Concurrency

Маклашов Михаил

Терминология

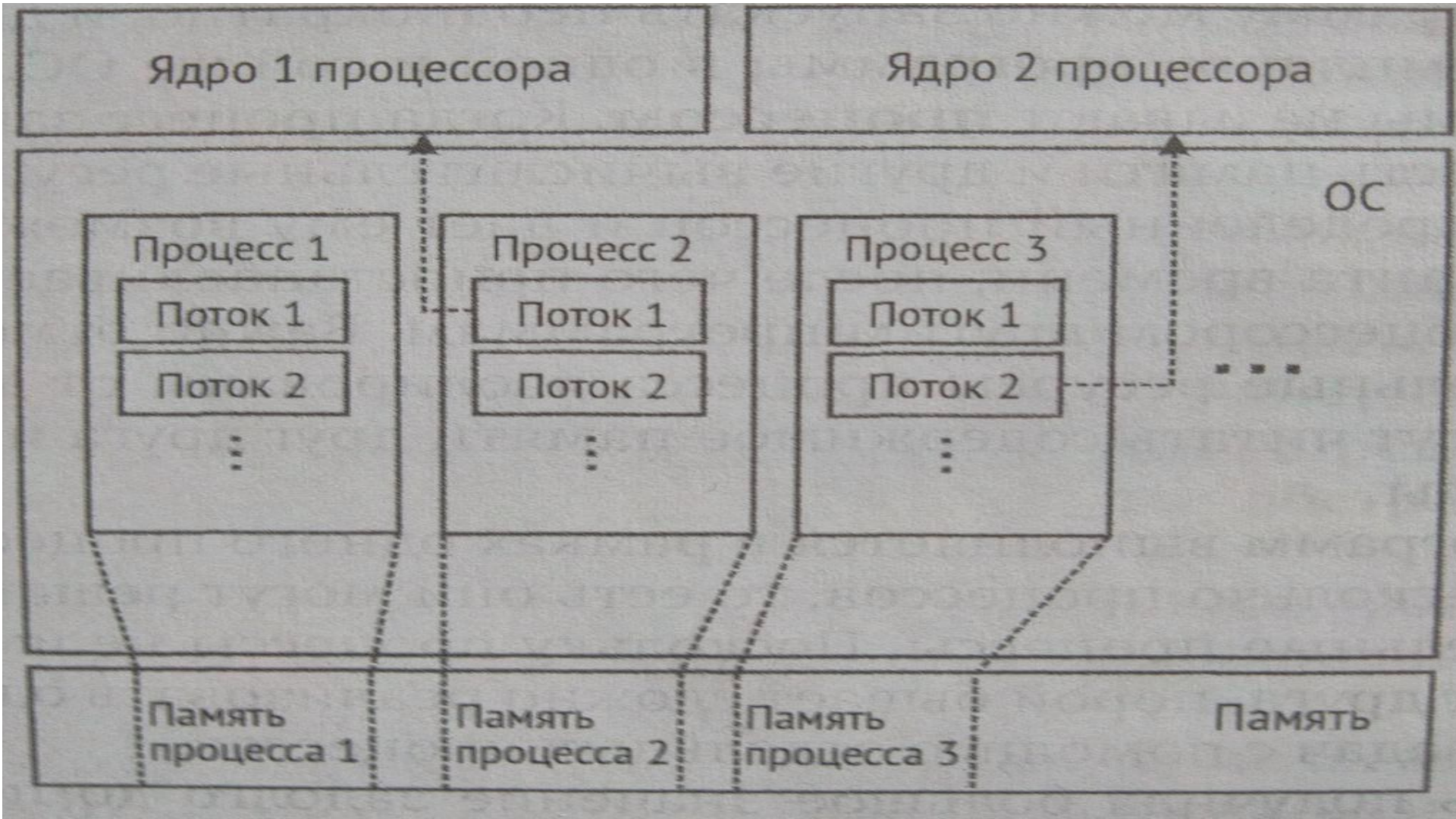
Каждый экземпляр программы называют процессом.

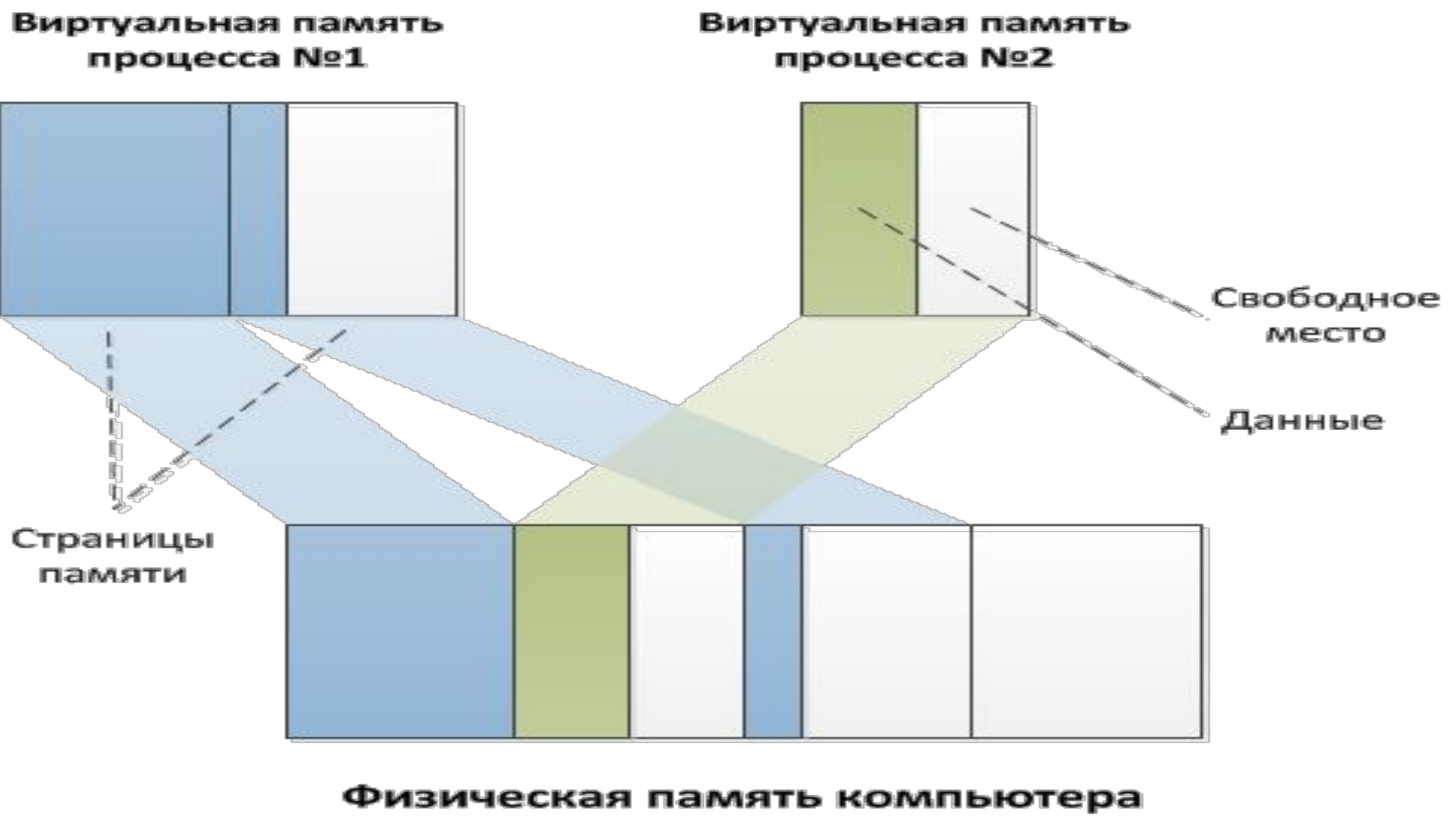
Поток - независимые вычисления в рамках одного процесса.

Каждый поток имеет имеет свой стек и указатель команд

Основы

- Каждый поток имеет свой стек вызовов
- У потоков общая память!
- Java-поток != поток в ОС
- У JVM свой планировщик потоков, который не зависит от планировщика операционной системы под которой работает JVM
- В Java есть два вида потоков: потоки-демоны (daemon threads) и пользовательские потоки (user threads)
- JVM завершает выполнение программы когда все пользовательские потоки завершат свое выполнение.


Ядро 1 процессора
Ядро 2 процессора
ОС
Процесс 1
Поток 1
Поток 2
Процесс 2
Поток 1
Поток 2
Процесс 3
Поток 1
Поток 2
Память процесса 1
Память процесса 2
Память процесса 3
Память


Виртуальная память процесса №1
Виртуальная память процесса №2
Свободное место
Данные
Страницы памяти
Физическая память компьютера

Потоки

- Класс Thread – поток
  - Позволяет создавать потоки и производить операции с ними
- Интерфейс Runnable – сущность, которая может быть запущена
  - public void run();

Взаимодействие потоков

```
Runnable runnable = () -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
};

runnable.run();

Thread thread = new Thread(runnable);
thread.start();

System.out.println("Done!");
```

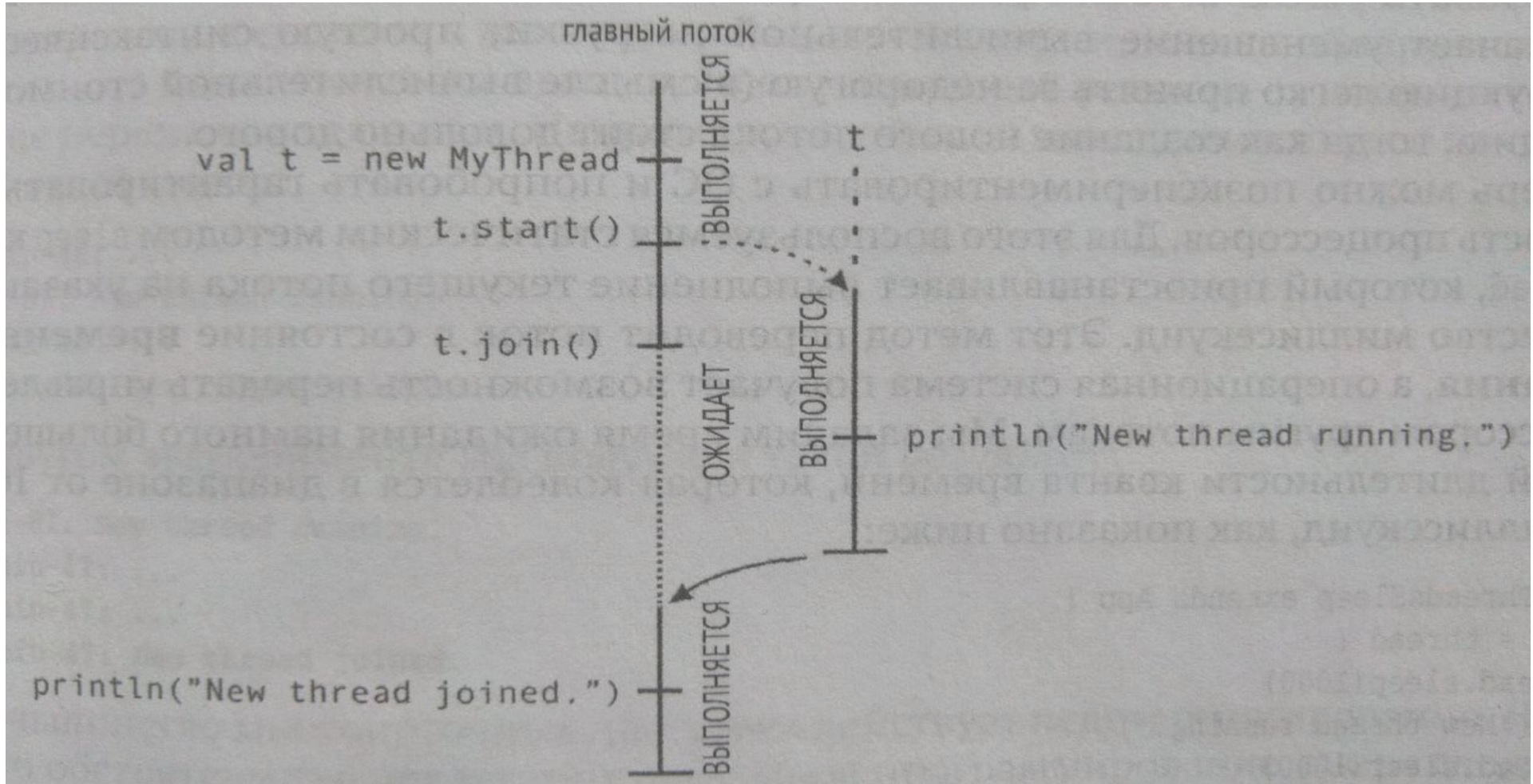


Рис. 2.2 ❖ Запуск и выполнение дочернего потока

Конструктор Thread

- Thread()
- Thread(String name)
- Thread(Runnable runnable)
- Thread(Runnable runnable, String name)

Свойства потока

- Основные свойства
  - id – идентификатор потока
  - name – имя потока
  - priority – приоритет
  - daemon – поток-демон
- Свойства потока не могут изменяться после запуска

Приоритеты потоков

- void **setPriority**(int priority) – устанавливает приоритет потока. Возможные значения priority — **MIN_PRIORITY**, **NORM_PRIORITY**(5) и **MAX_PRIORITY**(10).

- int **getPriority**() – получает приоритет потока.

Некоторые полезные методы класса Thread

- boolean **isAlive()** — возвращает true если myThready() выполняется и false если поток еще не был запущен или был завершен.
- **setName(String threadName)** – Задает имя потока.
- String **getName()** – Получает имя потока.
- static Thread **Thread.currentThread()** — статический метод, возвращающий объект потока, в котором он был вызван.
- long **getId()**– возвращает идентификатор потока. Идентификатор – уникальное число, присвоенное потоку.

Состояние потока

- Состояние потока возвращается методами
  - int **getState**()
  - boolean **isAlive**()

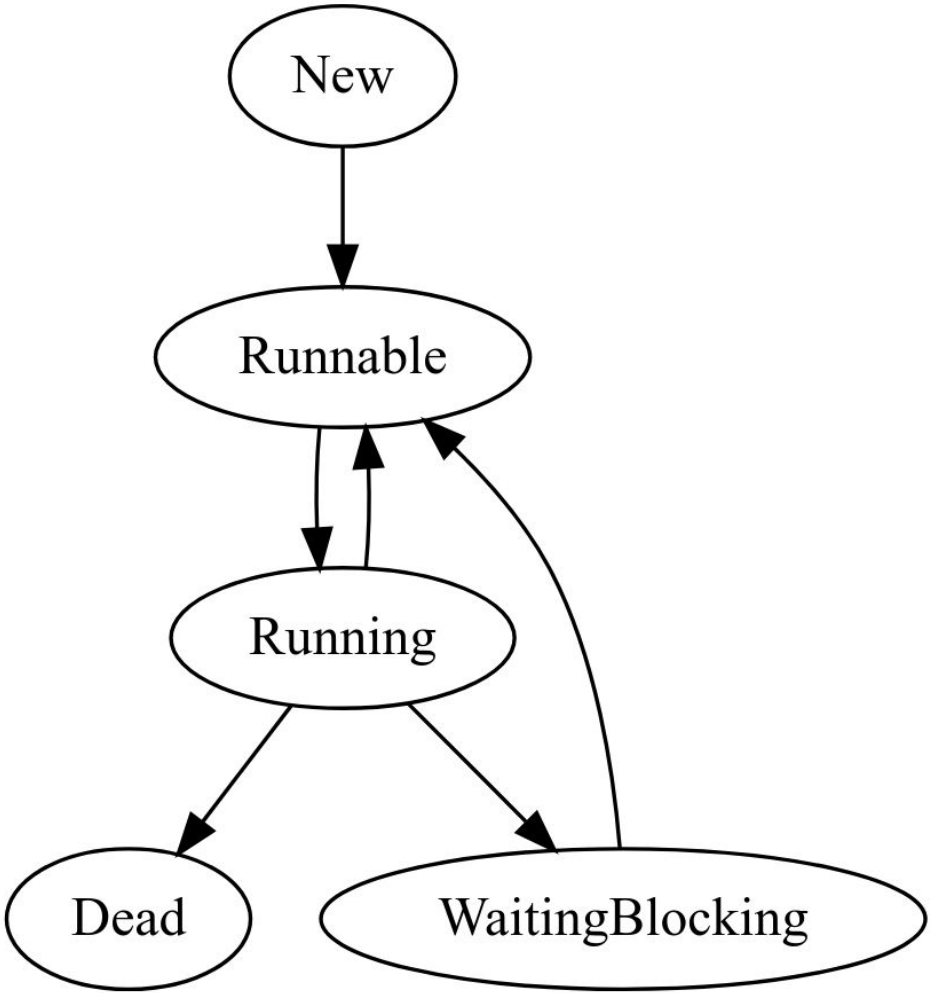
getState()	**isAlive()**
NEW	-
RUNNABLE	+
BLOCKED	+
WAITING	+
TIME_WAITING	+
TERMINATED	-

Потоки

- В процессе выполнения каждый поток проходит через последовательность **состояний потока**.
- При создании, поток получает состояние “**новый**”.
- После того как объект потока будет запущен на выполнение, он перейдет в состояние “**выполняющийся**”.
- Завершив работу - в состояние “**завершающийся**”. Его нельзя будет запустить повторно.

Поток может находиться в одном из пяти состояний:

- Новый (**new**). После создания экземпляра потока, он находится в состоянии **Новый** до тех пор, пока не вызван метод **start()**. В этом состоянии поток не считается живым.
- Работоспособный (**runnable**). Поток переходит в состояние Работоспособный, когда вызывается метод **start()**. Поток может перейти в это состояние также из состояния **Работающий** или из состояния **Блокирован**.
- Работающий (**running**). Поток переходит из состояния **Работоспособный** в состояние Работающий, когда Планировщик потоков выбирает его из runnable pool как работающий в данный момент.
- Ожидающий (**waiting**)/Заблокированный (**blocked**)/Спящий (**sleeping**). Эти состояния характеризуют поток как не готовый к работе.
- Мёртвый (**dead**).


New
Runnable
Running
Dead
WaitingBlocking

Взаимодействие потоков

- Создание потока
- Запуск потока (**start**)
- Ожидание окончания потока (**join**)
- Прерывание потока (**interrupt**)
- Засыпание потока (**sleep**)
- Переключение потоков (**yield**)

Блокировка другого потока

- Методы **sleep** и **yield** – статические! И действуют на текущий поток!
- Приостановить работу другого потока нельзя.

Удержание блокировки

При вызове **yield**, **sleep**, **join** поток удерживает все свои блокировки!

java.util.concurrent

в 2004 году в Java 5 добавили Concurrency API.

Он находится в пакете java.util.concurrent и содержит большое количество полезных классов и методов для многопоточного программирования.

ExecutorService

- Высокоуровневая замена работе с потоками напрямую.
- Исполнители выполняют задачи асинхронно и обычно используют пул потоков, так что нам не надо создавать их вручную.
- Все потоки из пула будут использованы повторно после выполнения задачи, а значит, мы можем создать в приложении столько задач, сколько хотим, используя один исполнитель.

ExecutorService

- Если надо создать пул с 2мя потоками, то делается это так:

```
ExecutorService service =
Executors.newFixedThreadPool(2);
```

- Если требуется использовать кэширующий пул потоков, который создает потоки по мере необходимости:

```
ExecutorService service =
Executors.newCachedThreadPool();
```

ExecutorService

Работу исполнителей надо завершать явно.
Для этого в интерфейсе **ExecutorService** есть два метода:

- **shutdown()**, который ждет завершения запущенных задач
- **shutdownNow()**, который останавливает исполнитель немедленно.

Пример остановки

- Executors1

Callable и Future

```
package java.lang;

@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

ScheduledExecutorService

- планировщик возвращает особый тип **Future** — **ScheduledFuture**, который предоставляет метод **getDelay()** для получения оставшегося до запуска времени

- у планировщика есть 2 метода:
  - **scheduleAtFixedRate()**
    - устанавливает задачи с определенным интервалом
    - принимает начальную задержку, которая определяет время до первого запуска.
  - **scheduleWithFixedDelay()**
    - работает так же, но указанный интервал будет отсчитываться от времени завершения предыдущей задачи

Попробуем запустить этот код 10000 раз в разных потоках

какой будет результат?

```
int count = 0;
void increment() {
    count = count + 1;
}
```

Результат

9965, 9964, 9980, ...

Почему это произошло?

Причина этого — использование изменяемой переменной несколькими потоками без синхронизации, что вызывает состояние гонки (race condition).

Что делать?

```
synchronized void incrementSync() {
    count = count + 1;
}
```

Synchronized

```
void incrementSync() {
    synchronized (this) {
        count = count + 1;
    }
}
```

Synchronized

- Под капотом Java использует так называемый монитор (**monitor lock**, **intrinsic lock**) для обеспечения синхронизации
- Этот монитор привязан к объекту, поэтому синхронизированные методы используют один и тот же монитор соответствующего объекта
- Все неявные мониторы устроены реентерабельно (**reentrant**), т.е. таким образом, что поток может без проблем вызывать блокировку одного и того же объекта, исключая взаимную блокировку (например, когда синхронизированный метод вызывает другой синхронизированный метод на том же объекте).

Блокировки

- Любой объект может служить блокировкой
- Снятие блокировки производится автоматически
- Синтаксис

```
synchronized (o) { // Получение блокировки
    ...
} // Снятие блокировки
```

Методы экземпляра

Метод экземпляра может быть объявлен синхронизированным:

```
public synchronized int getValue() { ... }
```

Эквивалентно (почти) :

```
public int getValue() {
    synchronized (this) {
        ...
    }
}
```

Методы экземпляра

Когда поток доходит до секции синхронизации,
он обращается к планировщику потоков и говорит, что хочет захватить объект.
Если этот объект уже захвачен, планировщик переводит наш поток в состояние Blocked. Как только поток, захвативший объект выходит из секции синхронизации, он сообщает планировщику,
что объект ему больше не нужен, и планировщик, когда в следующий раз дойдет до потока,
который ждет освободившийся объект, разблокирует его и дает захватить объект.

Блокировка потоков

1. Предположим, что есть захваченный объект и два потока, которые его ждут. Никто не может предсказать, какой из них проснется первым. То есть потоки просыпаются не в том порядке, в котором они просили блокировку, а в каком-то произвольном. Это называется нечестная блокировка.
2. **synchronized()** не бросает **InterruptedException**. Поэтому прервать поток, ожидающий блокировку, нельзя. Это увеличивает шанс получить дедлок.

Методы экземпляра

- http://www.javenue.info/post/87
- https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf
-

Методы экземпляра

- Поток может владеть блокировками сразу по нескольким объектам или даже по одному объекту несколько раз.
- Это полезно, когда, например, есть код, который синхронизируется по объекту и внутри себя вызывает метод, который тоже синхронизируется по этому объекту.
- Дедлока при этом не произойдет. Блокировка освободится, когда поток отпустит объект столько же раз, сколько захватил.

Статические synchronized-методы синхронизируются по своему классу

```
class Example {
    public static synchronized void foo() { ... }
 }

// эквивалентно
class Example {
    public static void foo() {
        synchronized (Example.class) { ... }
    }
}
```

- При вызове yield(), sleep(), join() поток удерживает все свои блокировки. То есть когда вы отправляетесь спать, вы отправляетесь спать со всем багажом блокировок, которые вы захватили.

Рассмотрим пример

- У нас есть очередь, которая может хранить не больше одного элемента.
- И есть два потока – производитель и потребитель. Производитель складывает объекты в очередь, потребитель достает их.
- Производитель не может ничего положить в очередь, если там занято, а потребитель не может ничего достать, если там ничего нет.
- У очереди есть объект `private Object data` и методы:

`public void set(Object data){...}` и
`public Object get(){...}`

```
public void set(Object data) {
  while (true) {
    synchronized (this) {
      if (this.data == null) {
        this.data = data;
        break;
      }
    }
  }
}
```

```
public Object get() {
  while (true) {
    synchronized (this) {
      if (data != null) {
        Object d = data;
        data = null;
        return d;
      }
    }
  }
}
```

- Вопрос первый. Зачем здесь брать блокировку?
  - Потому что могло так случиться, что два производителя одновременно посмотрели, что место в очереди есть, потом каждый стал складывать туда свой объект, и тот объект, который записался чуть позже, победил. А другой потерялся.
- Вопрос второй. Почему нельзя сделать synchronized-методы?
  - Потому что как только мы в него войдем, мы заблокируем всем остальным потокам доступ к очереди, и место никогда не освободится (или объект никогда не добавится).
- Вопрос третий. Чем плох этот код?
  - Тем, что мы внутри цикла постоянно берем блокировку. Эта операция довольно дорогая, поэтому производительность от этого пострадает.

Мониторы и условия

- В классе **Object** есть три метода: **wait()**, **notify()**, **notifyAll()**.
- Когда текущий поток вызывает у объекта метод **obj.wait();** , он переходит в состояние Waiting и ждет, что кто-нибудь вызовет у этого же объекта метод **obj.notify();**
- Метод **notifyAll()** будит все потоки, которые ждут данный объект, метод **notify()** будит только один поток. Такая блокировка, опять же, нечестная, то есть мы не знаем, какой поток проснется первым.

Мониторы и условия

Важно, что все эти методы можно вызывать от объекта, только если текущий поток владеет блокировкой на этот объект. Если это не так, бросится **IllegalMonitorStateException**.

Что произойдет?

```
Thread t1 = new Thread(() -> {
    . . .
    synchronized (x) {
        try {
            x.wait();
        } catch (InterruptedException e) {}// wait() кидает
InterruptedException!
    }
});
Thread t2 = new Thread(() -> {
    . . .
    synchronized (x) {
        x.notify();
    }
});
```

Ответ

Метод **wait()** освобождает блокировку в начале ожидания и снова захватывает в конце. Так что у других потоков будет возможность захватить блокировку, пока все ожидающие спят.

Зачем тогда методу wait() нужна блокировка?

- У каждого потока есть некое множество объектов, которые он заблокировал. И у каждого объекта такого объекта есть флажок, ждем ли мы его. Эта информация нужна планировщику потоков, чтобы он мог нас потом разбудить.
- В примере с очередью мы делали какие-то синхронизированные действия с очередью, затем ждали, и продолжали что-то делать с очередью. В таком случае wait() автоматически окажется внутри синхронизированного блока, и ему не придется снова ждать блокировку.

Вернемся к примеру с очередью

```
public synchronized void set(Object data) throws InterruptedException {
    if (this.data != null) {
        wait(); // Пассивное ожидание, wait() вызван от this
    }
    this.data = data;
    notify(); // notify() вызван от this
}
public synchronized Object get() throws InterruptedException {
    if (data == null) {
        wait(); // Пассивное ожидание, wait() вызван от this
    }
    Object d = data;
    data = null;
    notify(); // notify() вызван от this
    return d;
}
```

Очередная проблема

- Эта проблема называется "внезапное/спонтанное пробуждение". Дело в том, что поток при ожидании wait() может проснуться до того, как его уведомили. Это сделали для защиты от дедлоков.

Решение

- При sleep() и join() спонтанное пробуждение случиться не может

Блокировки

```
public interface Lock {
   void lock();

   void lockInterruptibly() throws InterruptedException;

   boolean tryLock();

   boolean tryLock(long var1, TimeUnit var3) throws InterruptedException;

   void unlock();

   Condition newCondition();
}
```

Стандартные реализации Lock из JDK

- ReentrantLock
- ReadWriteLock
- StampedLock

ReentrantLock

- **lock**() - блокировка
- **unlock**() - освобождение ресурса
- **isLocked**() - залочен или нет
- **isHeldByCurrentThread**() - имеет ли текущий поток блокировку
- **tryLock**() - в отличие от обычного **lock**() не останавливает текущий поток в случае, если ресурс уже занят. Он возвращает булевый результат, который стоит проверить перед тем, как пытаться производить какие-то действия с общими объектами (истина означает, что контроль над ресурсами захватить удалось).

Очень важно оборачивать код в try{}finally{}

ReadWriteLock

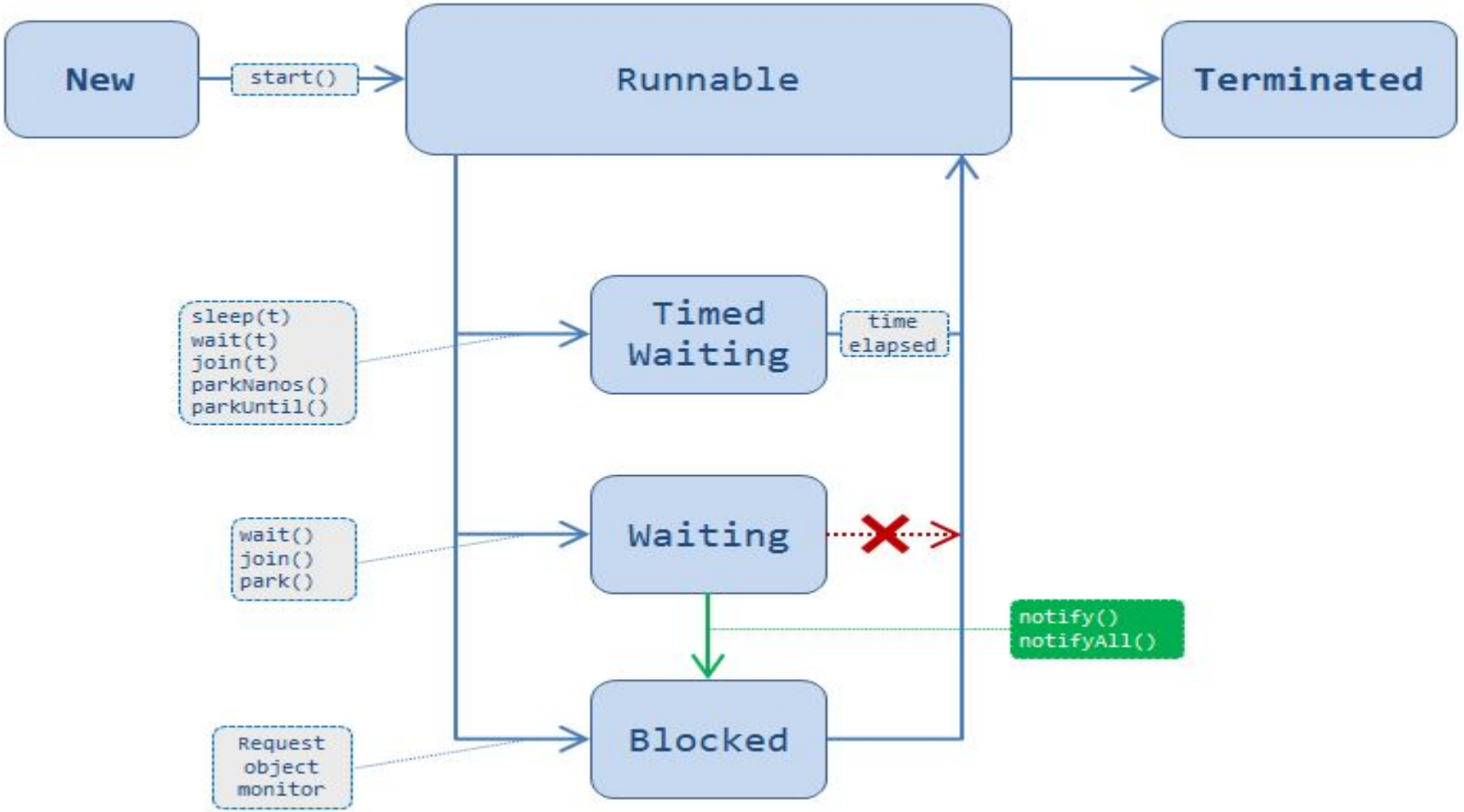
```
public interface ReadWriteLock {
    Lock readLock();

    Lock writeLock();
}
```

ReadWriteLock

- считывать данные (любому количеству потоков) безопасно до тех пор, пока ни один из них не изменяет переменную.
- блокировку для чтения (**read-lock**) может удерживать любое количество потоков до тех пор, пока не удерживает блокировка для записи (**write-lock**).

StampedLock

- поддерживает разделение на **readLock()** и **writeLock()**.
- в отличие от **ReadWriteLock**, метод блокировки **StampedLock** возвращает «штамп» — значение типа long. Этот штамп может использоваться в дальнейшем как для высвобождения ресурсов, так и для проверки состояния блокировки. Вдобавок, у этого класса есть методы, для реализации «оптимистичной»


New
start()
Runnable
Terminated
sleep(t)
wait(t)
join(t)
parkNanos()
parkUntil()
Timed Waiting
time elapsed
wait()
join()
park()
Waiting
notify()
notifyAll()
Request object monitor
Blocked

ThreadPoolExecutor

- corePoolSize - это количество основных потоков, которые будут созданы и сохранены в пуле. Если все основные потоки заняты и отправлено больше задач, то пулу разрешается увеличиваться до максимального размера пула (maximumPoolSize)
- maximumPoolSize - максимальное количество потоков
- keepAliveTime - это интервал времени, в течение которого избыточным потокам (потокам, экземпляры которых превышают corePoolSize) разрешено существовать в состоянии ожидания

Что под капотом?

- Если запущено больше потоков чем размер начального пула:

  `private final BlockingQueue<Runnable> workQueue;`

- Если запущено меньше потоков начального размера пула, пул попробует стартовать новый поток:

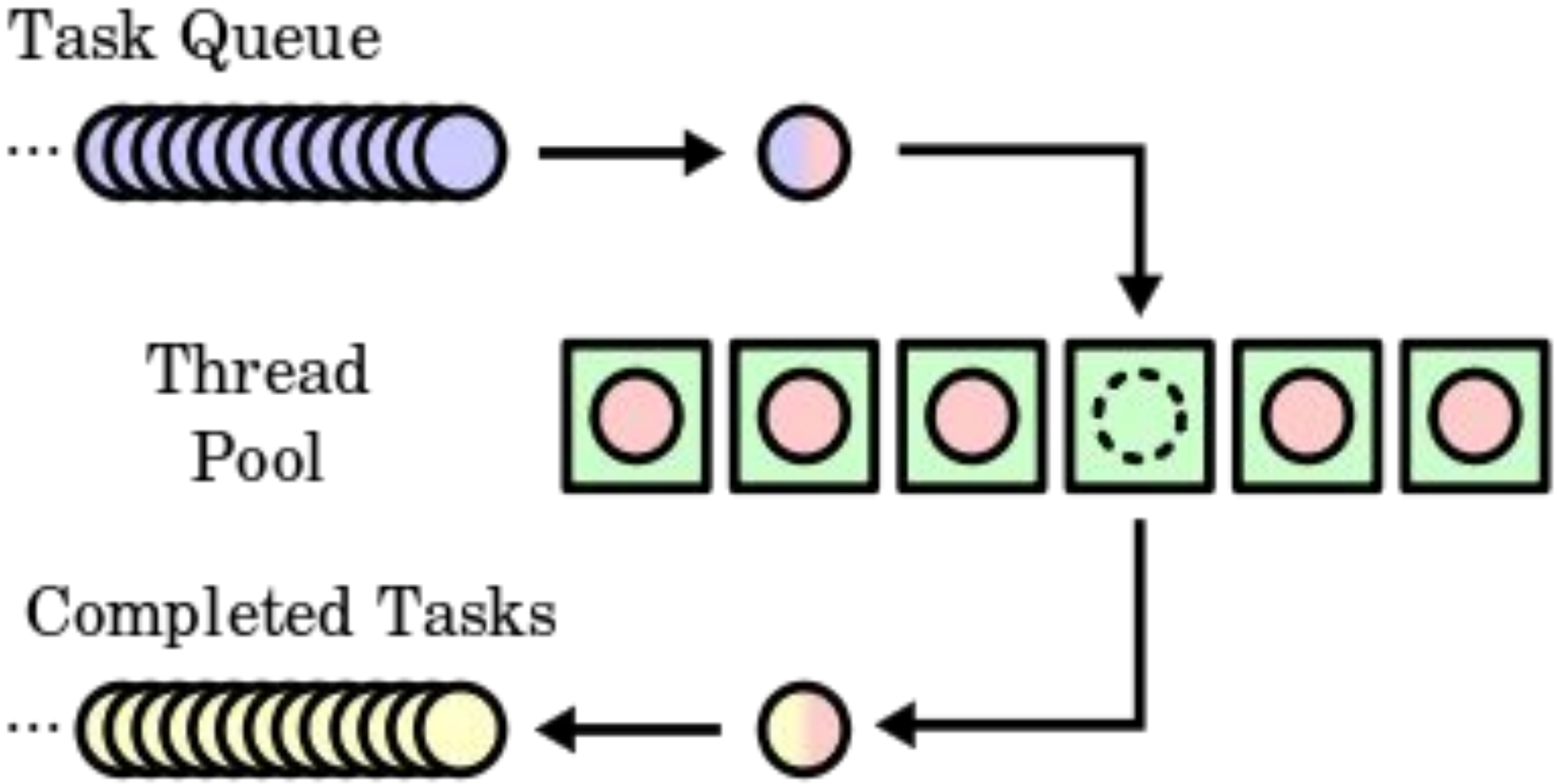
```
public void execute(Runnable command) {
    ...
     if (workerCountOf(c) < corePoolSize) {
         if (addWorker(command, true))
             return;
    ...
    if (isRunning(c) && workQueue.offer(command)) {
    ...
         addWorker(null, false);
    ...
    }
}
```

- Каждый addWorker запускает новый поток с задачей Runnable, которая опрашивает workQueue на наличие новых задач и выполняет их:

```
final void runWorker(Worker w) {
    ...
     try {
         while (task != null || (task = workQueue.poll(keepAliveTime,
TimeUnit.NANOSECONDS)) != null) {
             ...
                  task.run();
    ...
}
```

CachedThreadPool

- Эти значения параметров означают, что пул кэшированных потоков может расти без границ для размещения любого количества отправленных задач . Но когда потоки больше не нужны, они удаляются через 60 секунд бездействия. Типичный вариант использования - это когда в вашем приложении много недолговечных задач.
- CorePoolSize фактически установлен в 0, а maximumPoolSize устанавливается в Integer.MAX_VALUE для этого экземпляра. KeepAliveTime составляет 60 секунд для этого.


Task Queue
Thread Pool
Completed Tasks

Семафоры

- Семафоры — отличный способ ограничить количество потоков, которые одновременно работают над одним и тем же ресурсом.

- https://habr.com/ru/post/277669/

Atomic

Пакет **java.concurrent.atomic** содержит много полезных классов для выполнения атомарных операций.

Операция называется атомарной тогда, когда ее можно безопасно выполнять при параллельных вычислениях в нескольких потоках, не используя при этом ни блокировок, ни **synchronized**.

Atomic

- Внутри атомарные классы очень активно используют сравнение с обменом (compare-and-swap, CAS), атомарную инструкцию, которую поддерживает большинство современных процессоров.

- Эти инструкции работают гораздо быстрее, чем синхронизация с помощью блокировок. Поэтому, если вам просто нужно изменять одну переменную с помощью нескольких потоков, лучше выбирать атомарные классы.

Atomic

- AtomicBoolean
- AtomicInteger
- AtomicIntegerArray
- AtomicLong
- AtomicLongArray
- DoubleAccumulator
- DoubleAdder
- LongAccumulator
- AtomicReference
- ...

LongAdder

increment() и **add()**:

- Но вместо того, чтобы складывать числа сразу, он просто хранит у себя набор слагаемых. Узнать результат можно с помощью вызова **sum()** или **sumThenReset()**
- Этот класс используется в ситуациях, когда добавлять числа приходится гораздо чаще, чем запрашивать результат (часто это какие-то статистические исследование, например подсчёт количества запросов).
- **LongAdder** требует гораздо большего количества памяти из-за того, что он хранит все слагаемые.

LongAccumulator

- Класс **LongAccumulator** несколько расширяет возможности **LongAdder**
- Вместо простого сложения он обрабатывает входящие значения с помощью лямбды типа **LongBinaryOperator**, которая передается при инициализации.

ConcurrentMap

```
map.forEach((key, value)
    -> System.out.printf("%s = %s\n", key, value));

String value = map.putIfAbsent("c3", "p1");
System.out.println(value); // p0

String value = map.getOrDefault("hi", "there");
System.out.println(value); // there
```

Deadlock

Взаимная блокировка – это ситуация в которой, два или более процесса занимая некоторые ресурсы, пытаются заполучить некоторые другие ресурсы, занятые другими процессами и ни один из процессов не может занять необходимый им ресурс, и соответственно освободить занимаемый.

Deadlock

```
Thread t1 = new Thread(() -> {
    synchronized (a) { synchronized (b) {...}}
});

Thread t2 = new Thread(() -> {
    synchronized (b) { synchronized (a) {...}}
});

t1.start();
t2.start();
```

Compare-And-Swap, CAS

Операция CAS концептуально эквивалентна:

```
public Boolean compareAndSet(Long ov, Long nv) {
    synchronized (this) {
        if (this.get == ov) return false else {
            this.set(nv);
            return true;
        }
    }
}
```

Потокозащищенный счетчик

```
public final class Counter {
    private long value = 0;

    public synchronized long getValue() {
        return value;
    }

    public synchronized long increment() {
        return ++value;
    }
}
```

Неблокирующий счетчик, использующий CAS

```
public class NonblockingCounter {
    private AtomicInteger value;

    public int getValue() {
        return value.get();
    }

    public int increment() {
        int v;
        do {
            v = value.get();
        while (!value.compareAndSet(v, v + 1));
        return v + 1;
    }
}
```

Есть много потоков, и каждый из них 1000000 раз делает x++

Сравним скорость работы такой программы при разных подходах к синхронизации:

- Самым медленным будет подход с честной блокировкой. Пусть один поток захватил блокировку, другой стал ждать. Первый поток выполнил работу, и у него есть время сделать это еще раз. Он снова просит блокировку, но не получает её, так как другой поток запросил её раньше. Получается, что после каждого инкремента происходит переключение контекста, что очень сильно замедляет работу программы.
- На втором месте идет synchronized. Он нечестный, поэтому более быстрый.
- Lock работает еще немного лучше
- Самым быстрым будет использование AtomicInt без всяких блокировок, потому что он реализован нативно и супер оптимизирован.

Почитать отдельно

- https://www.nurkiewicz.com/2014/11/executorservice-10-tips-and-tricks.html
- https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf
- https://au.cdkrot.me/get/spbau-2020/term4-java
- https://www.ibm.com/developerworks/library/j-jtp0730/index.html
- https://habr.com/ru/post/277669/
-

Литература

- Конкурентное программирование на Scala / Александр Прокопец, 2018
- https://tproger.ru/translations/java8-concurrency-tutorial-1
- https://tproger.ru/translations/java8-concurrency-tutorial-2
- https://tproger.ru/translations/java8-concurrency-tutorial-3
- https://habr.com/ru/post/164487/
- https://www.ibm.com/developerworks/ru/library/j-5things15/index.html
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html