ТЕХНОПОЛИС

Advanced Java. Многопоточность

Галкин Александр Сергеевич

одноклассники
экосистема

ПОЛИТЕХ

Оглавление

Теория

Чтобы научиться программировать многопоточные программы, надо научиться думать в многопоточном стиле

Закон мура

Общая память

Произошло до

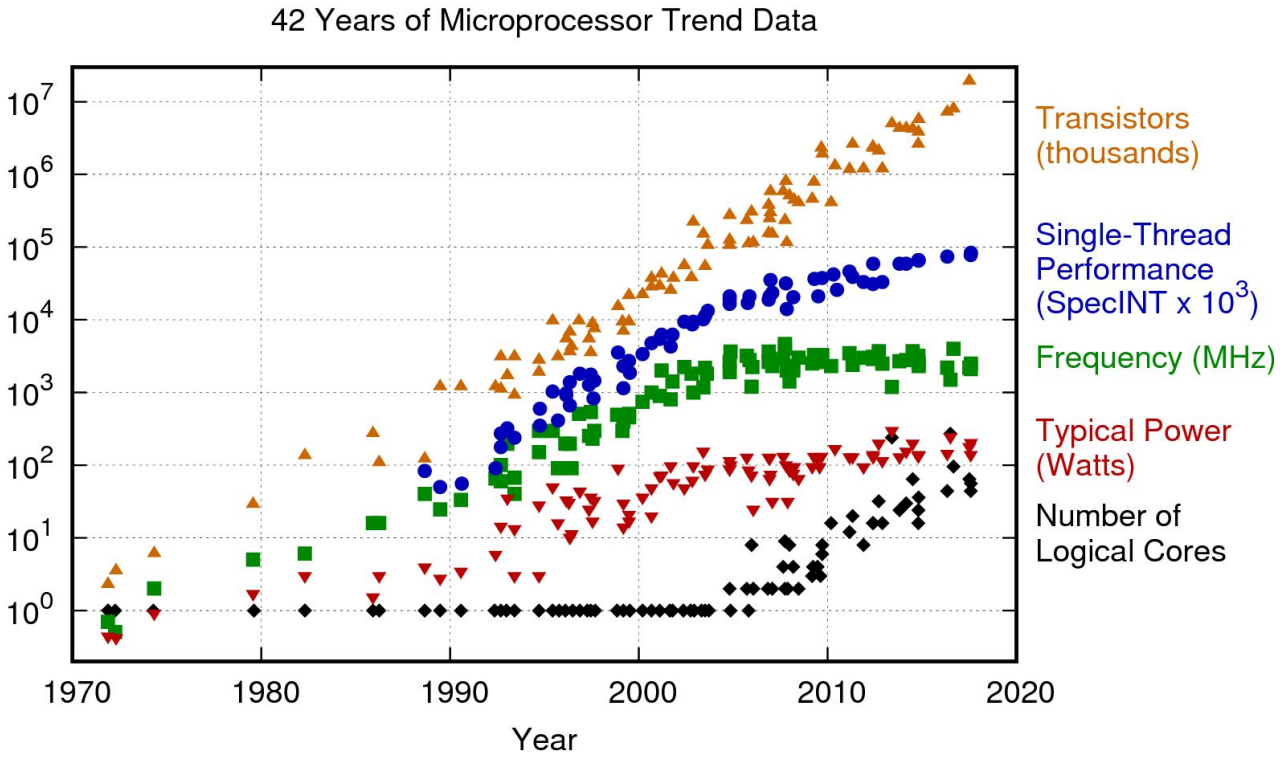
Глобально время

Согласованность

Линеаризуемость

Не все так просто, как нам кажется

Закон Мура



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Закон Мура

- Количество транзисторов, размещаемых на кристалле интегральной схемы, удваивается каждые 24 месяца.
- До 2005 года это достигалось при помощи одного ядра
- С 2005 года началась эра многопоточности. В одном процессоре помещают больше ядер => больше транзисторов

Подходы к описанию работы программы

- Однопоточный
  - Машина Тьюринга
  - Описываем сами алгоритмы и то что внутри них происходит
- Многопоточный
  - Различные модели
  - Во всех моделях нам не важно что делает один поток
  - Основная модель - общая память, описывается только взаимодействие потоков с общей памятью

Общие переменные

- Простейший тип объекта для общей памяти
- Имеют значение определенного типа
- Имеют операции чтения и записи
- Являются основой для параллельных алгоритмов
- Хорошая абстракция работы современных многопроцессорных систем

Общие переменные. Пример

- Две общие переменные ***int x = 0; int y = 0;***
- Поток P
  - ***x = 1*** — P1
  - ***r1 = y*** — P2
- Поток Q
  - ***y = 1*** — Q1
  - ***r2 = x*** — Q2

- ***S*** — общее состояние системы
  - Состояние всех потоков
  - Состояние всех общих объектов
- ***f, g*** — операции над общими объектами
  - Для переменных это операция чтения вместе с результатом, и операция записи вместе со значением
- ***f(S)*** — новое состояние системы, после применения операции ***f*** к состоянию ***S***
- От порядка операция зависит результат

Общие переменные. Теория

- P1 -> P2 -> Q1 -> Q2
  - ***r1 = 0***
  - ***r2 = 1***
- P1 -> Q1 -> P2 -> Q2; Q1 -> P1 -> Q2 -> P2
  - ***r1 = 1***
  - ***r2 = 1***
- P1 -> Q1 -> Q2 -> P2; Q1 -> P1 -> P2 -> Q1
  - ***r1 = 1***
  - r2 = 1
- Q1 -> Q2 -> P1 -> P2
  - ***r1 = 1***
  - ***r2 = 0***

Общие переменные. Пример, практика


```
@JCStressTest
@State
@Outcome(id = "0, 1", expect = Expect.ACCEPTABLE)
@Outcome(id = "1, 1", expect = Expect.ACCEPTABLE)
@Outcome(id = "1, 0", expect = Expect.ACCEPTABLE)
public class ConcurrencyTest {
    int x;
    int y;

    @Actor
    public void threadP(II_Result r) {
        x = 1;
        r.r1 = y;
    }

    @Actor
    public void threadQ(II_Result r) {
        y = 1;
        r.r2 = x;
    }

}
```

> Моя теория была очень ясна и правдоподобна - как и большинство ошибочных теорий.

Герберт Уэллс

Машина времени

- Запись в память в современных процессорах идет не мгновенно, а попадает в очередь на запись
- Другой поток не увидит эту запись, так как она реально еще не произошла

- Получилось, как будто в программе произошла перестановка (поменялись местами запись и чтение)
- Как учитывать оптимизации процессоров и компиляторов, не вникая в кишки?

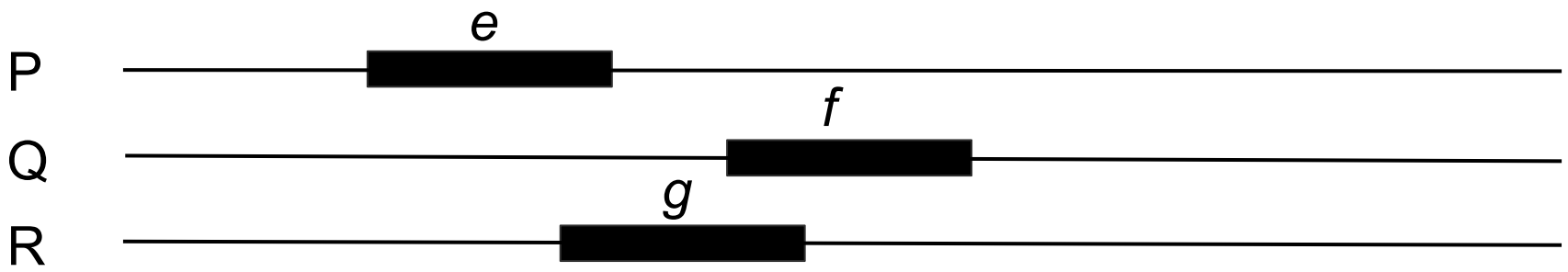
- Ожидание
  - Все операции происходят последовательно, но порядок не задан
- Реальность
  - Чтение и запись не мгновенные операции, они проходят параллельно даже внутри одного ядра
  - Физика: скорость света 3*10^8 м/с, за такт процессора с частотой 3 ГГц свет проходит 10 см в вакууме
  - Существует только частичный порядок (только про некоторые события можно сказать, что одно следует за другим)

Модель “произошло до”

- В 1978 г. впервые введена Лампортом
- Исполнение системы — это пара ***(H, ->)***
  - ***H*** — это множество операций произошедших во время исполнения
  - **->** — это транзитивное, асимметричное, антирефлексивное отношение (частично строгий порядок) на множестве операций
  - ***e -> f*** — означает, что e произошло до ***f*** в исполнении ***H***
- ***e*** и ***f*** параллельны, если e не произошло до ***f*** и наоборот
- Система — это набор всех возможных исполнений системы

Модель глобального времени

- В этой модели каждая операция это временной интервал
- Мы говорим что ***f -> e***, значит что конец ***f*** меньше начала ***e***

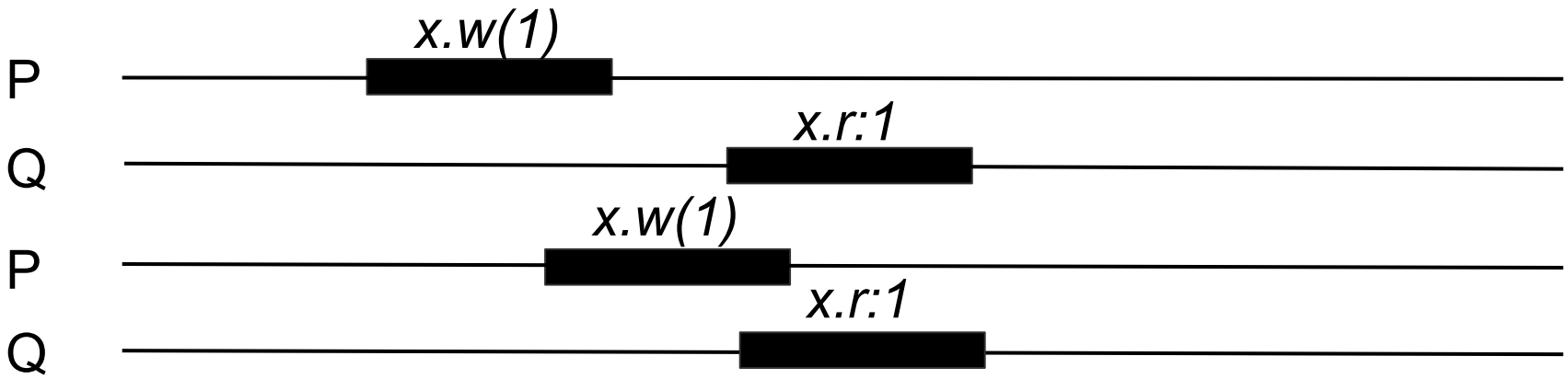


Модель глобального времени. Проблемы

- На самом деле, нет никакого глобального времени и быть не может (из-за физических ограничений)
- Это всего лишь механизм, с помощью которого можно визуализировать факт существования параллельных операций
- При доказательстве корректности алгоритма время не используется
- Но всегда можно нарисовать пример, который опровергает какую либо теорию

- Модель памяти языка программирования определяет то, каким образом исполнение операций синхронизаций создает отношение “произошло до”
  - Без них разные потоки выполняются параллельно
  - Можно доказать некоторые св-ва многопоточного кода, используя гарантии на возможные исполнения, которые дает модель памяти
- Различные операции синхронизации
  - Чтение и запись ***volatile*** переменных
  - Создание потоков и ожидание их завершения
  - библиотечные примитивы для синхронизации

- Запись: x.w(1)
- Чтение: x.r:1
- Исполнение системы называется последовательным, если все операции линейно-упорядочены отношением “произошло до”

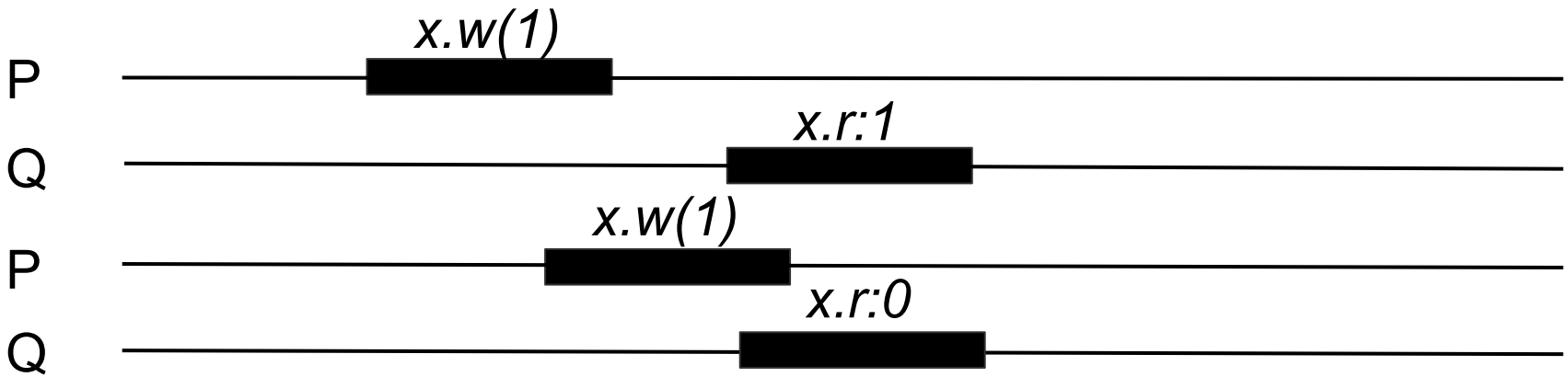


Определения: Правильное исполнение

- ***H|p*** — сужение исполнения на поток ***P***, т.е. исполнение, где остались только операции происходящие в потоке ***P***
- Исполнение называется правильным, если его сужение на каждый поток ***P*** является последовательным исполнением
- Это упрощение, на самом деле процессор умеет в одном ядре делать до 4-ех операций, но он это делает так, что вы этого не замечаете

- ***H|x*** — сужение исполнения на объект ***x***, т.е. исполнение, где остались только операции происходящие над объектом ***x***
- Если сужение на объект является последовательным, то можно проверить его на соответствие последовательной спецификации объекта
- Последовательная спецификация объекта, это то что пишется в документации, например чтение переменной должно вернуть последнее записанное значение

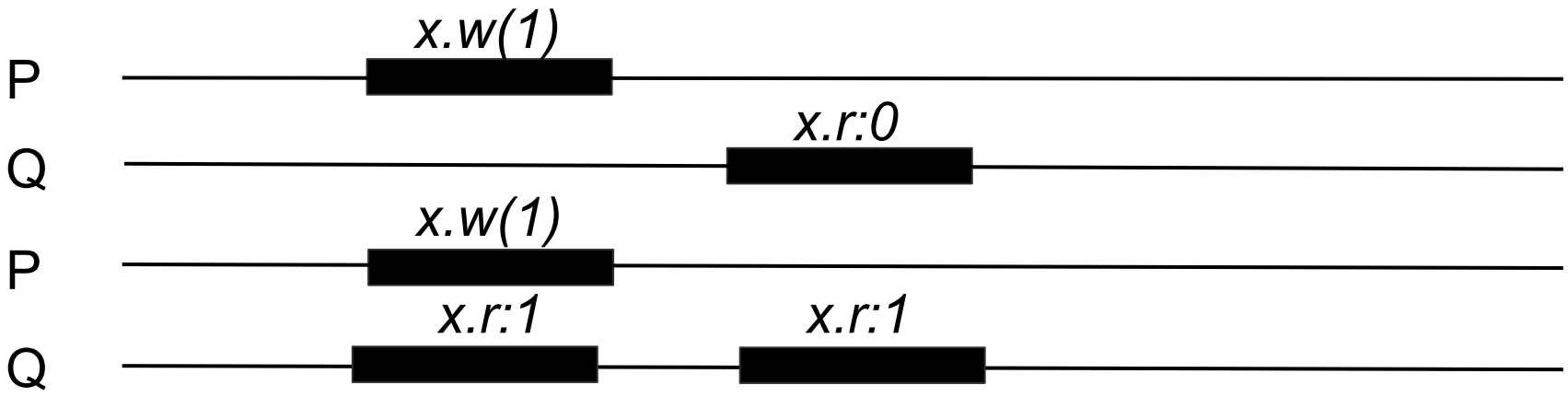
- Последовательное исполнение является допустимым, если выполнены последовательные спецификации всех объектов



Условие согласованности (корректности)

- Как определить допустимость параллельного исполнения?
  - Сопоставив ему эквивалентное (состоящее из тех же событий и операций) допустимое последовательное исполнение
  - Как именно - зависит от условия согласованности
- Основные условие согласованности
  - Последовательная согласованность
  - Линеаризуемость

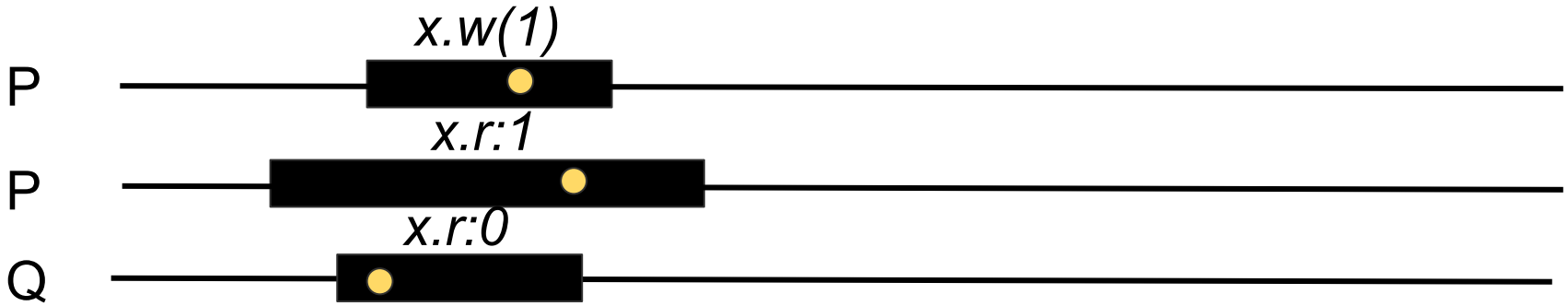
- Исполнение линеаризуемо, если можно сопоставить эквивалентное ему допустимое последовательное исполнение, которое сохраняет порядок “произошло до”



Свойства линеаризуемости

- В линеаризуемом исполнении каждой операции e можно сопоставить некое глобальное время, точку линеаризации, ***t(e)*** так, что время разных операций различно и если событие ***e -> f***, то ***t(e) < t(f)***
- Линеаризуемость локальна. Линеаризуемость исполнения на каждом объекте эквивалентна линеаризуемости исполнения системы целиком
- Операции над линеаризуемыми объектами называют атомарными

- В глобальном времени исполнение линеаризуемо тогда и только тогда, когда точки линеаризации могут быть выбраны так, что ***t(e)*** больше или равна началу операции и меньше или равна концу операции

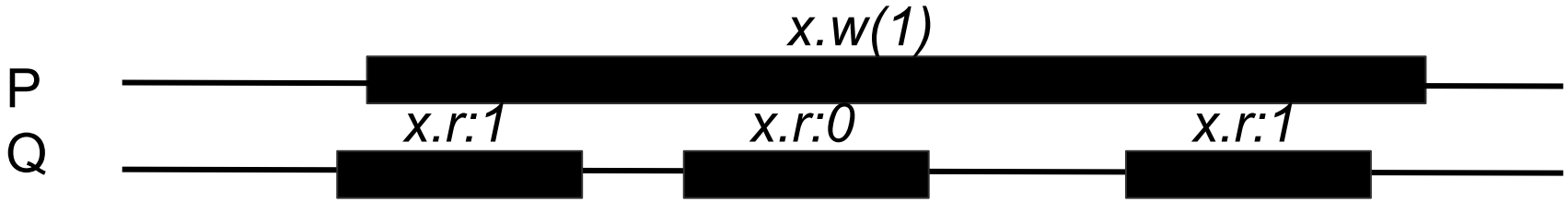


Исполнение системы, выполняющее операции над линеаризуемыми объектами, можно анализировать с помощью модели чередований

Иерархия линеаризуемых объектов

- Из простых линеаризуемых объектов можно делать линеаризуемые объекты более высокого уровня
- Доказав линеаризуемость сложного объекта, можно абстрагироваться от деталей реализации в нем, считать операции над ним атомарными и строить объекты более высокого уровня
- Когда говорят, что какой-то объект безопасен для использовании из нескольких потоков (thread-safe), то по-умолчанию имеют в виду линеаризуемость операций над ним

- В спецификации Java нигде не используется линеаризуемость
- Но там же написанно, что все операции над volatile полями являются операциями синхронизации, которые всегда линейно-упорядочены в любом исполнении и согласованы с точки зрения чтения и записи
- Не volatile поля без синхронизации могут нарушать это (и не только)



Проблемы многопоточного программирования

- Гонка (race condition)
- Взаимная блокировка (deadlock)
- Частая блокировка (livelock)

Практика

Чтобы научиться программировать многопоточные программы, надо научиться думать в многопоточном стиле

Потоки

synchronized

Многопоточные коллекции

Executor

Многопоточные примитивы

Fork Join Pool

Вам шашечки или ехать?

- Класс **java.lang.Thread** отвечает за потоки исполнения программы
- `String getName()` — запуск
- `long getId()` — запуск
- `StackTraceElement[] getStackTrace()` — запуск
- `boolean isDaemon()` — запуск
- Создать поток можно или унаследовавшись от класса ***Thread*** и переопределить метод run или передать в конструктор объект типа ***Runnable***

Жизненный цикл потока

- Создание
- `void start()` — запуск
- Работа (выполняется метода ***run***),
  метод `boolean isAlive()` выведет ***true***
- Завершение (метод ***run*** закончился или бросил исключение)

- Завершенный поток нельзя перезапустить

- Поток можно прервать с помощью метода `void interrupt()`
- Если поток находится в состоянии ожидания, то ожидание прервется исключением ***InterruptedException***
- Иначе у потока просто выставляется флаг ***interrupted***, который можно получить методом `boolean isInterrupted()`. Завершать поток надо самостоятельно
- Если мы хотим дождаться остановки потока, у которого вызвали метод ***interrupt***, то надо также вызывать метод `void join()`

- Взаимное исключении — пока один поток что-то делает, другие не могут ему помешать.
- Ожидание и уведомление — поток ничего не делает, пока ему не разрешили.

- Можно вешать на методы (синхронизация будет на текущем объекте или классе для статических методов)
- Синхронизированный блок внутри метода (синхронизация будет на переданном объекте)

```
private void doSomething1() {
    synchronized (obj) {
        doSomething();
    }
}
```

- Все методы реализованы в классе ***Object***
- Можно вызывать только внутри блока ***synchronized*** и только на том объекте на котором вызван монитор
- Методы ожидания
  - `void wait()`
  - `void wait(long timeout)`
- Методы пробуждения
  - `void notify()`
  - `void notifyAll()`

Произошло до в Java

- Чтение и запись для ***volatile*** полей
- Освобождение монитора происходит до его захвата следующим потоком
- Старт потока происходит до выполнения метода ***run***

- Синхронизированные обертки над основными коллекциями создаются через методы в классе ***Collections***

```
Collection<T>
synchronizedCollection(Collection<T> c)
```

- ***ConcurrentHashMap***
- ***ConcurrentSkipListMap*** -- аналог ***TreeMap***
- ***ConcurrentSkipListSet*** -- аналог ***TreeSet***
- ***CopyOnWriteArrayList***
- ***CopyOnWriteArraySet***

- Класс ***java.util.concurrent.ConcurrentLinkedQueue*** — очередь без блокировок, с поддержкой многопоточности
- Интерфейс ***java.util.concurrent.BlockingQueue*** — очередь с ожиданием, с поддержкой многопоточности (***LinkedBlockingQueue***, ***ArrayBlockingQueue***)

- Интерфейс ***java.util.concurrent.Executor*** — отвечает за создание и управление потоками
- Инкапсулирует не только создание потоков, но и очередь задач, которые передаются ему в методе

```
void execute(Runnable command)
```

- Более функциональный класс ***ExecutorService***

- `Future<?> submit(Runnable task)` — Ставит задачу в очередь на исполнение
- `Future<T> submit(Callable<T> task)` — Ставит задачу (с возвращаемым результатом) в очередь на исполнение
- `void shutdown()` — Ждет завершения текущих потоков и закрывает все ресурсы, не запускает потоки из очереди
- `List<Runnable> shutdownNow()` — Пытается закрыть работающие потоки, возвращает задачи из очереди
- ***Callable*** - аналог ***Runnable***, только единственный метод возвращает результат

- Класс-обертка над результатом вычисления
- `boolean isDone()` — Проверка на завершенную (по какой-либо причине) задачу
- `boolean isCancelled()` — Проверка на то, что задача была закрыта до того, как завершилось вычисление
- `V get()` — Блокирующая операция, выдающая завершение результата
- `boolean cancel(boolean mayInterruptIfRunning)` — Закрывает выполнение задачи

ExecutorService. Примеры

- Почти все сервисы создаются с помощью класса ***Executors***
- `ExecutorService newSingleThreadExecutor()`
  — Сервис только с одним потоком внутри себя
- `ExecutorService newFixedThreadPool(int nThreads)`
  — Сервис ровно с одним потоком
- `ExecutorService newCachedThreadPool()`
  — Количество потоков не ограничено, может переиспользовать потоки и убивать уже созданные

- В Java есть обертки над примитивами, которые являются потоко безопасными
- ***AtomicBoolean***, ***AtomicInteger***, ***AtomicLong***, ***AtomicReference<T>***
- В принципе, они бы просто заменяли ***volatile*** переменные, если бы не метод

```
boolean compareAndSet(int expect, int update)
```

- Класс ***java.util.concurrent.Semaphore*** ограничивает доступ к ресурсу
- В отличие от блока ***synchronized***, с объектом могут работать N потоков параллельно
- `void acquire(int permits)` — захват семафора
- `void release(int permits)` — освобождение семафора
- После захвата семафора, обязателен блок кода ***try-finally***, чтобы при любой ошибке мы освободили семафор

CountDownLatch

- Класс ***java.util.concurrent.CountDownLatch*** обеспечивает синхронизацию между N потоками
- `void await()` — Ждет, пока счётчик не станет равный ***0***
- `void countDown()` — уменьшает счетчик на ***1***
- Часто используется в тестировании многопоточных программ
- Может использоваться только один раз

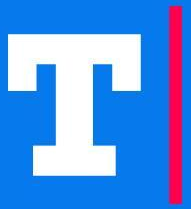
- Класс ***java.util.concurrent.CyclicBarrier*** обеспечивает синхронизацию между ***N*** потоками (многоразовый)

- Класс ***java.util.concurrent.locks.ReentrantLock*** обеспечивает синхронизацию между потоками, аналогично блоку ***synchronized***
- Более быстрый, позволяет блокировать и отпускать ресурс в разных точках программы
- Можно задать ожидание по аналогии FIFO
- `void lock()` — блокирует ресурс
- `void unlock()` — разблокирует ресурс

- Класс ***java.util.concurrent.locks.Condition*** обеспечивает возможность ждать и просыпаться для ***Lock***-ов
- На одном ***Lock***-е может быть несколько ***Condition***-ов
- Можно задать ожидание по аналогии FIFO
- `void signal()` — Пробуждаем ожидающие потоки
- `void await()` — Переводим поток в ожидание

- Класс **java.util.concurrent.locks.ReentrantReadWriteLock** Lock с разделением на читателей и писателей
- Много потоков могут одновременно читать
- Если кто-то пишет, другие потоки не имеют доступа к ресурсу

- Особый вид ***ExecutorService***, в котором декомпозиция задачи происходит внутри
- Задачу разделяем на подзадачи, потом на объединенных результатах выполняется еще одна задача
- Принимает внутри себя ***ForkJoinTask***
- `T invoke(ForkJoinTask<T> task)` — Добавляет в очередь на исполнение задачу
- `<List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)` — Добавляет в очередь на исполнение сразу несколько задач



Спасибо за внимание!