .	16
5.5	Универсальность консенсуса . . . . .	16
<b>6</b>	<b>Алгоритмы без блокировок: Построение на регистрах</b>	<b>16</b>
6.1	Безусловные условия прогресса . . . . .	16
<b>7</b>	<b>Практические построения на списках</b>	<b>17</b>
7.1	MНожество на односвязном списке . . . . .	17
7.1.1	Алгоритм . . . . .	17
7.1.2	Псевдокод . . . . .	17
7.1.3	Проблема . . . . .	18
7.2	Грубая синхронизация . . . . .	18
7.3	Тонкая блокировка . . . . .	18
7.3.1	Корректность . . . . .	18
7.4	Оптимистичная синхронизация . . . . .	18
7.4.1	Алгоритм абстрактной операции . . . . .	18
7.4.2	Проверка, что узел не удалён . . . . .	19
7.4.3	Валидация окна . . . . .	19

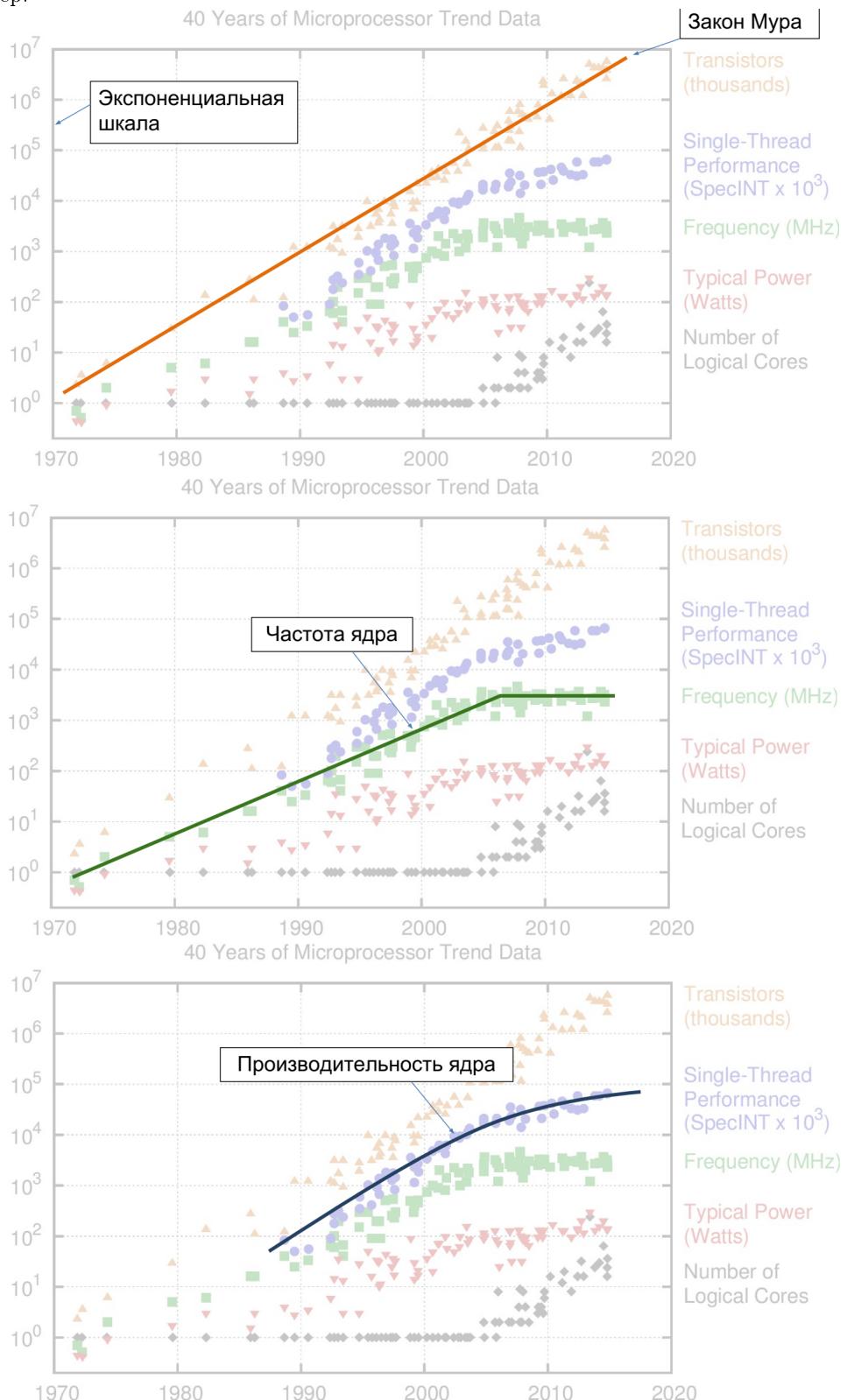
7.4.4	Корректность . . . . .	19
7.5	Ленивая синхронизация . . . . .	19
7.5.1	Идея ленивого удаления . . . . .	19
7.5.2	Псевдокод . . . . .	19
7.6	Неблокирующая синхронизация . . . . .	19
7.6.1	Неблокирующий поиск . . . . .	19
7.6.2	Неблокирующая модификация . . . . .	20
<b>8</b>	<b>Lock-free хеш таблица с открытой адресацией</b>	<b>20</b>
8.1	ConcurrentHashMap . . . . .	20
8.2	Skip List . . . . .	21
8.3	Потенциальные проблемы . . . . .	21
8.4	NonBlockingHashMap . . . . .	21
8.4.1	Пара деталей . . . . .	22
8.5	MRSW хеш таблица (рисунок 26) . . . . .	22
8.6	MRMW хеш таблица . . . . .	22
8.6.1	Кооперация при переносе . . . . .	22
<b>9</b>	<b>CASN</b>	<b>23</b>
9.1	Списки против массивов . . . . .	23
9.2	CASn . . . . .	23
9.2.1	DCSS . . . . .	24
9.3	Наблюдения и замечания . . . . .	26
9.4	Подход к реализации . . . . .	27
9.5	DCSS Mod . . . . .	27
9.6	CASN . . . . .	28
9.6.1	Псевдокод CAS2 . . . . .	28
<b>10</b>	<b>Мониторы и ожидание</b>	<b>29</b>
10.1	Объекты как функции . . . . .	29
10.2	Очередь ограниченного размера с ожиданием . . . . .	30
10.3	Лианеризуемость операций с ожиданием . . . . .	31
10.4	Реализация через монитор . . . . .	32
10.5	Монитор в Java . . . . .	32
10.6	Циклическая очередь с ожиданием на массиве . . . . .	33
10.6.1	size() . . . . .	33
10.6.2	poll() . . . . .	33
10.6.3	take() . . . . .	33
10.6.4	offer() . . . . .	33
10.6.5	put() . . . . .	33
10.7	notify() vs notifyAll() . . . . .	34
10.8	ReentrantLock . . . . .	34
10.8.1	Проблематичный сценарий . . . . .	34
10.9	Подробней про interrupt . . . . .	34
10.9.1	Ненужный InterruptedException . . . . .	35
10.10	Поток обрабатывающий очередь . . . . .	35
10.11	Ожидание без блокировки . . . . .	35
10.11.1	Реализация с блокировкой . . . . .	36
10.11.2	Реализация без блокировок . . . . .	36
10.12	Ожидание из многих потоков . . . . .	37
10.12.1	Пишем свой внутренний синхронайзер . . . . .	37
10.12.2	Используем его . . . . .	38
10.12.3	Улучшаем производительность . . . . .	38
<b>11</b>	<b>FAA Based Queue</b>	<b>39</b>
11.1	Fetch-And-Add . . . . .	39
11.2	Obstruction-free queue on infinite array . . . . .	39
11.2.1	Michael-Scott queue of array segments . . . . .	39

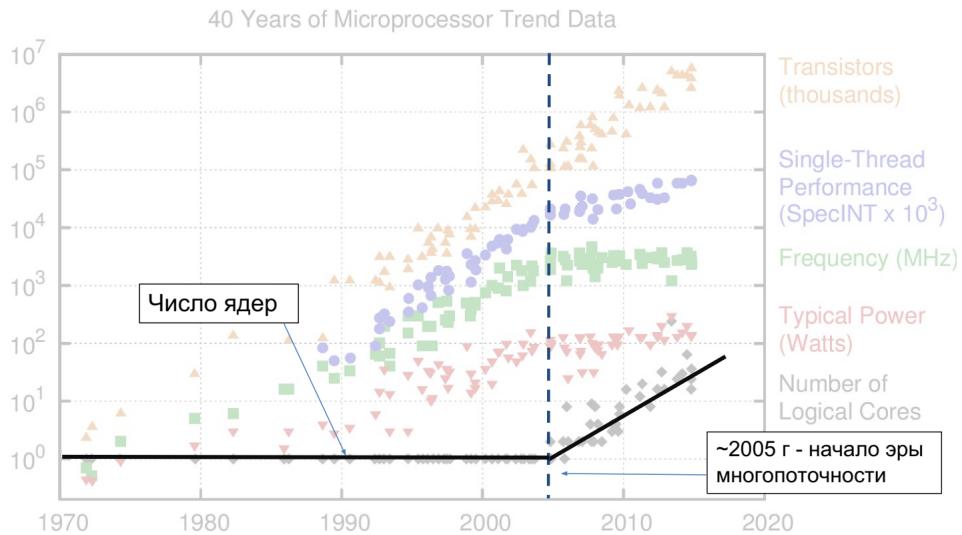
<b>12 Counter Sharding</b>	<b>40</b>
12.1 Naive counter . . . . .	40
12.2 Sharding . . . . .	40
<b>13 Elimination technique</b>	<b>41</b>
13.1 Naive Stack implementation . . . . .	41
13.2 Elimination . . . . .	41
<b>14 Железо и спин-локи</b>	<b>41</b>
14.1 Работа с общей памятью . . . . .	41
14.2 MESI . . . . .	41
14.2.1 MOSI . . . . .	41
14.3 Test-And-Set (aka get-And-Set) . . . . .	42
14.3.1 lock . . . . .	45
14.4 CLH Lock . . . . .	46
14.5 NUMA . . . . .	47
14.6 MCS Lock . . . . .	49

# 1 Introduction

## 1.1 Закон Мура

Каждые 2 года количество транзисторов на процессоре удваивается. До, примерно, 2005 года также росла частота ядра. Также начал замедляться рост производительность ядра. С 2005 года начался рост числа ядер.





**Определение.** *Масштабирование* - свойство системы выполнять больше действий при увеличении мощности(традиционное), количества ядер(многопоточное).

В реале не получается сделать все идеально и для этого нужно изучать многопоточное программирование.

## 1.2 Закон Амдала

$$S = \frac{1}{N}$$

где  $S$  - это ускорение кода

Или

$$S = \frac{1}{1 - P + P/N}$$

где  $P$  - доля параллельного кода

Максимальное ускорение кода достигается при  $N \rightarrow \infty$  и равно  $1/(1 - P)$

$$\begin{array}{ll} P & S \\ \hline 60\% & 2.5 \\ 95\% & 20 \\ 99\% & 100 \end{array}$$

Поэтому нам необходимо увеличивать долю параллельного кода для достижения наилучшей масштабируемости.

## 1.3 Разные виды параллелизма

### 1.3.1 Параллелизм на уровне инструкций (ILP)

Способы использования ILP

- Конвейер
- Суперскалярное исполнение<sup>1</sup>
  - Внеочередное исполнение
  - Переименование регистров<sup>2</sup>
  - Спекулятивное исполнение<sup>3</sup>
  - Предсказание переходов
- Длинное машинное слово (VLIW<sup>4</sup>)
- Векторизация (SIMD)

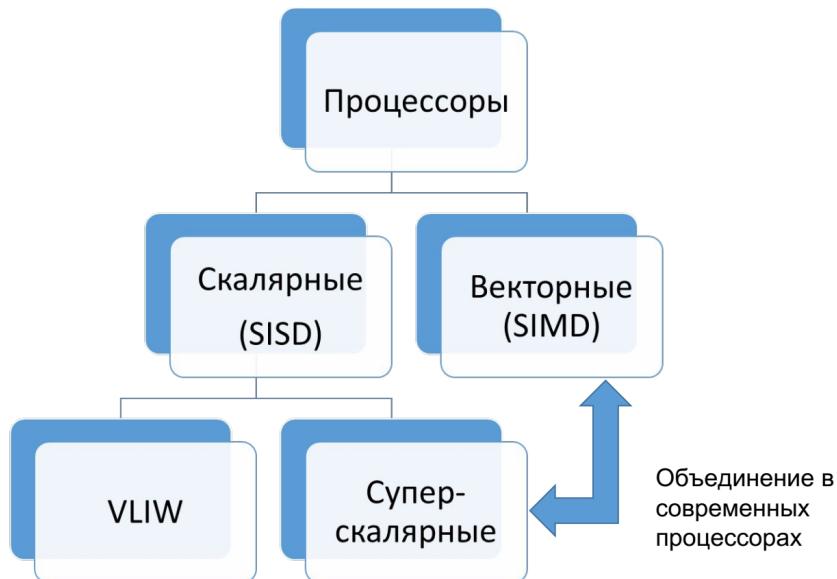


Рис. 1:

У параллелизма на уровне инструкций есть предел, поэтому нам необходимо параллельное программирование

**Симметричная мультипроцессорность (SMP)** Несколько вычислительных ядер у каждого свой поток исполняемых ресурсов

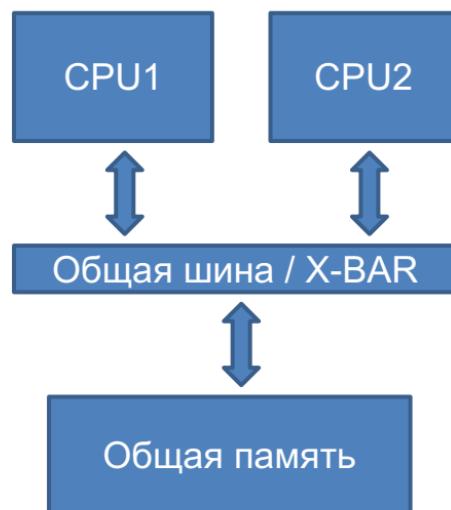


Рис. 2: SMP

**Одновременная многозадачность (SMT)** Два или более потока одновременно исполняются одним физическим ядром. Снаружи выглядит как SMP.

## <sup>0</sup>Instruction Level Parallelism

<sup>1</sup> Несколько операций за такт

<sup>2</sup>Чтобы не возникало ложной зависимости по регистрам

<sup>3</sup>Начинает выполнять одну из веток перехода, пытаясь ее предсказать

<sup>4</sup>Very Long Instruction Word

## Very Long Instruction Word 4 Symmetric Multiprocessing

#### <sup>4</sup>Simultaneous Multithreading

## Simultaneous Multithreading

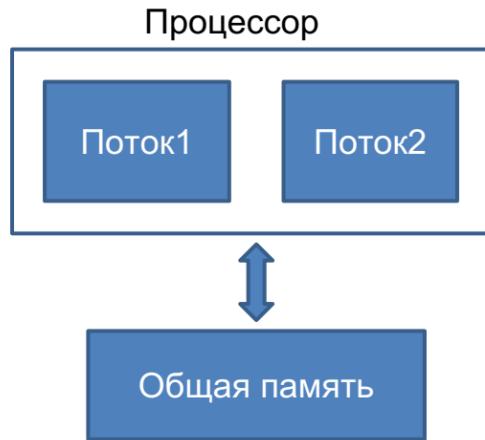


Рис. 3: SMT

**Ассимметричный доступ к памяти (NUMA)** Модель программирования та же, что в SMP, но без общей памяти.

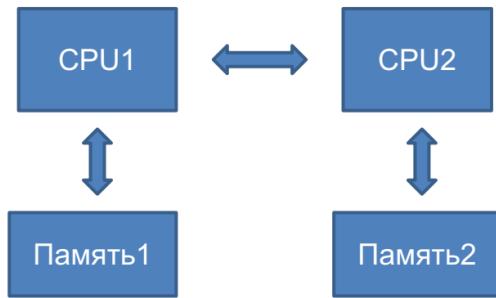


Рис. 4: NUMA

## 1.4 Операционные системы

- Типы
  - Однозадачные
  - Системы с пакетными задачами (batch processing)
  - Многозадачные / с разделением времени (time sharing)
    - \* Кооперативная многозадачность (cooperative multitasking)
    - \* Вытесняющая многозадачность (preemptive multitasking)
- История многозадачности
  - Изначально нужно было для раздела одной дорогой машины между несколькими пользователями
  - Теперь нужно для использования ресурсов одной многоядерной машины для множества задач

## 1.5 Основные понятия в современных ОС

- 
- **Определение.** *Процесс – владеет памятью и ресурсами.*
- 
- **Определение.** *Поток – контекст исполнения внутри процесса.*
  - В одном процессе может быть несколько потоков

- Все потоки работают с общей памятью процесса
- Но в теории мы их будем смешивать

## 1.6 Формализм

### 1.6.1 Модели программирования

- «Классическое» однопоточное / однозадачное
  - Можем использовать ресурсы многоядерной системы только запустив несколько независимых задач
- Многозадачное программирование
  - Возможность использовать ресурсы многоядерной системы в рамках решения одной задачи
  - Варианты:
    - \* Модель с общей памятью
    - \* Модель с передачей сообщений (распределенное программирование)

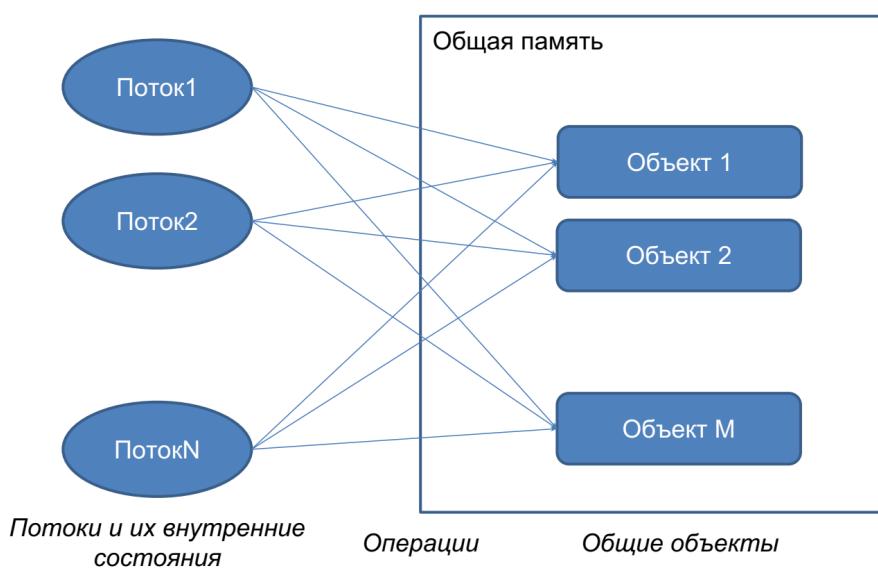


Рис. 5: Модель с общими объектами (общей памятью)

### 1.6.2 Общие объекты

- Потоки выполняют операции над общими, разделяемыми объектами
- В этой модели не важны операции внутри потоков
- Важна только коммуникация между потоками
- В этой модели единственный тип коммуникации между потоками — это работа с общими объектами

### 1.6.3 Общие переменные

- Общие переменные — это простейший тип общего объекта:
  - У него есть значение определенного типа
  - Есть операция чтения (read) и записи (write)
- Общие переменные — это базовые строительные блоки для многопоточных алгоритмов
- Модель с общими переменными — это хорошая абстракция современных многопроцессорных систем и многопоточных ОС
  - На практике, это область памяти процесса, которая одновременно доступна для чтения и записи всем потокам этого процесса

В теоретических трудах общие переменные называют регистрами

#### 1.6.4 Свойства многопоточных программ

- Последовательные программы детерминированы
  - Если нет использования случайных чисел и другого явного общения с недетерминированным миром
  - Их свойства можно установить анализируя последовательное исполнение при данных входных параметрах
- Многопоточные программы в общем случае недетерминированы
  - Даже если код каждого потока детерминирован
  - Результат работы зависит от фактического исполнения при данных входных параметрах
  - А этих исполнений может быть много
- Говорим «Программа A имеет свойство P» если она имеет это свойство при любом исполнении

#### 1.6.5 Моделирование многопоточного исполнения

1        shared int x = 0, y = 0

Thread P:

1            x = 1  
2            r1 = y  
3            stop

Thread Q:

1            y = 1  
2            r2 = x  
3            stop

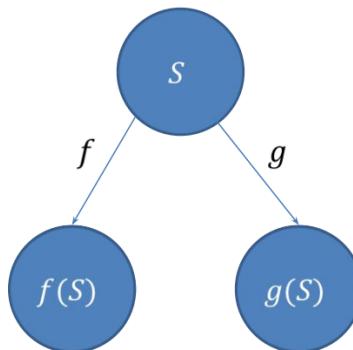


Рис. 6:

#### Моделирование исполнений через чередование операций

- $S$  — это общее состояние:
  - Состояние всех потоков (IP+locals)
  - И состояние всех общих объектов
- $f$  и  $g$  — это операции
  - Количество различных операций в каждом состоянии равно количеству потоков
- $f(S)$  — это новое состояние после применения операции  $f$  к состоянию  $S$

После исполнения этого кода для  $r1, r2$  возможны следующие пары значений:  $(0, 0), (0, 1), (1, 0), (1, 1)$ . Хотя при моделировании через чередование (рисунок 7) первого варианта не получается. Это случается, так как в современном процессоре запись не попадает сразу в общую память, а в начале буферизируется (т.к. запись долгая операция). Поэтому мы можем прочитать старое значение, т.к. чтение

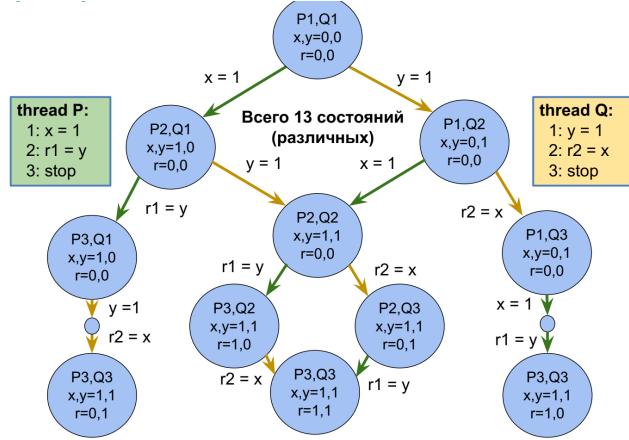


Рис. 7:

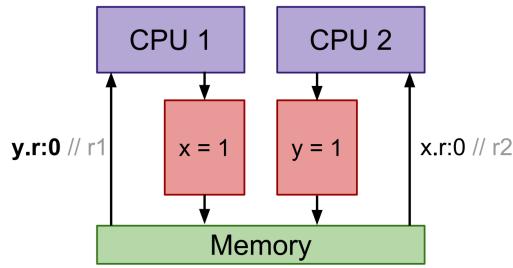


Рис. 8:

быстрая операция и новые значения еще лежат в буфере. Процессор может переставить инструкции, т.к. это может ускорить однопоточный код (процессор не знает о параллельности).

Модель чередования не параллельна

На самом деле в настоящих процессорах операции чтения и записи не мгновенные. Они происходят параллельно как в разных ядрах, так и в одном.

И вообще процессор обменивается с памятью сообщениями о чтении / записи и таких сообщений одновременно в обработке может быть очень много.

## 2 Lock-free Treiber Stack and Michael-Scott Queue

### 3 Определения и формализм

#### 3.1 Физическая реальность

- Свет (электромагнитные волны) в вакууме распространяется со скоростью  $\sim 3 \cdot 10^8$  м/с.
  - Это максимальный физический предел скорости
  - За один такт процессора с частотой 3 ГГц ( $3 \cdot 10^9$  Гц) свет в вакууме проходит всего 10 см.
- Соседние процессоры физически не могут синхронизировать свою работу и физически не могут определить порядок происходящих в них событий.
  - Они работают действительно физически параллельно
- Пусть  $a, b, c \in E$  — это физически атомарные (неделимые) события, происходящие в пространстве–времени (рисунок 9)
  - Говорим « $a$  предшествует  $b$ » или « $a$  произошло до  $b$ » (и записываем  $a \rightarrow b$ ), если свет от точки пространства–времени  $a$  успевает дойти до точки пространства–времени  $b$ .
  - Это отношение частичного порядка на событиях

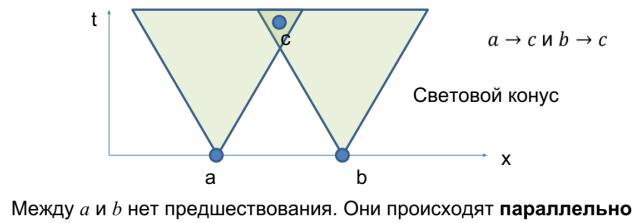


Рис. 9:

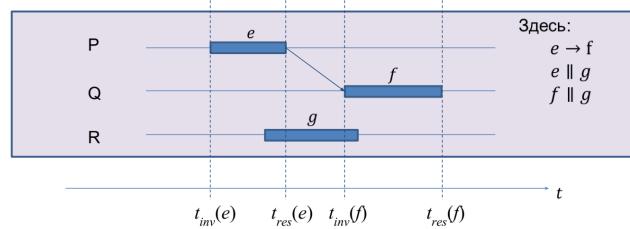


Рис. 10: Модель глобального времени

### 3.2 Модель «произошло до» (happens before)

- Впервые введена Л. Лампортом в 1978 году.
- Исполнение системы — это пара  $(H, \rightarrow_H)$ 
  - $H$  — это множество операций  $e, f, g, \dots$  (чтение и запись ячеек памяти и т. п.) произошедших во время исполнения
  - $\rightarrow_H$  — это транзитивное, антирефлексивное, ассимметричное отношение (частичный строгий порядок) на множестве операций
  - $e \rightarrow_H f$  означает, что « $e$  произошло до  $f$  в исполнении  $H$ ». Чаще всего исполнение  $H$  понятно из контекста и опускается
- Две операции  $e$  и  $f$  параллельны ( $e \parallel f$ ), если  $e \not\rightarrow f \wedge f \not\rightarrow e$ .
- Система — это набор всех возможных исполнений системы
- Говорим, что «система имеет свойство Р», если каждое исполнение системы имеет свойство Р.

### 3.3 Модель глобального времени

В этой модели каждая операция — это временный интервал (рисунок 10)  $e = [t_{inv}(e), t_{res}(e)]$  где  $t_{inv}(e), t_{res}(e) \in \mathbb{R}$  и

$$e \rightarrow f \stackrel{\text{def}}{=} t_{res}(e) < t_{inv}(f)$$

### 3.4 Обсуждение глобального времени

На самом деле никакого глобального времени нет и не может быть из-за физических ограничений.

- Это всего лишь механизм, позволяющий визуализировать факт существования параллельных операций.
- При доказательстве различных фактов и анализе свойств [исполнений] системы время не используется
  - Анализируются только операции и отношения «произошло до»

### 3.5 «Произошло до» на практике

- Современные языки программирования предоставляют программисту операции синхронизации:
  - Специальные механизмы чтения и записи переменных
  - Создание потоков и ожидание их завершения
  - Различные другие библиотечные примитивы для синхронизации
- Модель памяти языка программирования определяет то, каким образом исполнение операций синхронизации создает отношение «произошло до»
  - Без них разные потоки выполняются параллельно
  - Можно доказать те или иные свойства многопоточного кода, используя гарантии на возможные исполнения, которые дает модель памяти

### 3.6 Свойства исполнений над общими объектами

#### 3.6.1 Операции над общими объектами

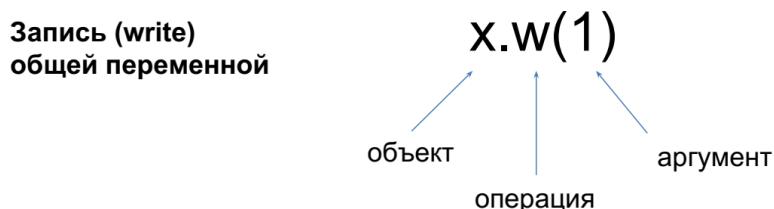


Рис. 11: Запись

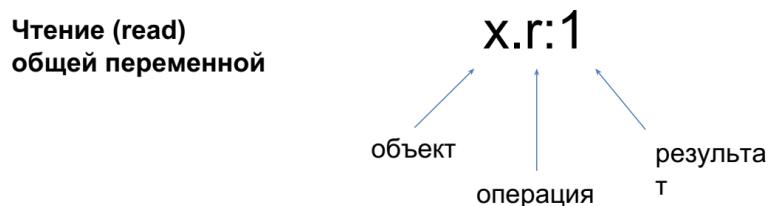


Рис. 12: Чтение

#### 3.6.2 Последовательное исполнение

**Определение.** Исполнение системы называется последовательным, если все операции линейно-упорядочены отношением «произошло до», то есть  $\forall e, f \in H: (e = f) \vee (e \rightarrow f) \vee (f \rightarrow e)$ .

#### 3.6.3 Конфликты и гонки данных (data race)

- 

**Определение.** Две операции над одной переменной, одна из которых запись, называются конфликтующими.

Конфликтующие операции не коммутируют в модели чередования.

- 

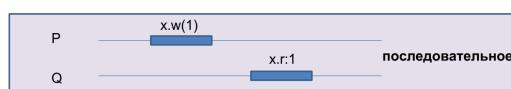


Рис. 13: Последовательное исполнение



Рис. 14: Параллельное исполнение

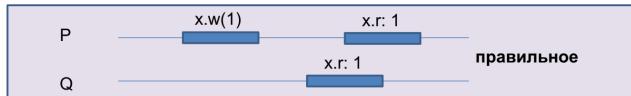


Рис. 15: Правильное исполнение

**Определение.** Если две конфликтующие операции произошли параллельно, то такая ситуация называется гонкой данных (*data race*).

- Это свойство конкретного исполнения.

**Определение.** Программа, в любом допустимом исполнении которой (с точки зрения модели памяти) нет гонок данных, называется корректно синхронизированной.

#### 3.6.4 Правильное исполнение

- $H|_p$  — сужение исполнения на поток  $P$ , то есть исполнение, где остались только операции, происходящие в потоке  $P$ .
- 

**Определение.** Исполнение называется **правильным** (*well-formed*), если его сужение на каждый поток  $P$  является последовательным (рисунок 15).

#### 3.6.5 Правильное исполнение и нотация

- $H|_p$  — сужение исполнения на поток  $P$  — это множество всех операций  $e \in H$ , таких что  $proc(e) = P$ .
  - Исполнение называется **правильным** (*well-formed*), если его сужение на каждый поток  $P$  является последовательным.
  - Задается программой, которую выполняет поток.
  -
- **Определение.** Объединение всех сужений на потоки называют **программным порядком** (*po* = *program order*).
  - Нас интересуют только правильные исполнения.
- $H|_x$  — сужение истории на объект  $x$  — это множество операций  $e \in H$ , таких что  $obj(e) = x$ . В правильном исполнении сужение на объект  $x$  обязательно является последовательным.

#### 3.6.6 Последовательная спецификация объекта

Если сужение исполнения на объект  $H|_x$  является последовательным, то можно проверить его на соответствие **последовательной спецификации объекта**.

#### 3.6.7 Допустимое последовательное исполнение

**Определение.** Последовательное исполнение является **допустимым** (*legal*), если выполнены последовательные спецификации всех объектов (рисунок 16).



Рис. 16: Допустимое исполнение

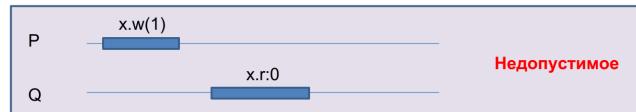


Рис. 17: Недопустимое исполнение

### 3.6.8 Условия согласованности (корректности)

**Корректные** последовательные программы должны считаться **согласованными**.

Условия согласованности:

- Согласованность при покое
- Последовательная согласованность
- Лианеризуемость
- и другие

#### Последовательная согласованность

**Определение.** Исполнение **последовательно согласовано**, если ему можно сопоставить эквивалентное ему допустимое исполнение, сохраняющее его программный порядок (рисунок 18).

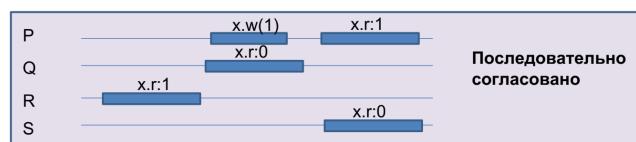
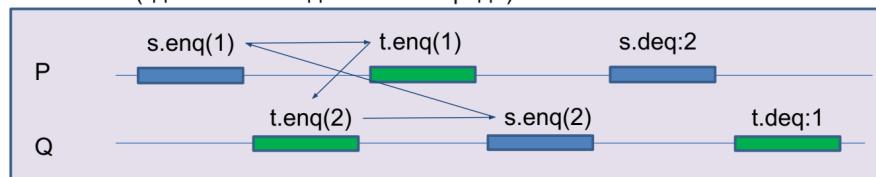


Рис. 18: Последовательно согласовано

Последовательная согласованность на каждом объекте исполнения не влечет последовательную согласованность всего исполнения (рисунок 19)

**ПРИМЕР:** (здесь  $s$  и  $t$  это две FIFO очереди)



Ой! Цикл -- не упорядочить линейно

Рис. 19:

Модель памяти языков программирования и системы исполнения кода используют последовательную согласованность для своих формулировок.

#### Лианеризуемость

**Определение.** Исполнение **лианеризуемо**, если можно сопоставить эквивалентное ему допустимое последовательное исполнение, которое сохраняет отношение «произошло до».

Свойства лианеризуемости:

- В лианеризуемом исполнении каждой операции  $e$  можно сопоставить точку глобального времени (**точку лианеризации**)  $t(e) \in \mathbb{R}$  так, что время различных операций различно и

$$e \rightarrow f \Rightarrow t(e) < t(f)$$

- Лианеризуемость **локальна**. Лианеризуемость исполнения на каждом объекте эквивалентна лианеризуемости системы целиком.

•

**Определение.** Операции над лианеризуемыми объектами называют **атомарными**.

**Лианеризуемость в глобальном времени** В глобальном времени исполнение лианеризуемо тогда и только тогда, когда точки лианеризуемости можно выбрать так, что

$$\forall e: t_{inv}(e) < t(e) < t_{res}(e)$$

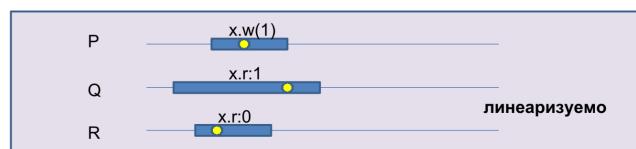


Рис. 20: Лианеризуемость

### 3.7 Лианеризуемость

Исполнение системы, выполняющей операции над лианеризуемыми объектами, можно анализировать в модели чередования.

Из более простых лианеризуемых объектов можно сделать лианеризуемые объекты более высокого уровня.

Когда говорят, что объект безопасен для использования из нескольких потоков, подразумевают, что операции над ним лианеризуемы.

#### 3.7.1 Применительно к Java

- Все операции над volatile полями в Java, согласно JMM, являются операциями синхронизации, которые всегда линейно-упорядочены в любом исполнении и согласованы с точки зрения чтения/записи.
- Но операции над не volatile полями могут нарушать не только лианеризуемость, но даже последовательную согласованность, при отсутствии синхронизации.
- Если программа корректно синхронизирована, то JMM дает гарантию последовательно согласованного исполнения всего кода.

## 4 Построение атомарных объектов и блокировки

### 4.1 Сложные и составные операции

#### 4.1.1 Формализация сложных операций

- Операция  $e \in H$  может быть сложной. Даже чтение из памяти это продолжительное по времени действие, много шагов.

- **Событие**

- Неделимое, простое физическое действие
- Множество событий обозначим  $G$
- Каждая операция это множество событий  $e \subset G$

- Из всех событий выделяют два наиболее важных
  - Вызов операции  $inv(e) \in G$
  - Завершение операции  $res(e) \in G$
- Декомпозиция исполнения —  $(H, G, \rightarrow_G, \text{inv}, \text{res})$ 
  - $H$  — это множество операций ( $\forall e \in H: e \subset G$ )
  - $G$  — это множество событий
  - $\rightarrow_G$  — отношение «произошло до» на событиях из  $G$ .
  - $\text{inv}, \text{res}$  — функции из  $H$  в  $G$ , такие что:

$$\forall e \in H: \text{inv}(e) \rightarrow_G \text{res}(e)$$

$$\forall e \in H, g \in e, g \neq \text{inv}(e), g \neq \text{res}(e): \text{inv}(e) \rightarrow_G g \rightarrow_G \text{res}(e)$$

## 5 Consensus

### 5.1 Задача о консенсусе

Каждый поток использует объект Consensus один раз

- Согласованность: все потоки должны вернуть одно и тоже значение из метода `decide`
- Обоснованность: возвращенное значение было входным значением какого-то из потоков
- Без ожидания

### 5.2 Консенсусное число

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

**Теорема 5.1.** Атомарные регистры имеют консенсусное число 1.

### 5.3 Модель

- x-валентное состояние системы — консенсус во всех нижестоящих листьях будет x.
- Бивалентное состояние — возможен консенсус как 0, так и 1.
- Критическое состояние — такое бивалентное состояние, у которого все дети одновалентны

### 5.4 Read–Modify–Write регистры

### 5.5 Универсальность консенсуса

**Теорема 5.2.** Любой последовательный объект можно реализовать без ожидания для N потоков, используя консенсусный протокол для N потоков

## 6 Алгоритмы без блокировок: Построение на регистрах

### 6.1 Безусловные условия прогресса

- Отсутствие помех — Если несколько потоков пытаются выполнить операцию, то любой из них должен выполнить ее за конечное время, если все другие остановить в любом месте
- Отсутствие блокировки — если несколько потоков пытаются выполнить операцию, то хотя бы один из них должен выполнить ее за конечное время, независимо от действия или бездействия других потоков
- Отсутствие ожидания — если поток хочет выполнить операцию, то он выполнит ее за конечное время независимо от других потоков

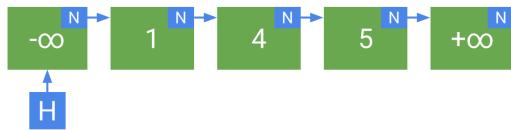


Рис. 21: Односвязный список

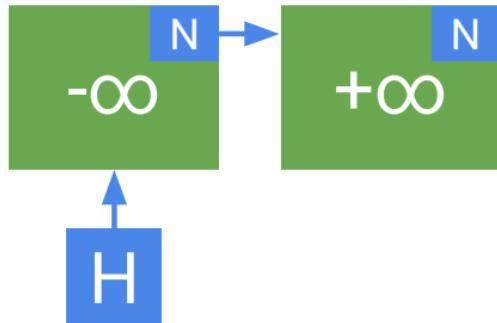


Рис. 22: Пустой список

## 7 Практические построения на списках

### 7.1 МНожество на односвязном списке

```

1 interface Set{
2     fun add(key: Int)
3     fun contains(key: Int): Boolean
4     fun remove(key: Int)
5 }

```

Элементы упорядочены по возрастанию (рисунок 21).

Пустой список состоит из двух граничных элементов (рисунок 22).

#### 7.1.1 Алгоритм

- Элементы упорядочены по возрастанию
- Ищем окно ( $cur, next$ ), что  $cur.KEY < key \leqslant next.KEY$  и  $cur.N = next$
- Искомый элемент будет в  $next$
- Новый элемент добавляем между  $cur$  и  $next$

#### 7.1.2 Псевдокод

```

1 class Node(var N: Node, val key: Int)
2
3 val head = Node(-infinity, Node(infinity, null))
4
5 fun findWindow(key: Int): (Node, Node) {
6     cur := head
7     next := cur.N
8     while (next.key < key) {
9         cur := next
10        next := cur.N
11    }
12    return (cur, next)
13 }
14
15 fun contains(key: Int): Boolean {
16     (cur, next) := findWindow(key)

```

```

17         return next.key = key
18     }
19
20    fun add(key: Int) {
21        (cur, next) := findWindow(key)
22        if (next.key != key) {
23            cur.N := Node(key, next)
24        }
25    }
26
27    fun remove(key: Int) {
28        (cur, next) := findWindow(key)
29        if (next.key == key) {
30            cur.N = next.N
31        }
32    }

```

### 7.1.3 Проблема

При параллельном удалении соседних элементов могут возникнуть проблемы с перенаправление ссылок: например, мы успели удалить *cur* перед тем как переназначить ссылки и у нас возникнет NullPointerException или список просто разорвётся

## 7.2 Грубая синхронизация

- Coarse-grained locking
- Используем общую блокировку для всех операций
- ⇒ обеспечиваем последовательное исполнение
- В Java для этого можно использовать synchronized и java.util.concurrent.locks.ReentrantLock

## 7.3 Тонкая блокировка

- Fine-Grained locking
- Своя блокировка на каждый элемент
- При поиске окна держим блокировку на текущий и следующий элемент

### 7.3.1 Корректность

- Поиск окна: запись и чтение *cur.N* не может происходить параллельно
- Модификация: во время изменения окно защищено блокировкой ⇒ атомарно
- $\forall k$ : операции с ключом *k* лианеризуемы ⇒ всё исполнение лианеризуемо
- Операции с ключом *k* упорядочены взятием блокировки

## 7.4 Оптимистичная синхронизация

### 7.4.1 Алгоритм абстрактной операции

1. Найти окно (*cur, next*) без синхронизации
2. Взять блокировки на *cur* и *next*
3. Проверить инвариант *cur.N = next*
4. Проверить, что *cur* не удалён
5. Выполнить операцию
6. При любой ошибке начать заново

#### 7.4.2 Проверка, что узел не удалён

- Держи блокировку на  $cur$  и  $cur$  удалён  $\Rightarrow$  не увидим при проходе
- Попробуем найти  $cur$  ещё раз за  $O(n)$  и проверим, что  $cur.N = next$

#### 7.4.3 Валидация окна

```
1 fun validate(cur: Node, next: Node): Boolean {  
2     node := head  
3     while (node.key < cur.key) {  
4         node = node.N  
5     }  
6     return (cur, next) = (node, node.N)  
7 }
```

#### 7.4.4 Корректность

- Поиск: запись и чтение  $cur.N$  связаны отношением «произошло до»
- Можем говорить о лианеризуемости операция над одинаковыми ключами
- Точка лианеризации - взятие блокировки над  $cur$

### 7.5 Ленивая синхронизация

#### 7.5.1 Идея ленивого удаления

- Добавим в  $Node$  поле  $removed$  типа  $Boolean$
- Удаление в две фазы
  1.  $node.removed = true$  — логическое удаление
  2. Физическое удаление из списка
- Инвариант: все не удалённые вершины в списке
- $\Rightarrow$  теперь не надо проходить по всему списку в  $validate()$

#### 7.5.2 Псевдокод

```
1 fun validate(cur: Node, next: Node): Boolean {  
2     return !cur.removed && !next.removed && cur.N==next  
3 }
```

### 7.6 Неблокирующая синхронизация

#### 7.6.1 Неблокирующий поиск

- На момент чтения поля  $N$  видим состояние на момент записи  $N$  или новее
- $\Rightarrow$  можем не брать блокировку при поиске

```
1 fun contains(key: Int): Boolean {  
2     (cur, next) := findWindow(key)  
3     return next.key = key  
4 }
```

### 7.6.2 Неблокирующая модификация

- Объединим  $N$  и  $removed$  в одну переменную, пару  $(N, removed)$
- Будем менять  $(N, removed)$  атомарно
- Каждая операция модификации будет выполняться одни успешным CAS-ом
- В Java для этого есть AtomicMarkableReference

## 8 Lock-free хеш таблица с открытой адресацией

### 8.1 ConcurrentHashMap

- Работает за  $O(1)$  в среднем
- Использует внутри блокировки  $\Rightarrow$  не так хорошо масштабируется
- Хранит списки при коллизии (рисунок 23)
  - $\Rightarrow$  лишние cache miss-ы
  - зато может хранить дерево при постоянных коллизиях (рисунок 24)

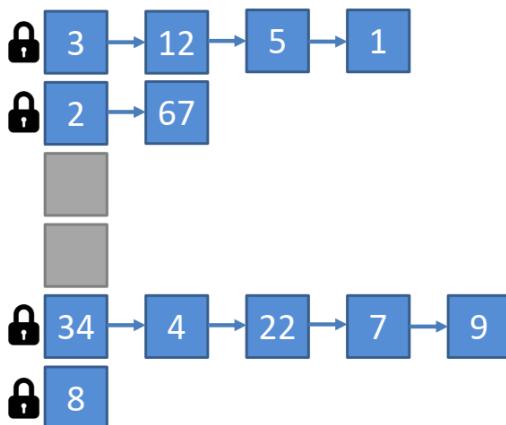


Рис. 23: Simple ConcureentHashMap

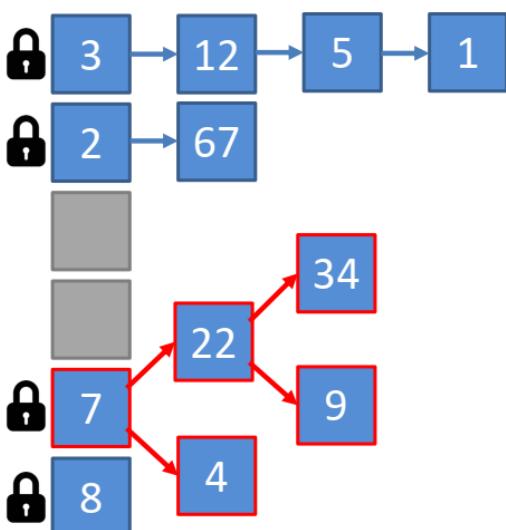


Рис. 24: ConcurrentHashMap with tree

## 8.2 Skip List

- Работает за  $O(\log(n))$  в среднем
- «Дерево» списков
  - Много лишних объектов
  - $\Rightarrow$  постоянные cache miss-ы и нагружает GC
- И вообще это не хеш таблица

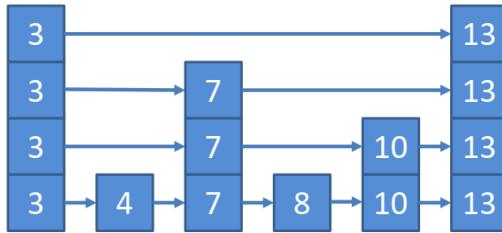


Рис. 25: Skip List

## 8.3 Потенциальные проблемы

- ConcurrentHashMap
  - Использует внутри блокировки  $\Rightarrow$  не lock-free<sup>5</sup>
  - Использует списки  $\Rightarrow$  cache miss-ы
- Много «лишних» объектов
- Постоянные cache miss-ы

## 8.4 NonBlockingHashMap

- Использует открытую адресацию
- Lock-free
  - Гарантирует, что система не стоит на месте даже при неудачном scheduling-e
  - Можно вставлять читать во время перехеширования

```
1 public T getInternal(long key) {  
2     int i = index(key);  
3     long k;  
4     int probes = 0;  
5     while ((k = keys.get(i)) != key) {  
6         if (k == NULL_KEY)  
7             return null;  
8         if (++probes >= MAX_PROBES)  
9             return null;  
10        if (i == 0)  
11            i = length;  
12        i--;  
13    }  
14    return values.get(i);  
15 }
```

<sup>5</sup>Это важно для высоконагруженных систем

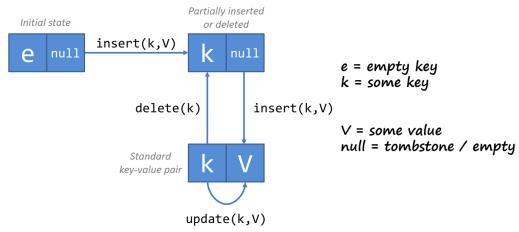


Рис. 26: Простая жизнь ячейки

#### 8.4.1 Пара деталей

- Теория говорит, что размер таблицы должен быть простым числом, но так как операция взятия по модулю дорогая, то на практике используется степени двойки, для которых требуется только сдвиг
- В теории при поиске элемента лучше смотреть не на следующий элемент, но на практике наоборот, так как следующий уже скорее всего закэширован

### 8.5 MRSW хеш таблица (рисунок 26)

Увеличение в случае одного писателя:

- Создаем новую таблицу
- Копируем элементы
- Меняем ссылку на таблицу

### 8.6 MRMW хеш таблица

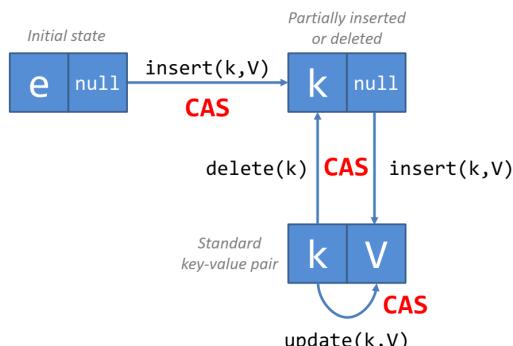


Рис. 27: Жизнь ячейки без переноса

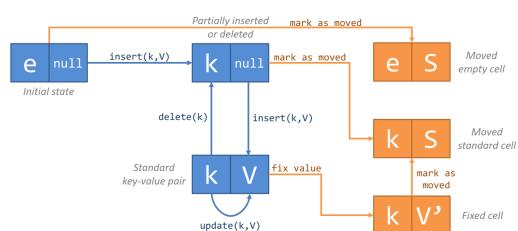


Рис. 28: Жизнь ячейки с переносом

#### 8.6.1 Кооперация при переносе

- Переносить блоками по  $k$  элементов
- Переносом занимается один выделенный поток

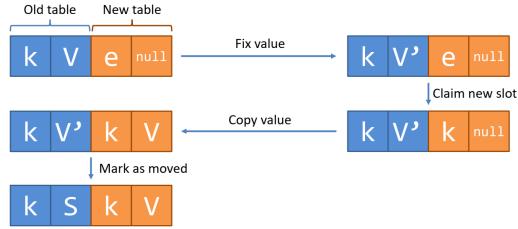


Рис. 29: Другой взгляд на перенос

## 9 CASN

### 9.1 Списки против массивов

- Список (рисунок 30)
  - $3 * N$  слов памяти (минимум) (рисунок 32)
  - при многопоточном использовании хватает одного CAS для добавления элемента (рисунок 34)
- Массив (рисунок 31)
  - $5 + N$  слов памяти (до  $5 + 2 * N$  при x2 резерве) (рисунок 33)
  - Как минимум в два раз быстрее работает с памятью (обычно растет на попрядок) (при однопоточной работе)
  - Для добавления нужно CAS2 (для size и элемента) — эта операция не поддерживается на процессорах (рисунок 35)



Рис. 30: Список

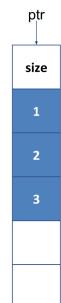
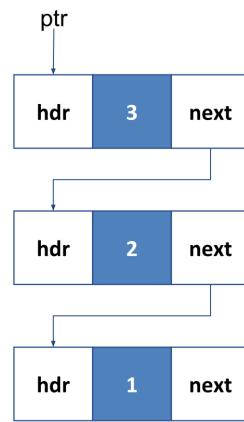


Рис. 31: Массив

### 9.2 CASn

```

1 class Ref<T>(initial: T) {
2     var _v = atomic<Any?>(initial)
3
4     var value: T
  
```



$3 \times N$  слов памяти

Рис. 32:

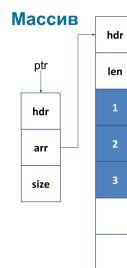


Рис. 33:

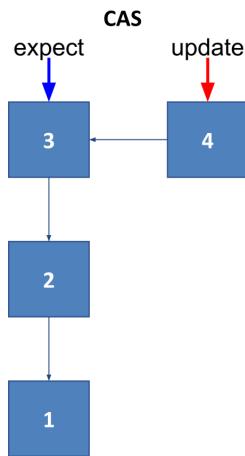


Рис. 34: Стек на списке

```

5     get() {
6         return _v as T //TODO
7     }
8
9     set(upd) {
10        this._v = upd //TODO
11    }
12 }
```

### 9.2.1 DCSS

1. Рисунок 36
2. Рисунок 37

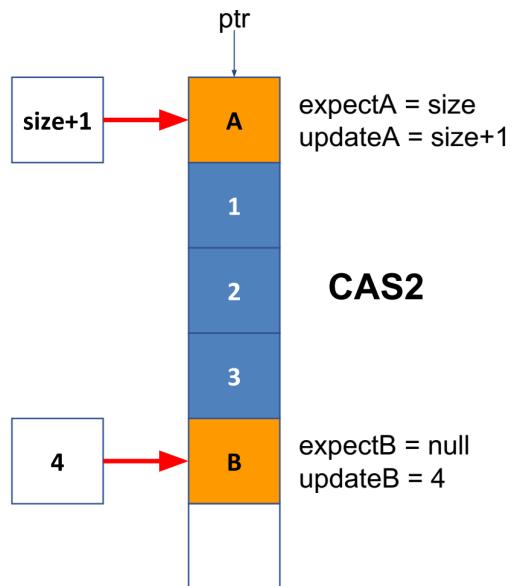


Рис. 35: Стек на массиве

3. Рисунок 38
4. Рисунок 39
5. Рисунок 40
6. Рисунок 41

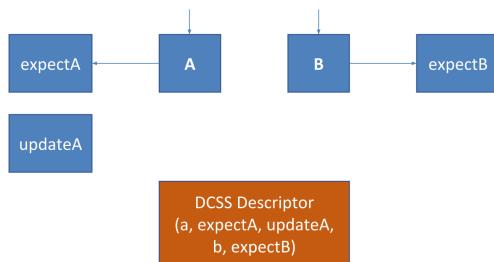


Рис. 36: init descriptor

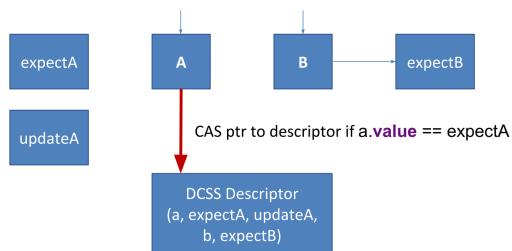


Рис. 37: prepare

### Псевдокод

```

1 fun <A,B> dcss(
2     a: Ref<A>, expectA: A, updateA: A,
3     b: Ref<B>, expectB: B) =
4     atomic {
5         if (a.value == expectA && b.value == expectB) {

```

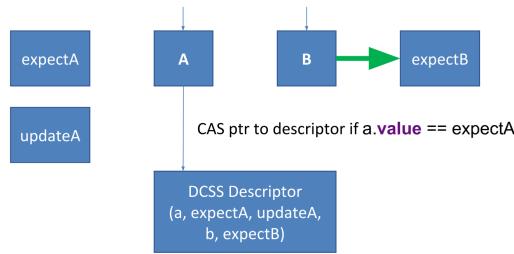


Рис. 38: read `b.value`

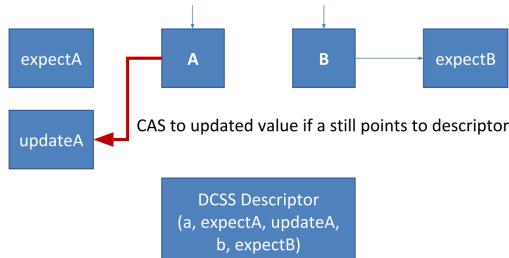


Рис. 39: complete when success

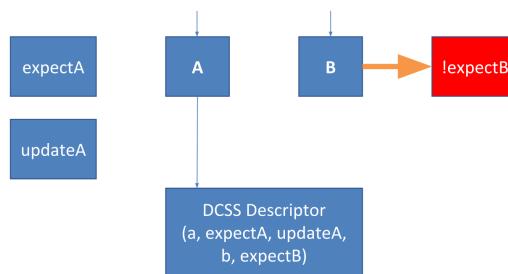


Рис. 40: complete (alternative)

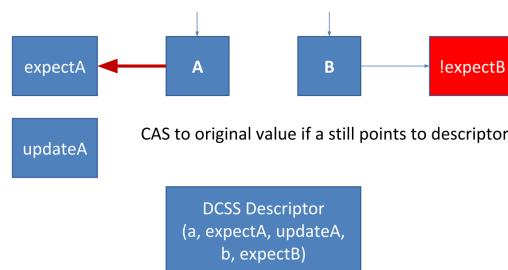


Рис. 41: compete when failed

```

6           a.value = updateA
7       }
8   }
```

### 9.3 Наблюдения и замечания

- `complete` работает без циклов

Реализует state machine которая идёт только вперёд

- Но есть циклы помохи другим потокам

Алгоритм lock-free

- Адреса `A` & `B` должны быть упорядочены

Либо может переполниться стек при помохи



Рис. 42: DCSS all

- Один из способов: Restricted DCSS (RDCSS)  
А и В берутся из разных "регионов" (не пересекающихся)

#### 9.4 Подход к реализации

```

1 abstract class Descriptor {
2     abstract fun complete()
3 }
4
5 class Ref<T>(initial : T) {
6     var _v = atomic<Any?>(initial)
7
8     var value: T
9         get() {
10         _v.loop{ cur ->
11             when(cur) {
12                 is Descriptor -> cur.complete()
13                 else -> return cur
14             }
15         }
16     }
17
18     set(upd) {
19         _v.loop{ cur ->
20             when(cur) {
21                 is Descriptor -> cur.complete()
22                 else -> if (_v.compareAndSet(cur, upd))
23                     return
24             }
25         }
26     }
27 }
28
29 class RDCSSDescriptor<A, B>(val a: Ref<A>, val expectA: A, val updateA: A, val b: Ref<B>,
30     override fun complete() {
31         val update = if (b.value == expectB) updateA else expectA
32         a._v.compareAndSet(this, update)
33     }
34 }
```

#### 9.5 DCSS Mod

1. Рисунок 43

2. Рисунок 44
3. Рисунок 45
4. Рисунок 46
5. Рисунок 47

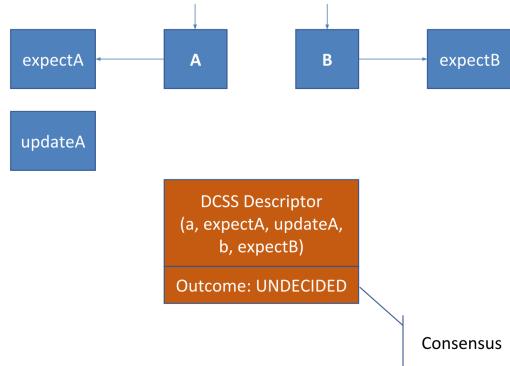


Рис. 43: init state

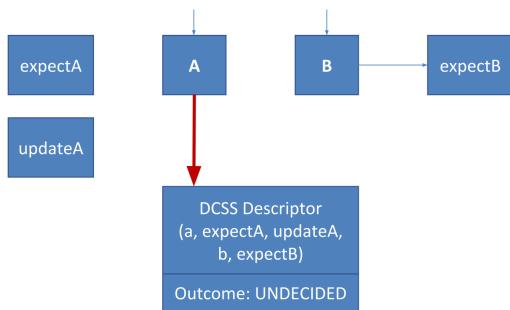


Рис. 44: prepare

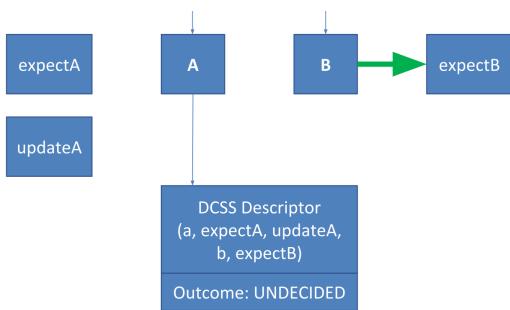


Рис. 45: read b.value

## 9.6 CASN

### 9.6.1 Псевдокод CAS2

```

1 fun <A,B> cas2 (a: Ref<A>, expectA: A, updateA: A, b: Ref<B>, expectB: B, updateB: B): Bool
2   if (a.value == expectA && b.value == expectB) {
3     a.value = updateA
4     b.value = updateB
5     true
6   } else {
7     false
  
```

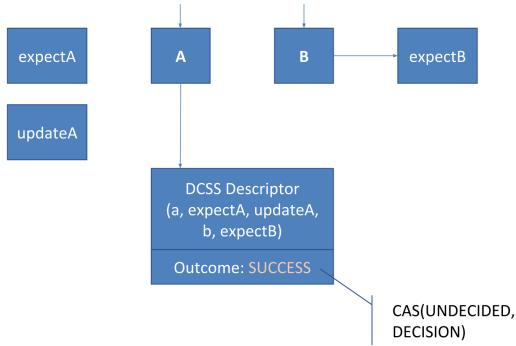


Рис. 46: reach consensus

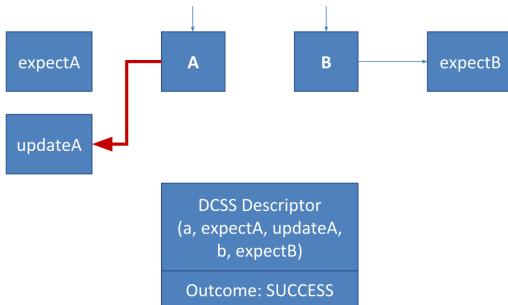


Рис. 47: complete

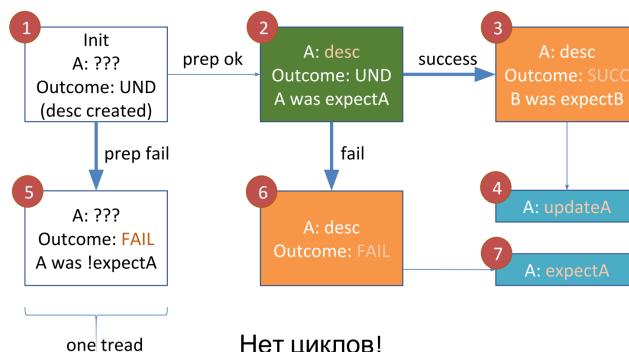


Рис. 48: all states

```
8      }
9 }
```

1. Рисунок 49
2. Рисунок 50
3. Рисунок 51
4. Рисунок 52
5. Рисунок 53
6. Рисунок 54

## 10 Мониторы и ожидание

### 10.1 Объекты как функции

- Операция над объектом как функция  $f(S, P) = (S', R)$

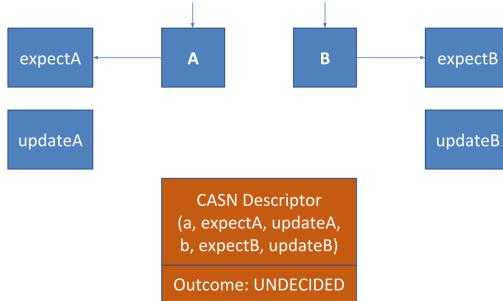


Рис. 49: init descriptor

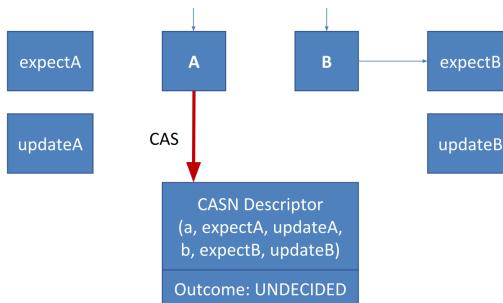


Рис. 50: prepare(1)

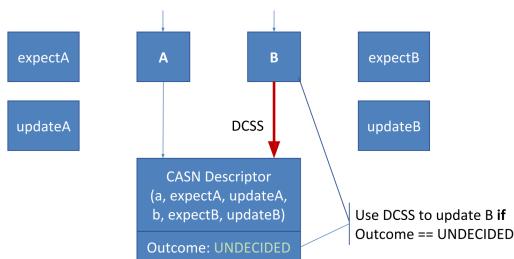


Рис. 51: prepare(2)

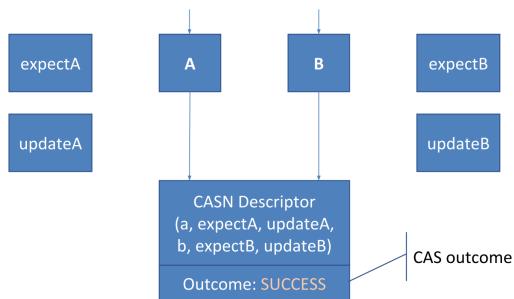


Рис. 52: decide

- Ранее операции были **всюду определены** на паре  $(S, P)$ 
  - Если операцию нельзя выполнить, то результат — ошибка или исключение
- В общем случае операции могут быть **частично определены** на множестве пар  $(S, P)$ 
  - Операция не может завершиться и *ждёт*

## 10.2 Очередь ограниченного размера с ожиданием

- Обычные операции
  - **size()**: int — узнать текущий размер

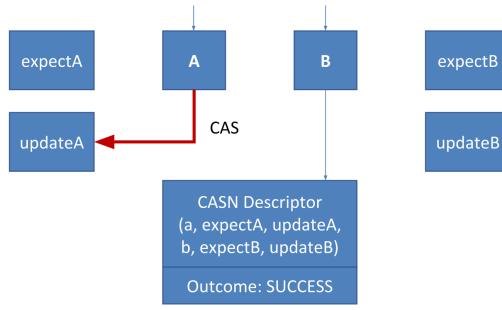


Рис. 53: complete(1)

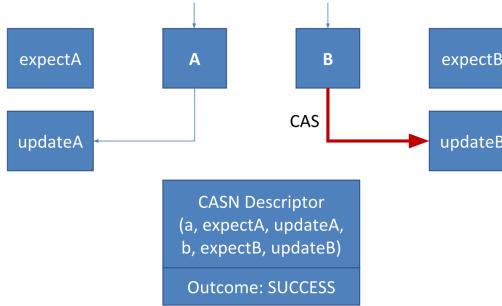


Рис. 54: complete(2)

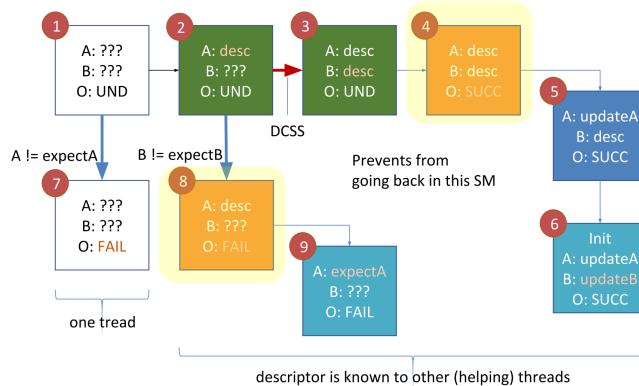


Рис. 55: all states

- **offer(item): Boolean** — вернет **false** если свободного места нет
- **poll(): item?** — вернёт **null** если очередь пуста

- Операции с ожиданием

- **put(item)** — положить элемент в очередь если есть свободное место (иначе ждёт, в этом состоянии операция не определена)
- **take(): item** — взять элемент из очереди если она не пуста (иначе ждёт)

### 10.3 Лианеризуемость операций с ожиданием

- Расширим понятие исполнения

- Есть событие вызова  $inv(A)$
- Но не обязательен ответ  $resp(A)$

\*

**Определение.**  $A$  это **незавершенная операция**, если нет  $resp(A)$

\*

**Определение.**  $inv(A)$  — это **незавершенный вызов**

- 

### Определение. Исполнение лианеризуемо

- Если для незавершенных операций можно
  - \* Либо добавить ответы
  - \* Либо выкинуть их из исполнения
- Чтобы получилось допустимое последовательное исполнение:  $inv(A_1) \rightarrow reps(A_1) \rightarrow inv(A_2) \rightarrow resp(A_2) \dots$

Рисунки 56 и 57

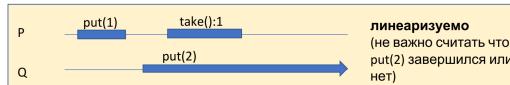


Рис. 56: Лианеризуемое исполнение

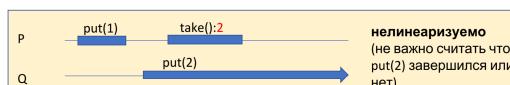


Рис. 57: Нелианеризуемое исполнение

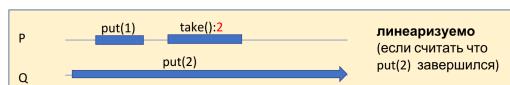


Рис. 58:

Но если очередь может содержать только один элемент, то

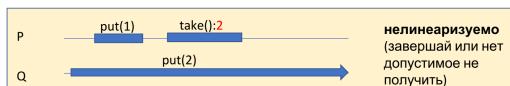


Рис. 59:

## 10.4 Реализация через монитор

**Monitor = mutex + conditional variables**

- Взаимное исключение для защиты данных от одновременного изменения
- Условные переменные для ожидания
- Мониторы придуманы Энтони Хоар (1974) (поэтому они также называются "Hoare monitors")

## 10.5 Монитор в Java

- В Java каждый объект имеет монитор с *одной* условной переменной
  - **synchronized**
    - \* **monitorenter** (lock)
    - \* **monitorexit** (unlock)
  - **wait**, **notify**, **notifyAll** — для работы с условной переменной

В некоторых системах `wait()` реализован как хэш-таблица, при чем при коллизии кого-нибудь разбудим. Поэтому ждать нужно в цикле, чтобы каждый раз проверять условие.

`Wait()` и `notify()` можно использовать только внутри критической секции (в `synchronized`)

Так как в мониторе только одна условная переменная, то, если у нас есть потоки ожидающие разных событий, то мы не можем использовать метод `notify()`, так как мы можем разбудить не тот поток и, поэтому тоже `wait` надо использовать в цикле.

## 10.6 Циклическая очередь с ожиданием на массиве

```
1 public class BlockingQueue<T> {  
2     private final T[] items;  
3     private final int n;  
4     private int head;  
5     private int tail;  
6 }
```

### 10.6.1 size()

Используем грубую синхронизацию через встроенный в Java объекты монитор

```
1 private synchronized int size() {  
2     return (tail - head + ) % n;  
3 }
```

### 10.6.2 Неждущий poll()

```
1 public synchronized T poll() {  
2     if (head == tail) return null;  
3     T result = items[head];  
4     items[head] = null;  
5     if ((tail + 1) % n == head) notifyAll();  
6     head = (head + 1) % n;  
7     return result;  
8 }
```

### 10.6.3 Ждущий take()

```
1 public synchronized T take() throws InterruptedException {  
2     while (head == tail) wait();  
3     T result = items[head];  
4     items[head] = null;  
5     if ((tail + 1) % n == head) notifyAll();  
6     head = (head + 1) % n;  
7     return result;  
8 }
```

### 10.6.4 Неждущий offer()

```
1 public synchronized boolean offer(T item) {  
2     int next = (tail + 1) % n;  
3     if (next == head) return false;  
4     items[tail] = item;  
5     if (tail == head) notifyAll();  
6     tail = next;  
7     return true;  
8 }
```

### 10.6.5 Ждущий put()

```
1 public synchronized void put(T item) throws InterruptedException {  
2     while (true) {  
3         int next = (tail + 1) % n;  
4         if (next == head) {  
5             wait();  
6         } else {  
7             items[tail] = item;  
8             if (tail == head) notifyAll();  
9             tail = next;  
10            return;  
11        }  
12    }  
13 }
```

```

6         continue;
7     }
8     items[ tail ] = item;
9     if ( tail == head ) notifyAll();
10    tail = next;
11    return;
12  }
13 }

```

## 10.7 `notify()` vs `notifyAll()`

Если бы для каждого условия была своя отдельная переменная, то можно использовать `notify()`.

## 10.8 `ReentrantLock`

- В пакете `java.util.concurrent` есть интерфейс `Lock`
  - С методами `lock`, `unlock`, `newCondition`
  - И `ReentrantLock` — его реализация
  - Интерфейс `Condition` для условных переменных с методами `await`, `signal`, `signalAll`
  - Можем завести отдельную переменную для каждого условия

```

1 private final Lock lock = new ReentrantLock();
2 private final Condition notFull = lock.newCondition();
3 private final Condition notEmpty = lock.newCondition();

```

Но даже в этом случае мы не можем бездумно использовать `signal`

### 10.8.1 Проблематичный сценарий

- Рассмотрим такую последовательность операций (рисунок 60)
  1. Два `put` ждут `notFull`
  2. Пришёл `take`, сделал очередь неполной, послал `notFull`
  3. Но один из `put` не успел проснуться и взять `lock` по сигналу
  4. Пришёл ещё `take`. Очередь уже не полная, так что он не посыпает еще один сигнал `notFull`
- Итого: один `put` проснулся, а другой продолжает спать, хотя очередь уже не полна
- Вывод: в отличии от `notifyAll`/`signalAll` оптимизировать вызовы `notify`/`signal` нужно осторожно (в данном случае нельзя)

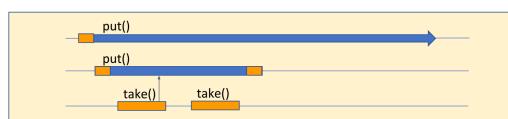


Рис. 60:

## 10.9 Подробней про `interrupt`

- У каждого потока есть `interrupted` флаг
  - Его ставит метод `Thread.interrupt`
  - Его проверяют методы `wait`, `await` и тому подобное
  - В случае обнаружения выставленного флага эти методы
    - \* Прекращают ждать
    - \* Сбрасывают флаг

\* Кидают **InterruptedException**

- Это **кооперативный** способ прерывания заблокированных потоков
  - Не нужно знать что именно ждёт поток
  - Но не надейтесь, что это будет работать в случае использования сторонних библиотек

### 10.9.1 Ненужный **InterruptedException**

Если нужно реализовать метод, который ждёт но не выкидывает **InterruptedException**, то interrupted флаг надо перевыставить

```
1 public T takeOrNull() {  
2     try {  
3         return take();  
4     } catch (InterruptedException e) {  
5         Thread.currentThread().interrupt();  
6         return null;  
7     }  
8 }
```

## 10.10 Поток обрабатывающий очередь

- Заводим свой флаг сигнализирующий, что поток надо остановить
  - В отличие от флага interrupted нет риска, что какой-то сторонний метод его случайно сбросит
- Метод **run** выходит в случае прерывания
  - Флаг interrupted перевыставлять не надо

```
1 public class DoSomething<T> extends Thread {  
2     private final BlockingQueue<T> queue;  
3     private volatile boolean closed;  
4  
5     public void close() {  
6         closed = true;  
7         interrupt();  
8     }  
9  
10    @Override  
11    public void run() {  
12        try {  
13            while (!closed) {  
14                T item = queue.take();  
15                doSomething(item);  
16            }  
17        } catch (InterruptedException e) {  
18            //ignore exception  
19        }  
20    }  
21 }
```

## 10.11 Ожидание без блокировки

Пример: обновляемое значение

- Почти очередь на один элемент
- Операции

- **update(item)** — обновить текущее значение
- **remove():item?** — забрать и сбросить текущее значение
- **take():item** — ждёт пока новое значение появится

### 10.11.1 Реализация с блокировкой

```

1 class DataHolder<T> {
2     private var value: T? = null
3     private val lock = ReentrantLock()
4     private val updated = lock.newCondition()
5
6     fun update(item: T) = lock.withLock {
7         value = item
8         updated.signal()
9     }
10
11    fun remove(): T? = lock.withLock {
12        value.also { value = null }
13    }
14
15    fun take(): T = lock.withLock {
16        while (value == null) updated.await()
17        value!!.also { value = null }
18    }
19 }
```

### 10.11.2 Реализация без блокировок

```

1 class DataHolder<T> {
2     private val v = atomic<T?>(null)
3
4     fun update(item: T) {
5         v.value = item // volatile write
6         LockSupport.unpark()
7     }
8
9     fun remove(): T? {
10        v.loop { cur ->
11            if (cur == null) return null
12            if (v.compareAndSet(cur, null)) return cur
13        }
14    }
15 }
```

**Ожидание без блокировки (park())** Здесь важен порядок: обновить → разбудить (**unpark()**)

```

1 class TakerThread<T>: Thread() {
2     // ...
3
4     fun take(): T {
5         assert(Thread.currentThread() == this)
6         v.loop { cur ->
7             if (cur == null) {
8                 LockSupport.park()
9                 if (interrupted()) {
10                     throw InterruptedException()
11                 }
12             return@loop // continue
13         }
14     }
15 }
```

```

13         }
14         if (c.v.compareAndSet(cur, null)) return cur
15     }
16 }
17 }
```

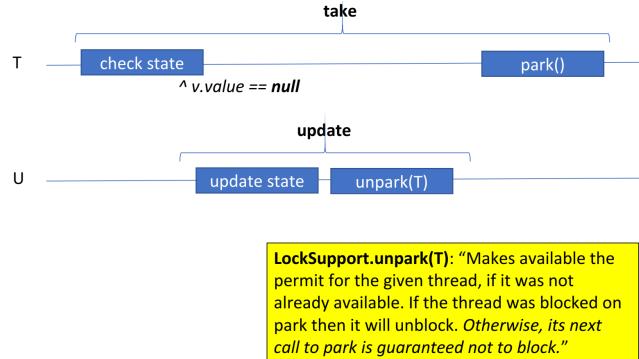


Рис. 61: Магия park/unpark

## 10.12 Ожидание из многих потоков

- Нужна очередь ждущих потоков
  - Нетривиально написать
- Возьмём готовую
- **java.util.concurrent.lock.AbstractQueuedSynchronizer** (рисунки 62 и 63)
  - **ReentrantLock**
  - **ReentrantReadWriteLock**
  - **Semaphore**
  - **CountDownLatch**

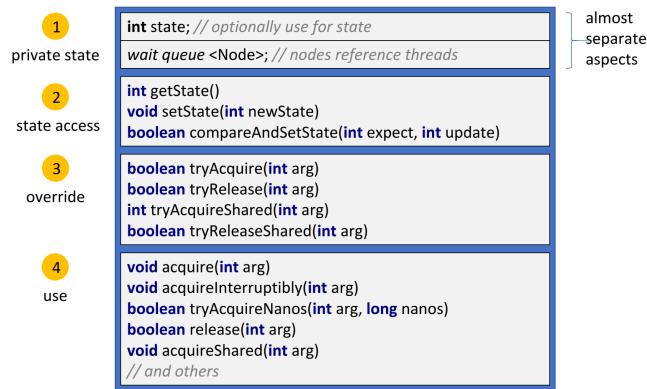


Рис. 62: Анатомия AbstractQueuedSynchronizer(1)

### 10.12.1 Пишем свой внутренний синхронайзер

```

1 inner class Sync: AbstractSynchronizer() {
2     override fun tryAcquire(arg: Int): Boolean {
3         val cur = v.value ?: return false
4         if (!v.compareAndSet(cur, null)) return false
5         // need to return value there
```

```

public final void acquireInterruptibly(int arg)
    throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    if (!tryAcquire(arg))
        doAcquireInterruptibly(arg); ← 2 adds to
}                                             wait queue

public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h); ← 4 unlink from
        return true;
    }
    return false;
}

```

Рис. 63: Анатомия AbstractQueuedSynchronizer(2)

```

6     results[arg] = cur
7     return true
8 }
9
10 //always "release" —— wake up next thread
11 override fun tryRelease(arg: Int): Boolean = true
12 }

```

### 10.12.2 Используем его

```

1 private val sync = Sync()
2
3 fun update(item: T) {
4     v.value = item
5     sync.release(0) //send signal
6 }
7
8 fun take(): T {
9     val arg = reserveResultSlot()
10    sync.acquireInterruptibly(arg) //wait inner
11    //recheck for not lost unpark
12    if (v.value != null) sync.release(0)
13    return releaseResultSlot(arg)
14 }

```

### 10.12.3 Улучшаем производительность

- Идея будить поток только когда обновляем: **null** → значение
- Не работает очень тонким образом (рисунок 64)

Из-за **update** параллельный **tryAcquire** может обламаться на CAS и запарковаться, но так как мы больше не вызываем **release**, то никто его больше не разбудит

```

1 fun update(item: T) {
2     T old = v.getAndSet(item)
3     if (old == null) sync.release(0)
4 }

```

**Исправляем tryAcquire** Используем цикл и повторяем, если CAS упал

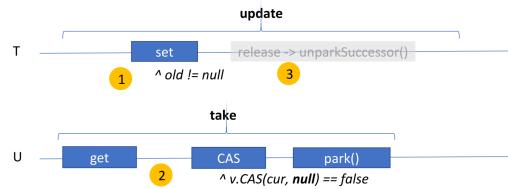


Рис. 64: Fail

```

1 override fun tryAcquire(arg: Int): Boolean {
2     while(true) {
3         val cur = v.value ?: return false
4         if (!v.compareAndSet(cur, null)) continue
5         results[arg] = cur
6         return true
7     }
8 }
```

## 11 FAA Based Queue

### 11.1 Fetch-And-Add

- FAA(adress, delta) — атомарно увеличивает значение на delta и возвращает старое значение
- FAA гораздо лучше масштабируется, чем CAS
- Modern queues use Fetch-And-Add

### 11.2 Obstruction-free queue on infinite array

- Бескнечный массив и указатели для enqueue и dequeue. Сначала увеличиваем индекс потом пишем читаем.
- Если dequeue придет читать раньше чем произошла запись, то мы пометим ячейку как сломанную и обе операции начнутся заново

```

1 fun enqueue(x: T) = while(true) {
2     val enqIdx = FAA(&this.enqIdx, 1)
3     if (CAS(&data[enqIdx], null, x))
4         return
5 }
6
7 fun dequeue(): T = while(true) {
8     if (isEmpty()) return null
9     val deqIdx = FAA(&this.deqIdx, 1)
10    val res = SWAP(&data[deqIdx], BROKEN)
11    if (res == null) continue
12    else return (T) res
13 }
14
15 fun isEmpty(): Boolean = deqIdx >= neqIdx
```

#### 11.2.1 Michael-Scott queue of array segments

```

1 fun enqueue(x: T) = while(true) {
2     val tail = this.tail
3     val enqIdx = FAA(&tail.enqIdx, 1)
4     if (enqIdx >= NODE_SIZE) {
5         //try to insert new Node with "x"
```

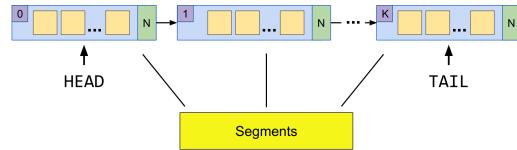


Рис. 65: MS queue of array segments

```

6      } else {
7          if (CAS(&data[enqIdx], null, x))
8              return
9      }
10 }
11
12 fun dequeue(): T = while(true) {
13     val head = this.head
14     if (head.isEmpty()) {
15         val headNext = head.next ?: return null
16         CAS(&this.head, head, headNext)
17     } else {
18         val deqIdx = FAA(&head.deqIdx, 1)
19         if (deqIdx >= NODE_SIZE) continue
20         val res = SWAP(&data[deqIdx], BROKEN)
21         if (res == null) continue
22         else return res
23     }
24 }
```

## 12 Counter Sharding

### 12.1 Naive counter

```

1 var counter = 0
2 fun inc() { FAA(&counter, 1) }
3 fun get(): Int = counter
```

- FAA берёт блокировку на уровне железа
- Часто вызываем inc  $\Rightarrow$  contention на поле **counter**

### 12.2 Sharding

- get() вызывается сильно реже inc()
  - Например, собирается какая-то performance метрика
- Разобьём на несколько мини счётчиков (рисунок 66)
  - **inc()** — увеличивает случайный из них
  - **get()** — суммирует значения всех мини счётчиков

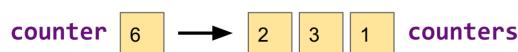


Рис. 66: Counter Sharding

## 13 Elimination technique

### 13.1 Naive Stack implementation

В наивной реализации создаётся слишком высокий contention на head

```
1 var head: Node = ...
2
3 fun push() {
4     ...
5     CAS(&head, ... )
6     ...
7 }
8
9 fun pop() {
10    ...
11 }
```

### 13.2 Elimination

Если выполняются много **push()** и **pop()** одновременно, то позволим им встретиться на "нейтральной территории" (рисунок 67). Используется старый алгоритм, если оптимизация не получилась.

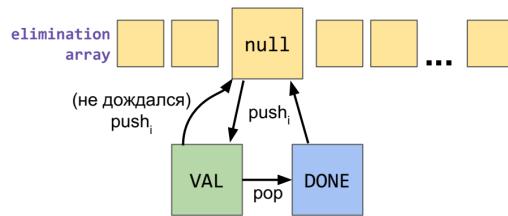


Рис. 67: Elimination

- Обе операции при выборе случайной ячейки пробуют посмотреть на соседние при неудаче
  - Это "дёшево" т.к. они скорее всего закешированы
- На практике elimination должен быть адаптивным, чтобы не тратить время на эту "оптимизацию" впустую и выбрать наиболее оптимальный размер массива

## 14 Железо и спин-локи

### 14.1 Работа с общей памятью

В простом случае работа происходит через общую шину напрямую с памятью, поэтому если один поток успел записать значение, то другой его увидит сразу текущее. Но, так как прямая связь дорогая и долгая, ядро добавляют собственную память, которая называется кеш. Все взаимодействие с памятью происходит через него: когда мы читаем, то вначале пытаемся из кеша, и лишь при неудаче читаем из памяти. Поэтому, когда другой поток обновит значение, мы этого сразу не увидим. Это ломает многопоточность. Решение - протокол MESI.

### 14.2 Протокол MESI

#### 14.2.1 MESI → MOSI (O = Owned)

- MESIF = MESI + Forward (Intel)
- MOESI = MESI + Owned (AMD)

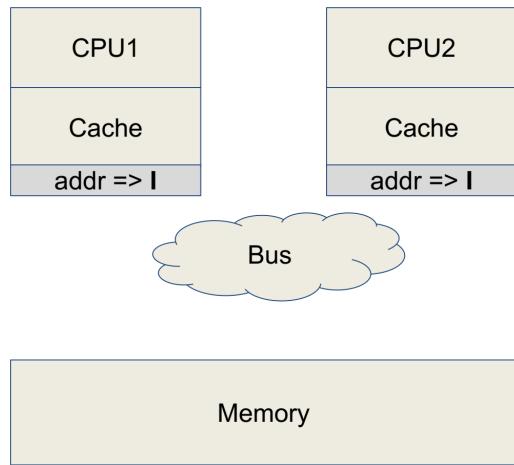


Рис. 68: MESI start

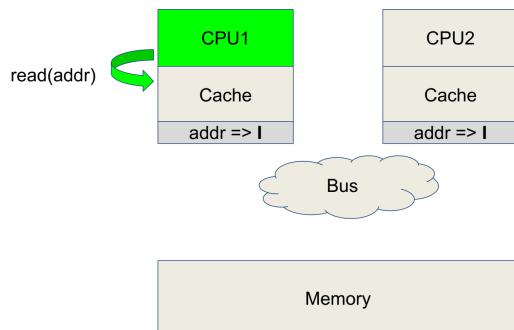


Рис. 69: MESI Exclusive(1)

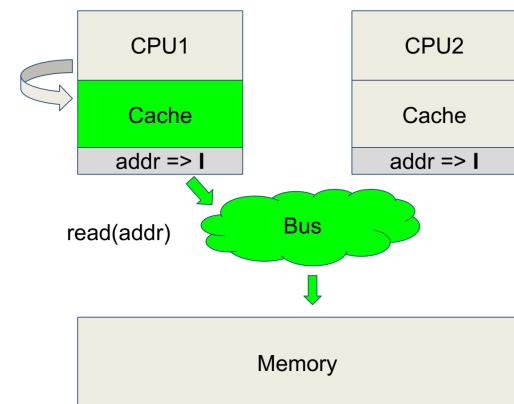


Рис. 70: MESI Exclusive(2)

### 14.3 Test-And-Set (aka get-And-Set)

```

1 class TASRegister :
2     shared r
3
4     def getAndSet(x): atomically do
5         old = r
6         r = x
7         return old
8
9     def write(x):
10        r = x
11

```

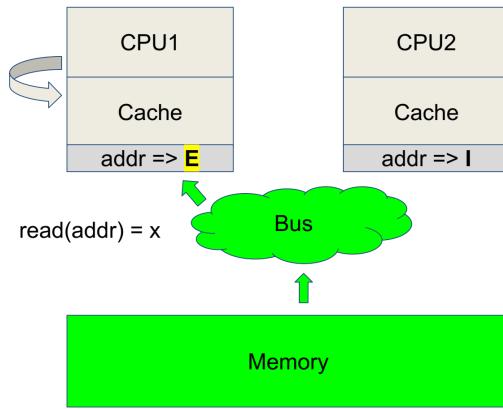


Рис. 71: MESI Exclusive(3)

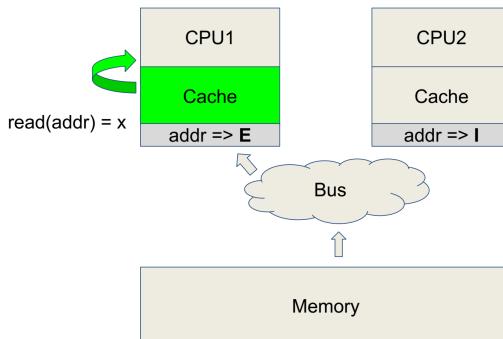


Рис. 72: MESI Exclusive(4)

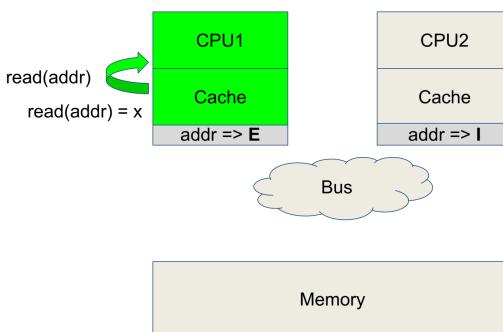


Рис. 73: MESI Exclusive(5)

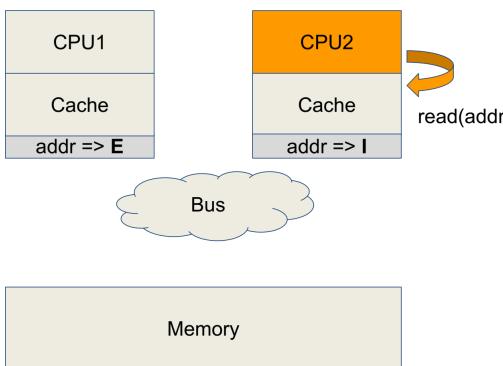


Рис. 74: MESI Shared(1)

```
12     def read():
13         return r
```

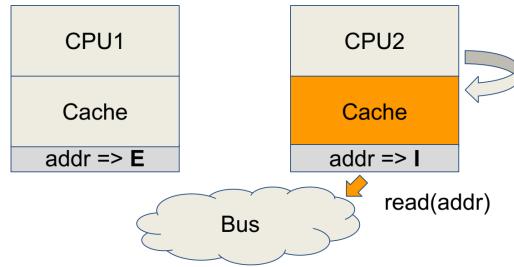


Рис. 75: MESI Shared(2)

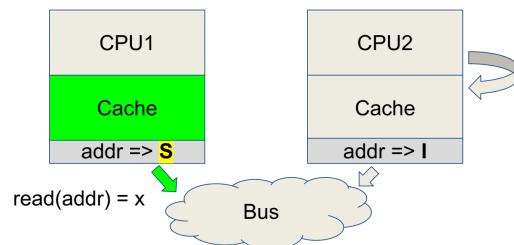


Рис. 76: MESI Shared(3)

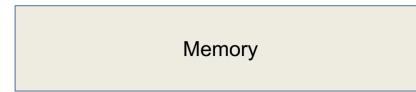
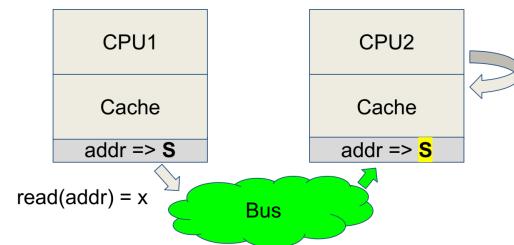


Рис. 77: MESI Shared(4)

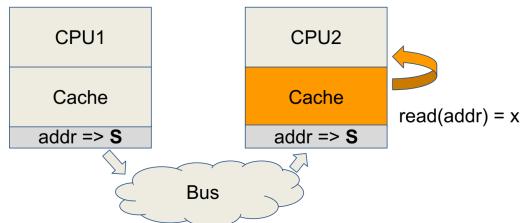


Рис. 78: MESI Shared(5)

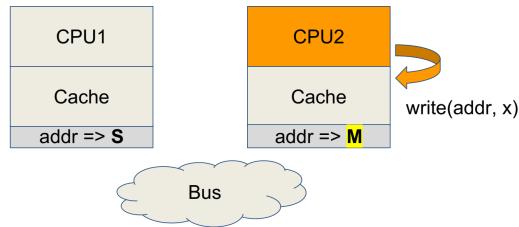


Рис. 79: MESI Modified(1)

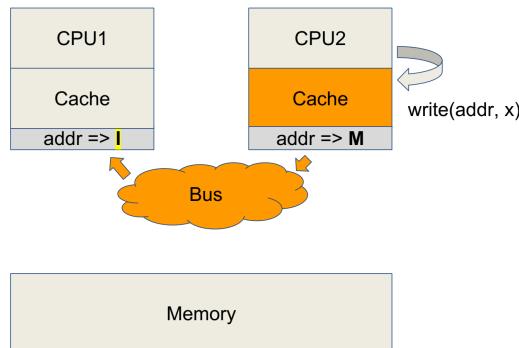


Рис. 80: MESI Modified(2)

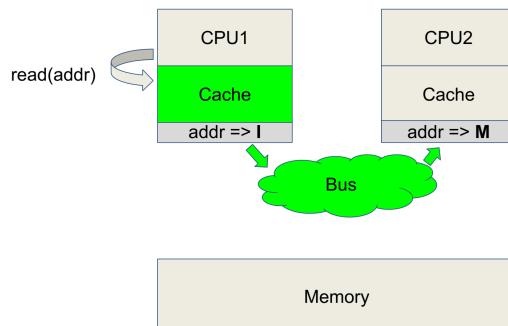


Рис. 81: MESI Flush(1)

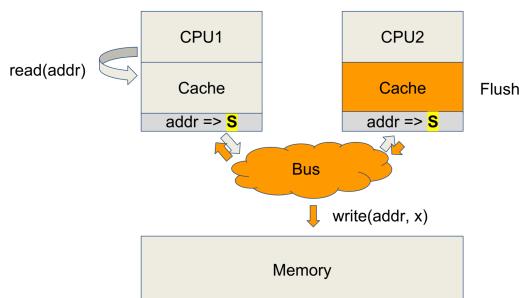


Рис. 82: MESI Flush(2)

#### 14.3.1 lock

```

1 class Lock:
2     boolean locked
3
4     def lock():

```

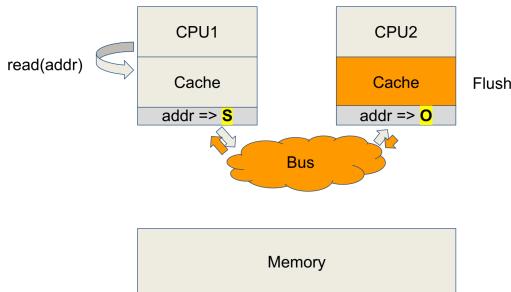


Рис. 83: MOSI

```

5     while locked .getAndSet( true ):
6         pass
7
8     def unlock():
9         locked = false

```

Цикл в **lock()** каждый раз перезаписывает `locked`, что вызывает состояние Exclusive и сброс всех кешей — не оптимально.

```

1 def lock():
2     loop:
3         while locked: pass
4         if !locked .getAndSet( true ):
5             break

```

В данном варианте while крутиться без шума в состоянии Shared. Но когда блокирующий поток вызывает **unlock()** все начинают ломиться в **getAndSet(true)**, что снова создаёт большую нагрузку и состояние Exclusive.

### Backoff

```

1 def lock():
2     loop:
3         while locked: pass
4         if !locked .getAndSet( true ):
5             break
6         delay()

```

Свойства `delay()`:

1. Случайное время
2. Экспоненциально увеличивать
3. Настройка под платформу

`delay()` - рандомизированный, т.к. если несколько потоков ломанулись в блокировку и одному повезло, то он может очень быстро выполнить операцию и без делёя все остальные потоки снова ломаются, это будет плохо, а если будет рандомизированный делэй, то кому-то из потоков повезет и он будет маленькое количество времени спать, и быстро схватит блокировку. Это очень важно, если у нас короткая блокировка.

Но этот алгоритм был нечестным, так как поток мог прийти последним и получить блокировку первым.

### 14.4 CLH Lock

- Travis Craig, Andres Landin, Erik Hagerstern
- Устраним лишние инвалидации
- Храним очередь ждущих потоков
- First-Come First-Served



**Не занято**

Рис. 84: Начало: не занято

Thread 1      Поток хочет lock



Рис. 85: Поток 1

Thread 1

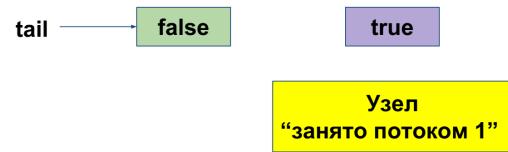


Рис. 86: Узел 1

Thread 1

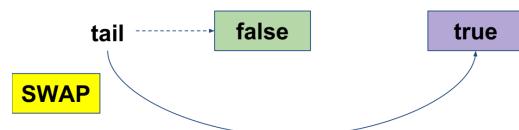


Рис. 87: Захват блокировки

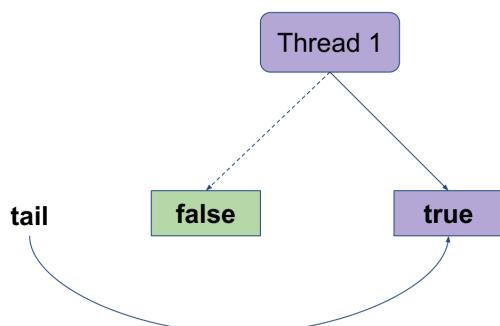


Рис. 88: Занято 1-м потоком

## 14.5 NUMA

Non-Uniform Memory Access

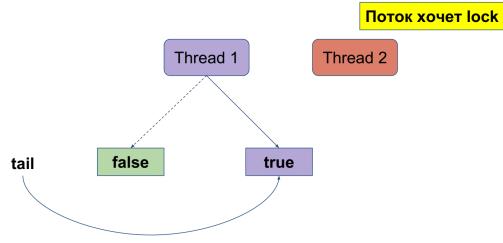


Рис. 89: Поток 2

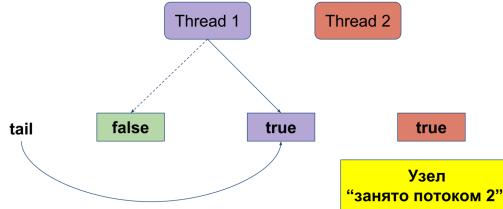


Рис. 90: Узел 2

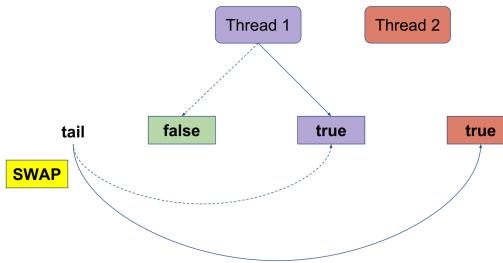


Рис. 91: Добавление в очередь

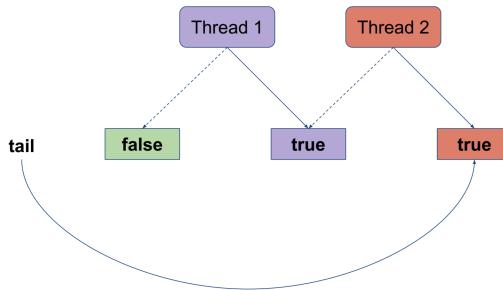


Рис. 92: Поток 2 в очереди

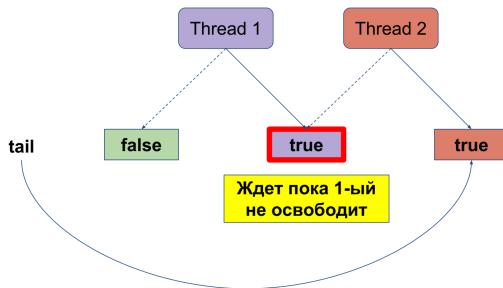


Рис. 93: Поток 2 ждёт (1)

CLH Lock:

- Ждём на "чужой памяти"
- Она может быть "далеко"

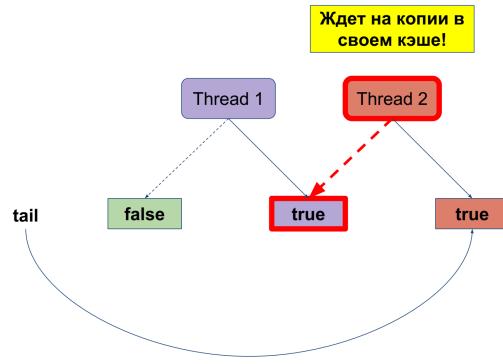


Рис. 94: Поток 2 ждёт (2)

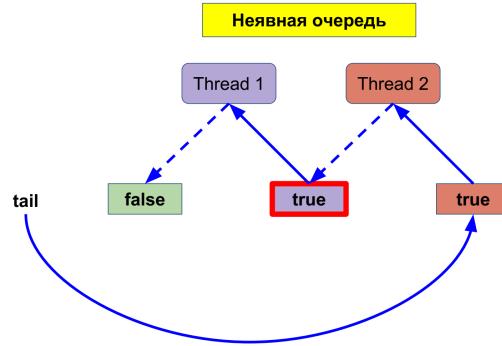


Рис. 95: Очередь

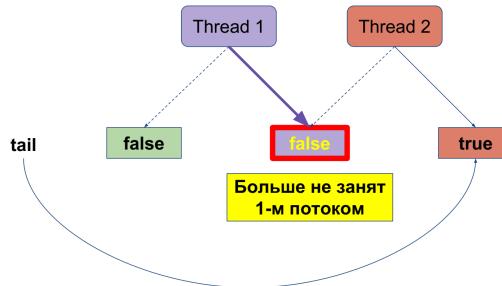


Рис. 96: Поток 1 освобождает

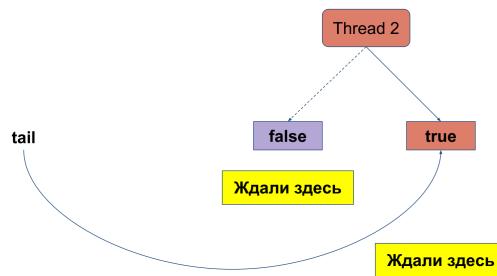


Рис. 97: Занято 2-м потоком (1)

## 14.6 MCS Lock

- Ждём на своей памяти
- First-Come First-Served

Isolation - пока транзакция работает, другие не должны видеть несогласованный результат Consistency - сотрудники не исчезают



Рис. 98: Занято 2-м потоком (2)

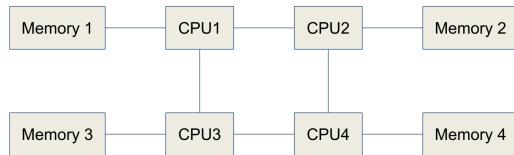


Рис. 99: NUMA



**Не занято**

Рис. 100: Начало: не занято



Рис. 101: Поток 1

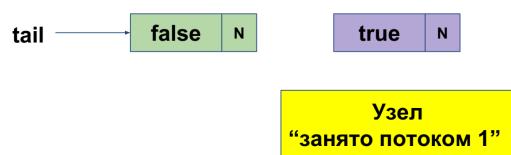


Рис. 102: Узел 1



Рис. 103: Захват

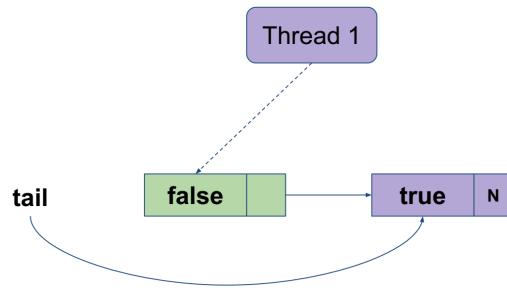


Рис. 104: Создали явный список

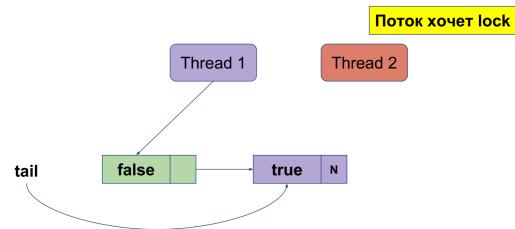


Рис. 105: Поток 2

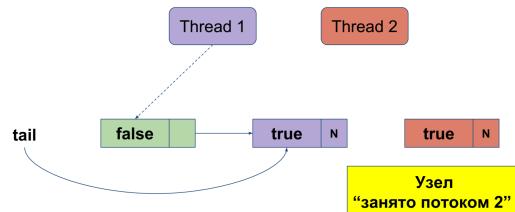


Рис. 106: Узел 2

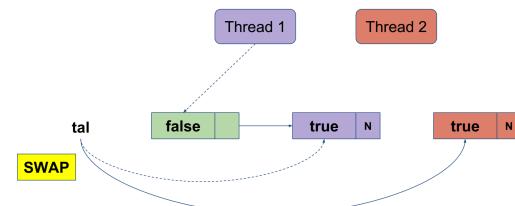


Рис. 107: Добавление в очередь

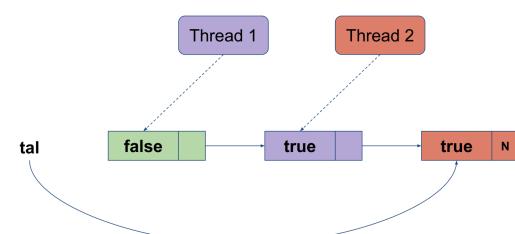


Рис. 108: Создание списка

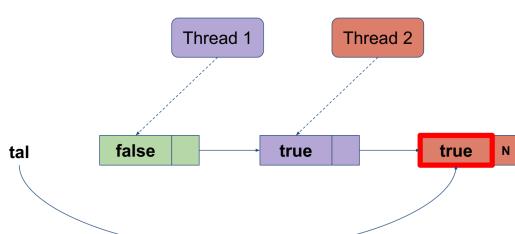


Рис. 109: Ждём на своём объекте

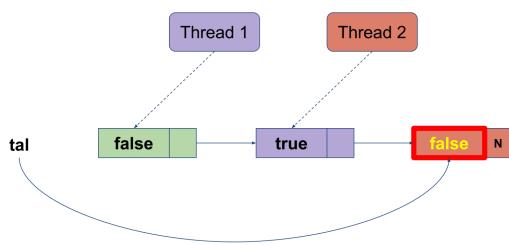


Рис. 110: Поток 1 освобождает