

Конспект по курсу Параллельное программирование ¹

Александра Лисицына ²

27 декабря 2018 г.

¹Читаемый Романом Елизаровым Никитой Ковалем в 2018-2019 годах

²Студентка группы М3334

Содержание

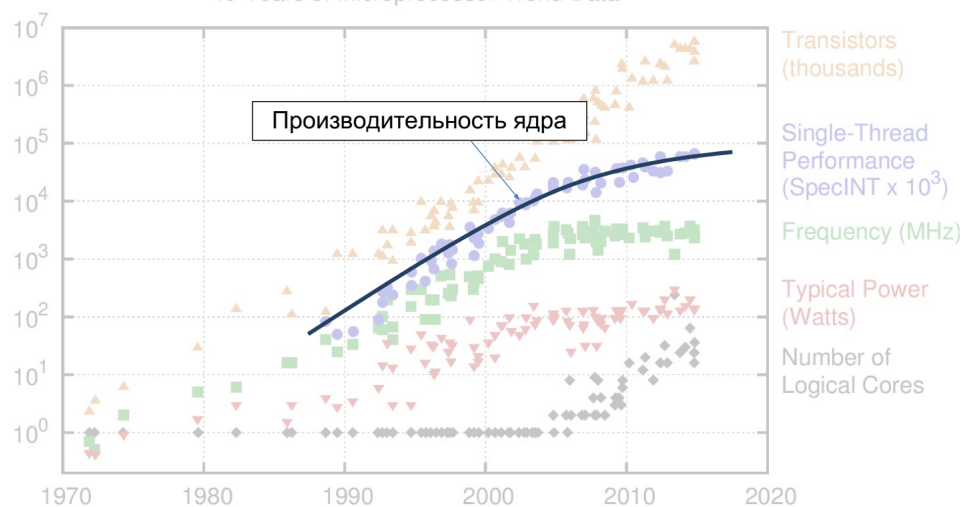
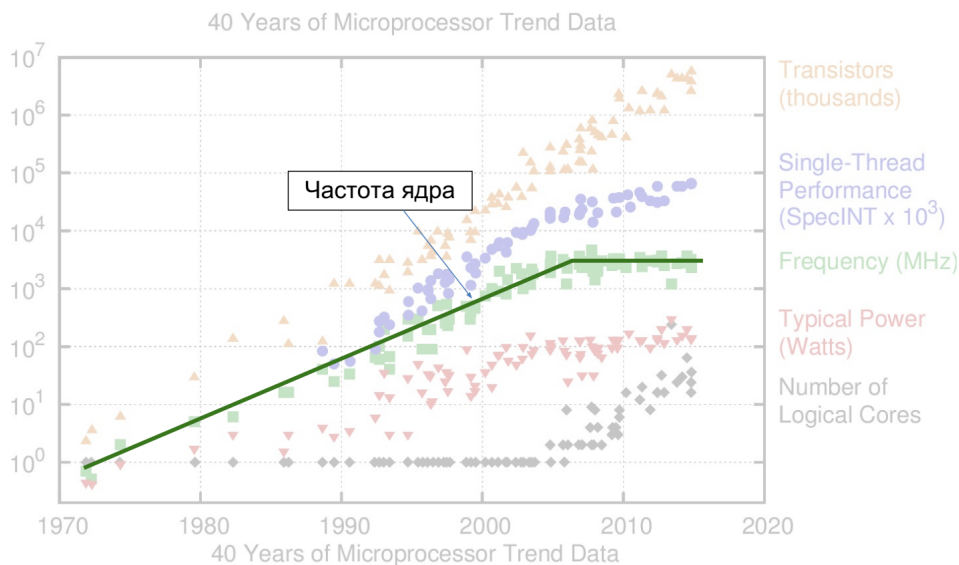
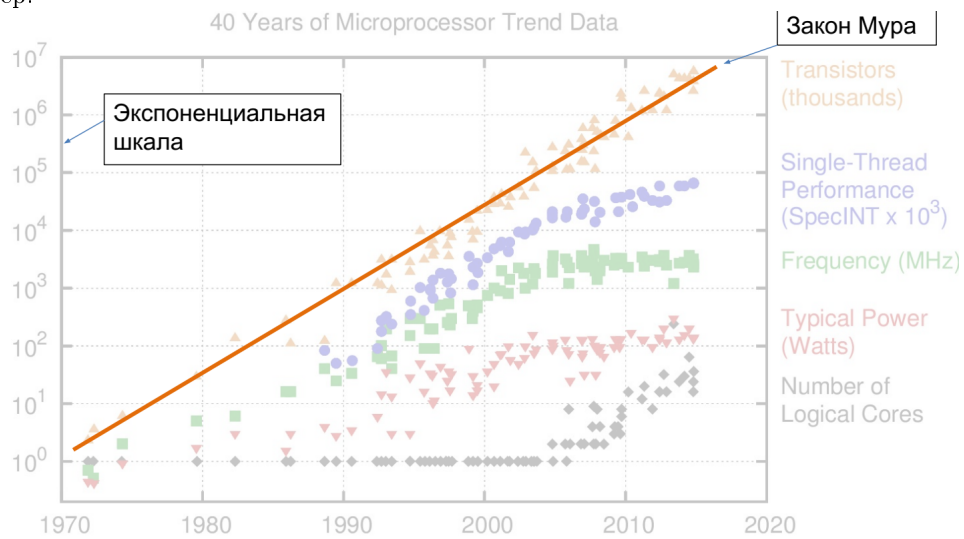
| | | |
|----------|--|-----------|
| 1 | Intoduction | 3 |
| 1.1 | Закон Мура | 3 |
| 1.2 | Закон Амдала | 4 |
| 1.3 | Разные виды параллелизма | 4 |
| 1.3.1 | Параллелизм на уровне инструкций (ILP) | 4 |
| 1.4 | Операционные системы | 6 |
| 1.5 | Основные понятия в современных ОС | 6 |
| 1.6 | Формализм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

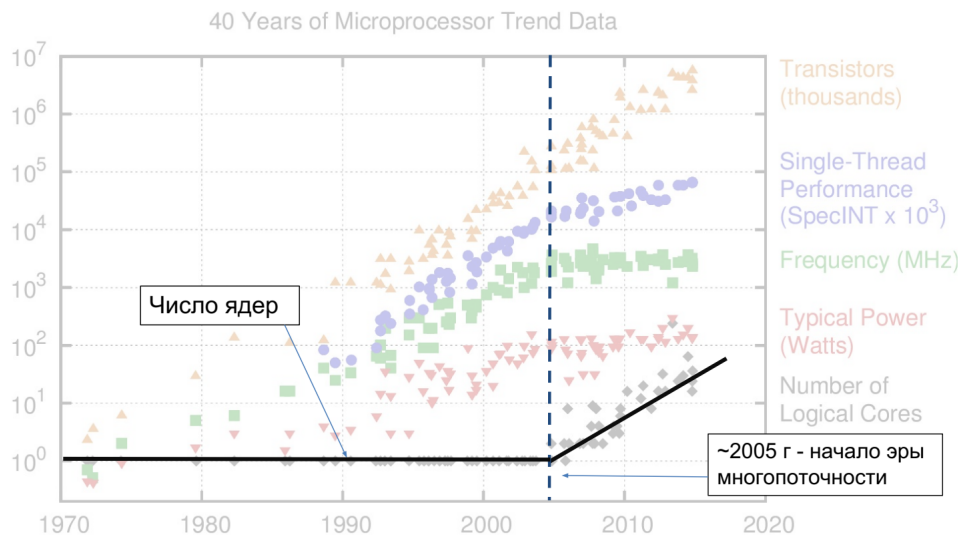
| | | |
|-----------|---|-----------|
| 8 | CASN | 17 |
| 8.1 | Списки против массивов | 17 |
| 8.2 | CASn | 18 |
| 8.2.1 | DCSS | 18 |
| 8.3 | Наблюдения и замечания | 18 |
| 8.4 | Подход к реализации | 18 |
| 9 | Мониторы и ожидание | 18 |
| 9.1 | Объекты как функции | 18 |
| 9.2 | Очередь ограниченного размера с ожиданием | 18 |
| 9.3 | Лианеризуемость операций с ожиданием | 18 |
| 9.4 | Реализация через монитор | 18 |
| 9.5 | Монитор в Java | 18 |
| 9.6 | ReentrantLock | 19 |
| 9.7 | Подробнее про interrupt | 19 |
| 9.8 | Ожидание без блокировки | 19 |
| 10 | Железо и спин-локи | 19 |
| 10.1 | Backoff | 19 |

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 $1/(1 - P)$

| P | S |
|-----|-----|
| 60% | 2.5 |
| 95% | 20 |
| 99% | 100 |

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 исполнение¹
 - Внеочередное исполнение
 - Переименование регистров²
 - Спекулятивное исполнение³
 - Предсказание переходов
- Длинное машинное слово (VLIW⁴)
- 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

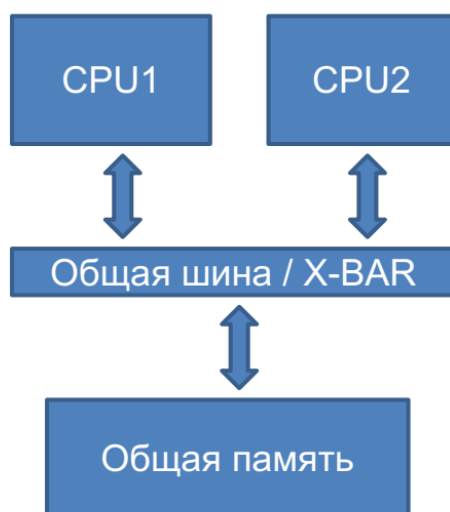


Рис. 2: SMP

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

⁰Instruction Level Parallelism

¹Несколько операций за такт

²Чтобы не возникало ложной зависимости по регистрам

³Начинает выполнять одну из веток перехода, пытаясь ее предсказать

⁴Very Long Instruction Word

⁴Symmetric Multiprocessing

⁴Simultaneous Multithreading

⁴Non-uniform memory access

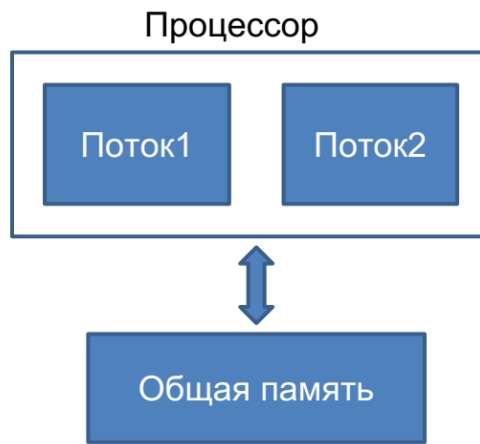


Рис. 3: SMT

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

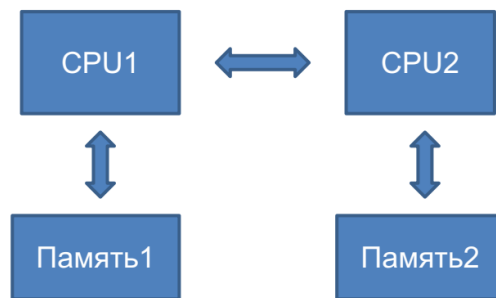


Рис. 4: NUMA

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)

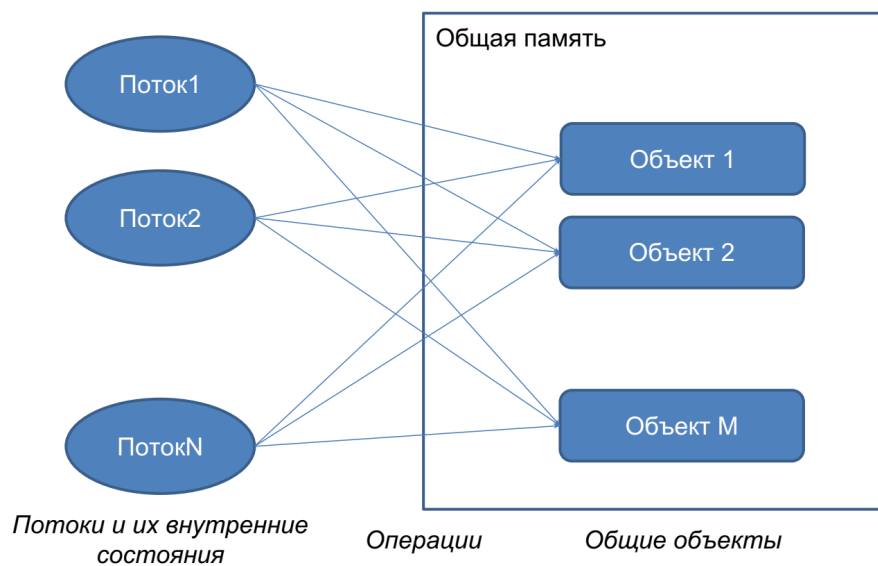


Рис. 5: Модель с общими объектами (общей памятью)

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 А имеет свойство Р» если она имеет это свойство при любом исполнении

1.6.5 Моделирование многопоточного исполнения

```
1      shared int x = 0, y = 0
```

Thread P:

Thread Q:

```
1      x = 1
2      r1 = y
3      stop
```

```
1      y = 1
2      r2 = x
3      stop
```

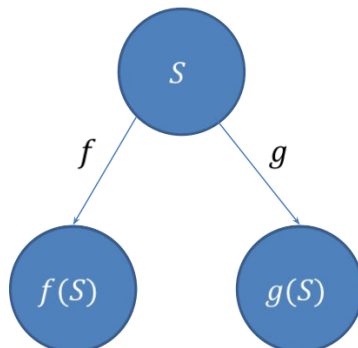


Рис. 6:

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

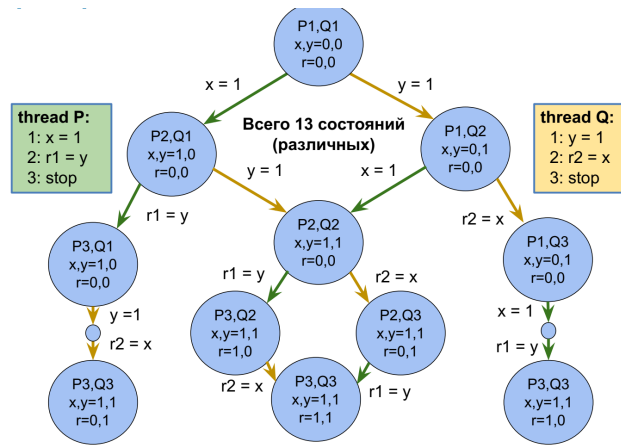


Рис. 7:

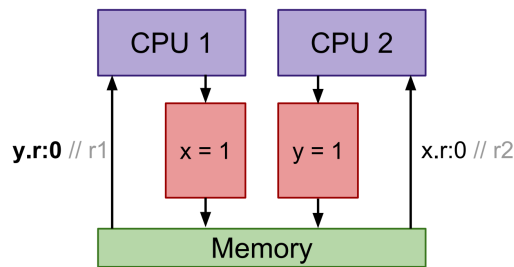


Рис. 8:

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 распространится 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 событиях.
 - 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

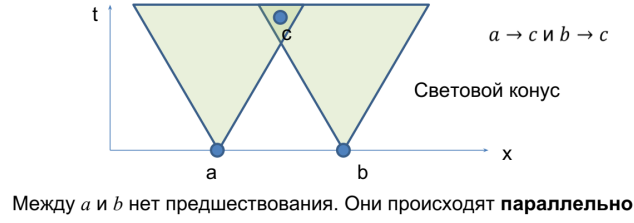


Рис. 9:

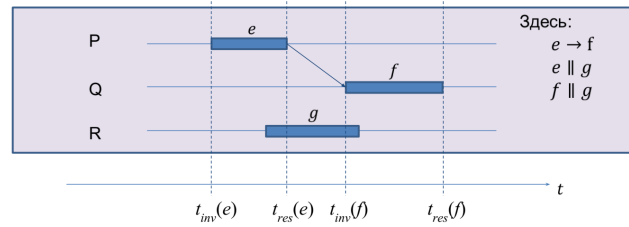


Рис. 10: Модель глобального времени

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, асимметричное 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 P», если каждое исполнение системы имеет свойство P.

3.3 Модель глобального времени

В этой модели каждая операция — это временной 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

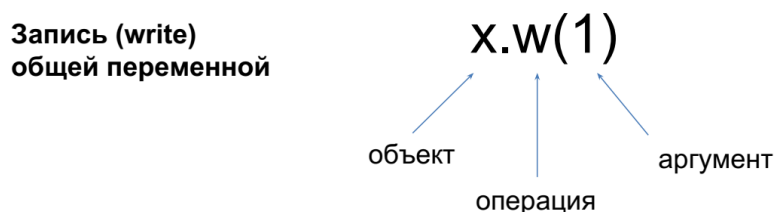


Рис. 11: Запись

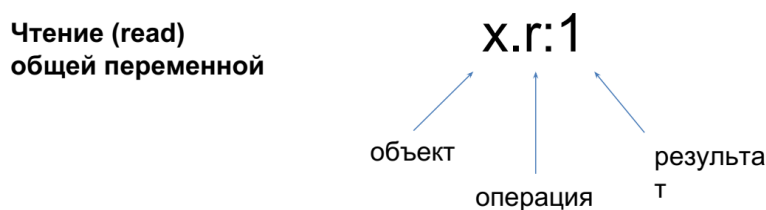


Рис. 12: Чтение

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.

•

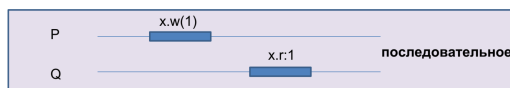


Рис. 13: Последовательное исполнение

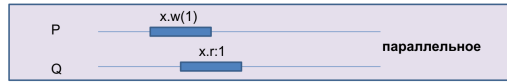


Рис. 14: Параллельное исполнение

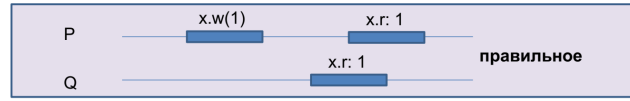


Рис. 15: Правильное исполнение

Определение. Если две конфликтующие операции произошли параллельно, то такая ситуация называется *гонка 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 e 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).

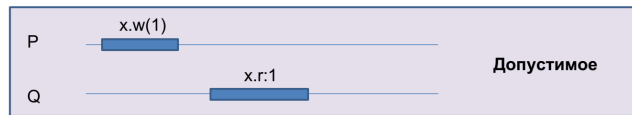


Рис. 16: Допустимое исполнение

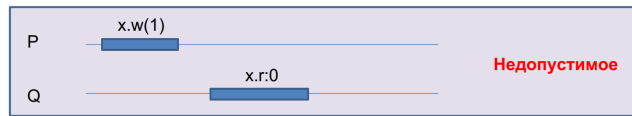


Рис. 17: Недопустимое исполнение

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 согласованно*, если ему можно сопоставить эквивалентное ему допустимое исполнение, сохраняющее его программный порядок (рисунок 18).

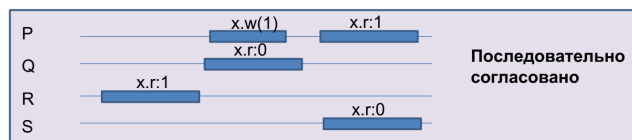
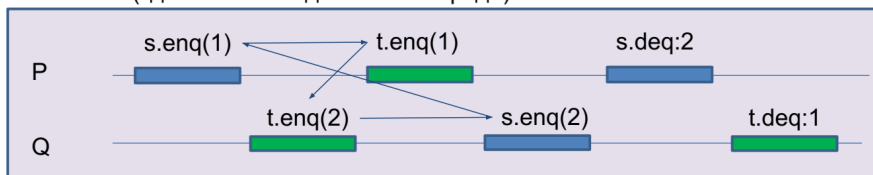


Рис. 18: Последовательно согласованно

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

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

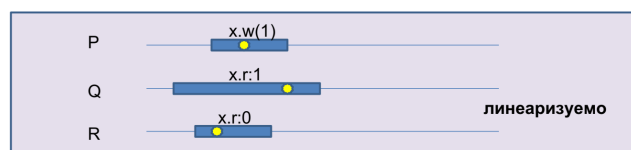


Рис. 20: Лианеризуемость

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 синхронизированна, 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 — $(H, G, \rightarrow_G, inv, res)$
 - H — это множество операций ($\forall e \in H: e \subset G$)
 - G — это множество событий
 - \rightarrow_G — отношение «произошло до» на событиях из G .
 - inv, res — функции из H в G , такие что:

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

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

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 C и атомарных регистров можно реализовать консенсусный протокол без ожидания с помощью детерминированного алгоритма для N потоков (и не больше), то говорят, что у класса C 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 остановятся 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

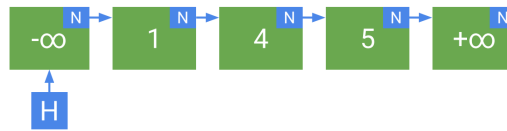


Рис. 21: Односвязный список

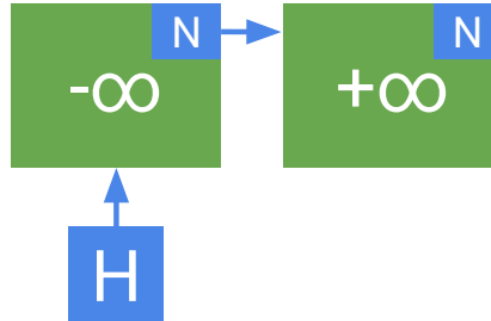


Рис. 22: Пустой список

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 $cur.KEY < key \leq next.KEY$ 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 перенаправлением ссылок: например, мы успели удалить *cur* перед тем как переназначить ссылки и у нас возникнет `NullPointerException` или список просто разорвется

7.2 Грубая синхронизация

- Coarse-grained locking
- Используем общую блокировку для всех операций
- \Rightarrow 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 \Rightarrow атомарно
- $\forall k$: операции с ключом *k* лианеризуемы \Rightarrow 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. Проверит инвариант *cur.N = next*
4. Проверить, что *cur* не удалён
5. Выполнить операцию
6. При любой ошибке начать занаво

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 одним успешным CAS-ом
- В Java для этого есть AtomicMarkableReference

8 CASN

8.1 Списки против массивов

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

8.2 CASn

```
1 class Ref<T>(initial: T) {  
2     var _v  
3     \\ ...  
}
```

8.2.1 DCSS

```
1 fun <A,B> dcsc(  
2     a: Ref
```

8.3 Наблюдения и замечания

8.4 Подход к реализации

9 Мониторы и ожидание

9.1 Объекты как функции

- Операция над объектом как функция
- Ранее операции были всюду определены на паре (S, P)

9.2 Очередь ограниченного размера с ожиданием

9.3 Лианеризуемость операций с ожиданием

- Расширим понятие исполнения
 - Есть событие вызова $inv(A)$
 - Но не обязателен ответ $resp(A)$

*

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

9.4 Реализация через монитор

Monitor = mutex + conditional variables

-

9.5 Монитор в Java

В Java каждый объект имеет монитор с одной условной переменной

В некоторых системах wait() реализован как хэш-таблица, при чем при коллизии кого-нибудь разбудим.

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.

9.6 ReentrantLock

9.7 Подробнее про interrupt

9.8 Ожидание без блокировки

10 Железо и спин-локи

Так как у каждого ядра есть кэш, то он читает значение не из глобальной памяти, а из кэша, что ломает нам многопоточность. Решение - протокол MESI

10.1 Backoff

delay() - рандомизированный, т.к. если несколько потоков лопанулись в блокировку и одному повезло, то он может очень быстро выполнить операцию и без дела все остальные потоки снова лопанутся, это будет плохо, а если будет рандомизированный delay, 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.

Напишем алгоритм с first-come-first-served : CLH Lock

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