

КРОК

Java: Многопоточное программирование

Александр Головин

Ведущий инженер-разработчик



Многозадачность

Многозадачность - свойство операционной системы или среды выполнения обеспечивать возможность параллельной обработки нескольких задач.



Процессы(Processes)

Процесс - изолированная, независимо выполняемая программа, которая пользуется ресурсами операционной системы, такими как:

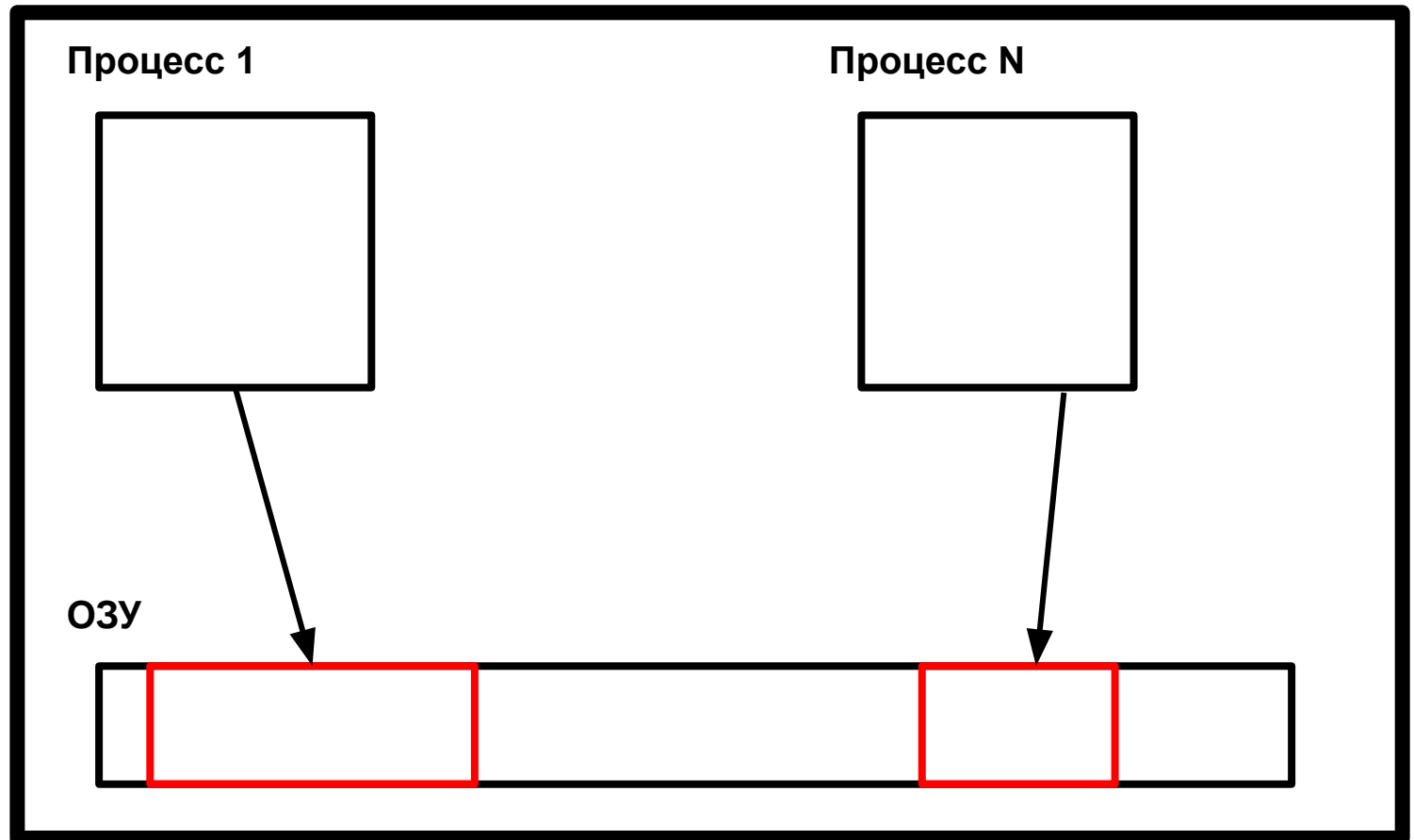
- память
- дескрипторы файлов
- учетные данные системы безопасности

Процессы могут взаимодействовать друг с другом с помощью коммуникационных механизмов:

- сокеты
- обработчики сигналов
- совместная память
- семафоры
- файлы

Процессы(Processes)

Операционная система




Потоки(Threads)

Поток - “облегченный процесс”, который существует в рамках процесса.

Потоки одного процесса имеют доступ к общей памяти(heap) этого процесса.

Каждый поток имеет свой собственный stack.

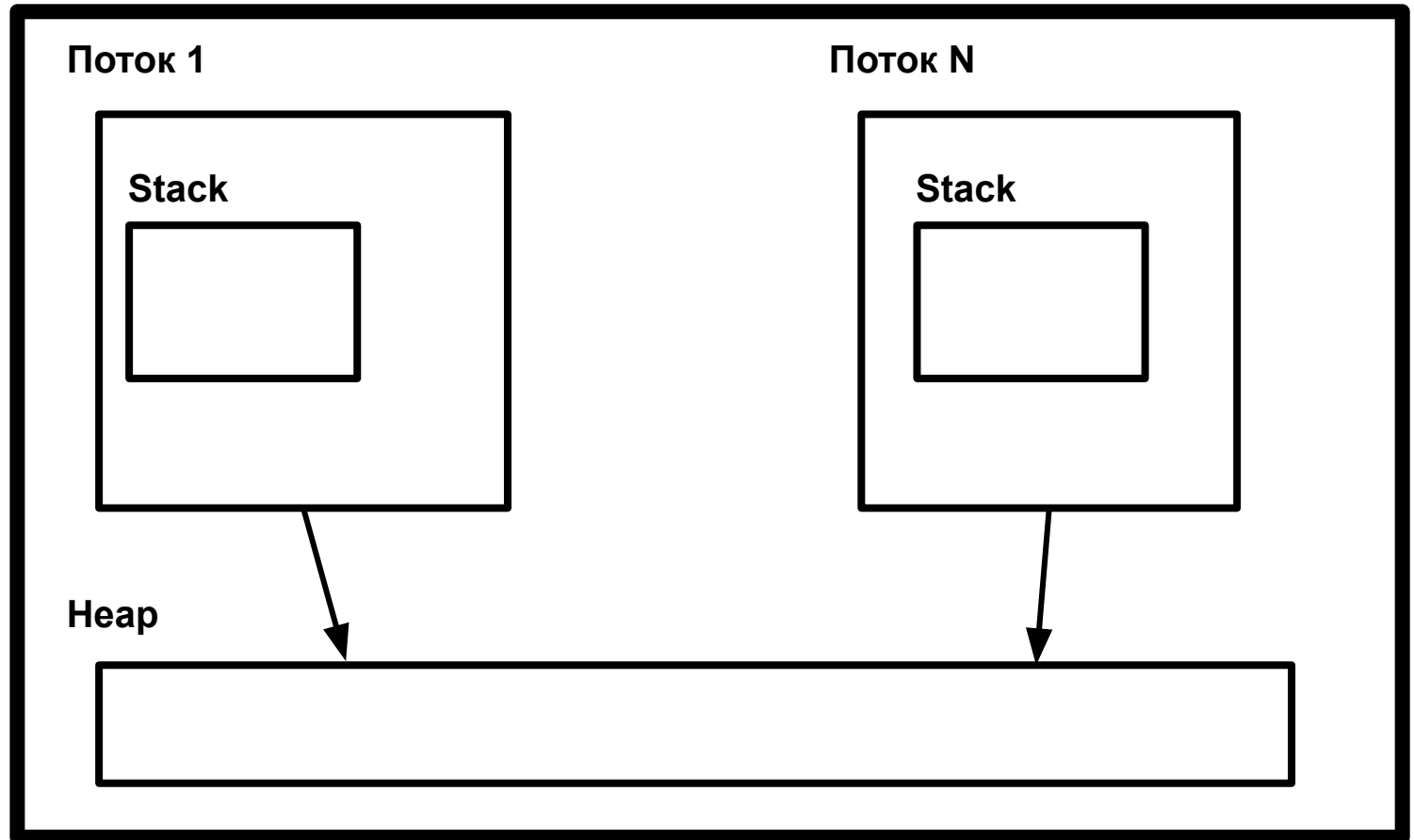


```
elif operation == "MIRROR_X":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
elif operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True

#selection at the end -add ba
mirror_ob.select= 1
modifier_ob.select=1
bpy.context.scene.objects.active
print("Selected" + str(modifier_o
    #mirror_ob.select = 0
    time = bpy.context.selected
    time_data = bpy.context.selected
```

Потоки(Threads)

Процесс



Закон Мура

В 1965 году один из основателей Intel Гордон Мур в процессе подготовки выступления нашел закономерность: появление новых моделей микросхем наблюдалось спустя примерно год после предшественников, при этом количество транзисторов в них возрастало каждый раз приблизительно вдвое. Он предсказал, что к 1975 году количество элементов в чипе вырастет до 216 (65536) с 26 (64) в 1965 году.

Мур пришел к выводу, что при сохранении этой тенденции мощность вычислительных устройств за относительно короткий промежуток времени может вырасти экспоненциально. Это наблюдение получило название — закон Мура.

Закон Мура

В 2003 году Мур опубликовал работу «No Exponential is Forever: But „Forever“ Can Be Delayed!», в которой признал, что экспоненциальный рост физических величин в течение длительного времени невозможен, и постоянно достигаются те или иные пределы. Лишь эволюция транзисторов и технологий их изготовления позволяла продлить действие закона ещё на несколько поколений.

Масштабирование

Вертикальное масштабирование — увеличение производительности каждого компонента системы с целью повышения общей производительности. Возможность заменять в существующей вычислительной системе компоненты более мощными и быстрыми по мере роста требований и развития технологий.

Горизонтальное масштабирование — разбиение системы на более мелкие структурные компоненты и разнесение их по отдельным физическим машинам. Возможность добавлять к системе новые узлы, серверы для увеличения общей производительности.

Потоки в Java. Класс Thread.

Thread - представляет поток исполнения в Java.

- start() - запускает новый поток
- run() - (не вызываем!) тут должен быть код, который запустится в рамках нового потока
- interrupt() - подаёт сигнал прерывания потока исполнения
- isInterrupted() - проверяет был ли поток прерван
- join() - блокирует текущий поток до завершения потока на экземпляре, которого был вызван метод
- isDaemon() - проверяет является ли поток демоном
- isAlive() - проверяет запущен ли поток
- dumpStack() - выводит стектрейс потока
- getName() - возвращает название потока
- getPriority() - возвращает приоритет потока

Runnable - представляет, “исполняемый метод”.

- run() - метод, который может быть запущен в потоке исполнения

Создание и запуск нового потока. Способ 1

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
        . . .
    }
}
```

The following code would then create a thread and start it running:

```
PrimeThread p = new PrimeThread(143);
p.start();
```


Создание и запуск нового потока. Способ 2

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
        . . .
    }
}
```

The following code would then create a thread and start it running:

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

Потокобезопасность

Написание потокобезопасного кода - это управление доступом к состоянию, в частности к разделяемому изменяемому состоянию (shared mutable state).

К разделяемой памяти могут обратиться несколько потоков.

Изменяемая состояние - может менять свое значение.

На самом деле мы пытаемся защитить от неконтролируемого конкурентного доступа не код, а данные.

Состояние гонки(Race condition)

```
boolean flag;
```

```
void thread1() {  
    while (!flag) {  
        System.out.println("Ждём");  
    }  
}
```

```
void thread2() {  
    flag = true;  
}
```


Атомарность

Например, операция `count++` состоит из других операций:

1. прочитать
2. изменить
3. записать

В 32 разрядных процессорах операции чтения и записи 64 битных данных были не атомарны.

Синхронизация

Java предоставляет встроенный замковый механизм для усиления атомарности - **синхронизированный блок**, состоящий из ссылки на **объект-замок(lock)**, и блока кода, который будет им защищен.

synchronized (lock) {

// обращение к разделяемому ресурсу, этот блок кода

// выполняется одновременно только в одном потоке

}

Замком одновременно может владеть только один поток. Поток дошедший до замка, будет блокирован до момента освобождения замка, потоком захватившим его ранее. Поток захвативший поток может захватить его повторно.

т.к. в этой области кода может находиться только один поток, то операции выполняются атомарно.

Взаимная блокировка(Deadlock)

```
void thread1() {
    synchronized (lock1) {
        while (!flag) {
            System.out.println("Ждем сообщения");
        }
        synchronized (lock2) {
            System.out.println(expectedValue);
        }
    }
}

void thread2() {
    synchronized (lock2) {
        expectedValue = "Сообщение для потока 1";
        synchronized (lock1) {
            flag = true; // сообщение записано
        }
    }
}
```


Синхронизация

Все обращения к изменяемой разделяемой переменной должны выполняться с удержанием одного и того же замка. Только тогда переменная защищена этим замком.

Каждая разделяемая переменная должна быть защищена только одним замком. Все должны четко понимать какой это замок.

Вопрос: Почему нельзя весь код программы сделать синхронизированным?

Закон Амдала

«В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента».

$$S_p = \frac{1}{\alpha + \frac{1 - \alpha}{p}}$$

α - доля последовательного кода

p - количество потоков

S_p - во сколько раз программа выполнится быстрее

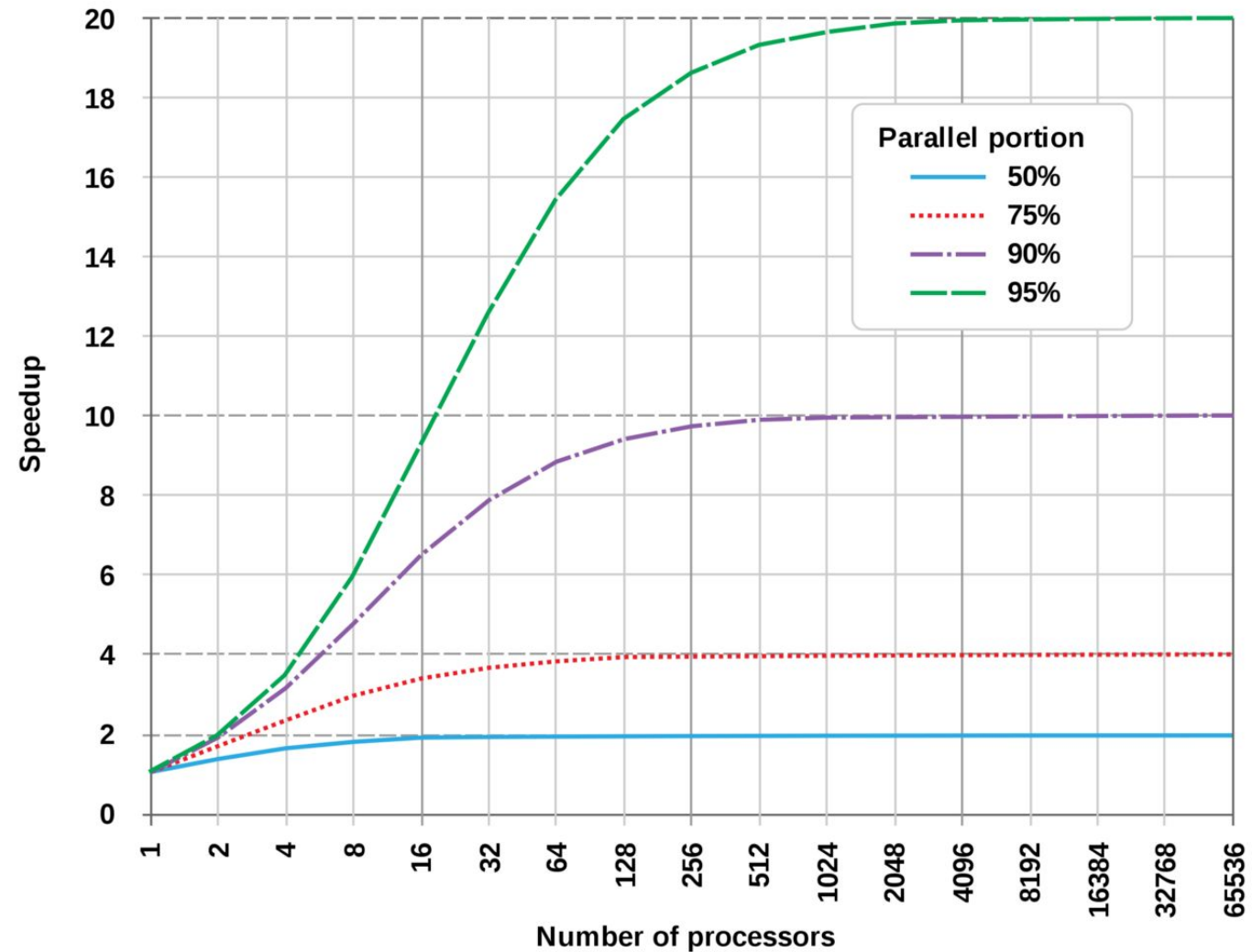
Закон Амдала

```
elif_operation == "MIRROR_X":  
    mirror_mod.use_x = False  
    mirror_mod.use_y = True  
    mirror_mod.use_z = False  
elif_operation == "MIRROR_Z":  
    mirror_mod.use_x = False  
    mirror_mod.use_y = False  
    mirror_mod.use_z = True
```

```
#selection at the end -add ba  
mirror_ob.select= 1  
modifier_ob.select=1  
bpy.context.scene.objects.active  
print("Selected" + str(modifier_o
```

```
    #mirror_ob.select = 0  
time = bpy.context.scene.frame
```

Amdahl's Law



Видимость

Синхронизация имеет еще один важный аспект: видимость памяти(memory visibility), она обеспечивает потокам возможность видеть изменения внесенные другими потоками.

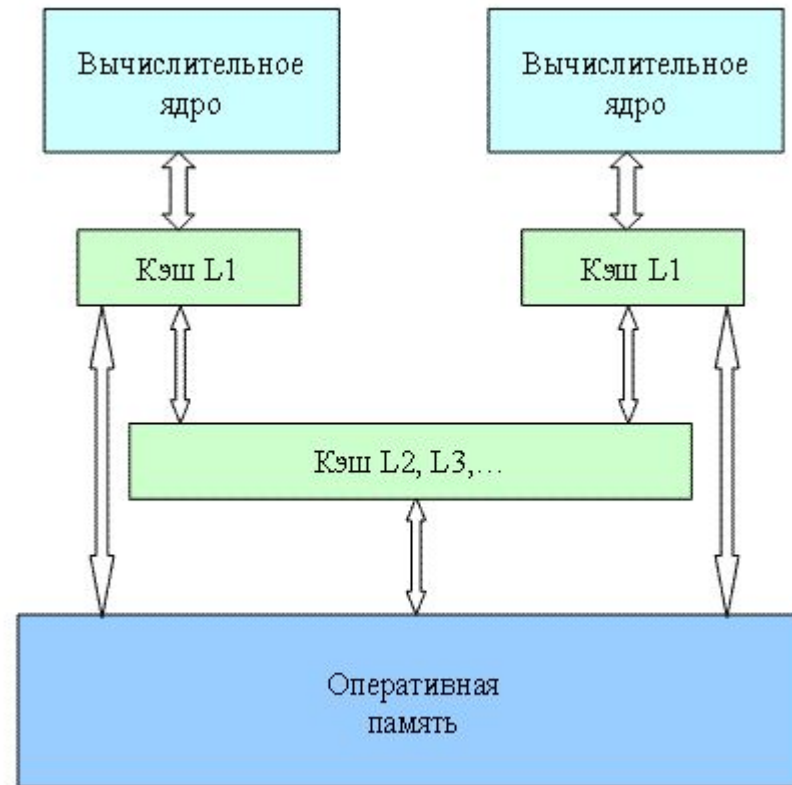
Когда чтение и запись переменной происходит из разных потоков, нет гарантии(без синхронизации), что читающий поток своевременно увидит значение, записанное другим потоком.

Без синхронизации компилятор, процессор и рабочая среда могут запутать порядок выполнения операций. Не стоит ожидать естественного порядка действий памяти в недостаточно синхронизированных многопоточных программах.

Нужно всегда применять правильную синхронизацию, когда данные используются потоками совместно!

Многоядерный процессор. Кэширование.

Многоядерный процессор — центральный процессор, содержащий два и более вычислительных ядра на одном процессорном кристалле или в одном корпусе.



Устаревшие данные

Устаревшие значения могут вызывать сбои в безопасности или жизнеспособности:

- неожиданные исключения
- поврежденные структуры данных
- неточные вычисления
- бесконечные циклы

Чтобы обеспечить видимость актуальных значений разделяемых переменных, синхронизируйте читающие и пишущие потоки на общем замке.

volatile

Переменные с модификатором volatile:

- операции над ними не будут переупорядочены
- они не кэшируются в регистрах или кэшах, где данные скрыты от других потоков
- их чтение всегда возвращает актуальный результат

Блокировка может гарантировать как видимость, так и атомарность, а volatile переменные только видимость.

Использование:

- запись в переменную не зависит от ее текущего значения
- при обращении к переменной заранее не требуется блокировка

volatile не может обеспечить атомарность операции i++.

Потокобезопасность

Класс является потокобезопасным, если он ведет себя правильно во время доступа из многочисленных потоков, без дополнительных синхронизаций или другой координации со стороны вызывающего кода.

Потокобезопасные классы инкапсулируют любую необходимую синхронизацию сами и не нуждаются в помощи со стороны.

Immutable object

Всегда потокобезопасны.
Просты.

Объект является immutable, если:

- его состояние невозможно изменить после создания
- все поля final
- он надлежаще создан т.е. ссылка this не ускользнула из объекта во вне

Java гарантирует безопасность инициализации final полей.

Immutable объекты можно использовать без синхронизации.

Безопасное совместное использование объектов

- Объект, ограниченный одним потоком, принадлежит эксклюзивно владеющему потоку, который может его изменять
- Потоки могут обращаться к объекту, предназначенному только для чтения, конкурентно, без синхронизации, но объект невозможно изменять
- Потокобезопасный объект выполняет синхронизацию внутри себя, поэтому потоки могут свободно обращаться к нему без синхронизации
- С удержанием конкретного замка можно обращаться к объекту

Atomics

- **Atomics** — классы с поддержкой атомарных операций над примитивами и ссылками
- Пакет `java.util.concurrent.atomic`
- **AtomicBoolean**
- **AtomicInteger**
- **AtomicLong**
- **AtomicIntegerArray**
- **AtomicLongArray**
- **AtomicReference**

Compare and swap (CAS)

- Параллельные алгоритмы на основе CAS называют безблокировочными, потому что потокам не приходится ожидать блокировки.
- Операция CAS проходит успешно или неуспешно, но независимо от исхода она завершается в предсказуемые сроки.
- Если CAS не реализуется, вызывающая программа может повторить попытку операции CAS или предпринять иные шаги, которые она считает нужными.

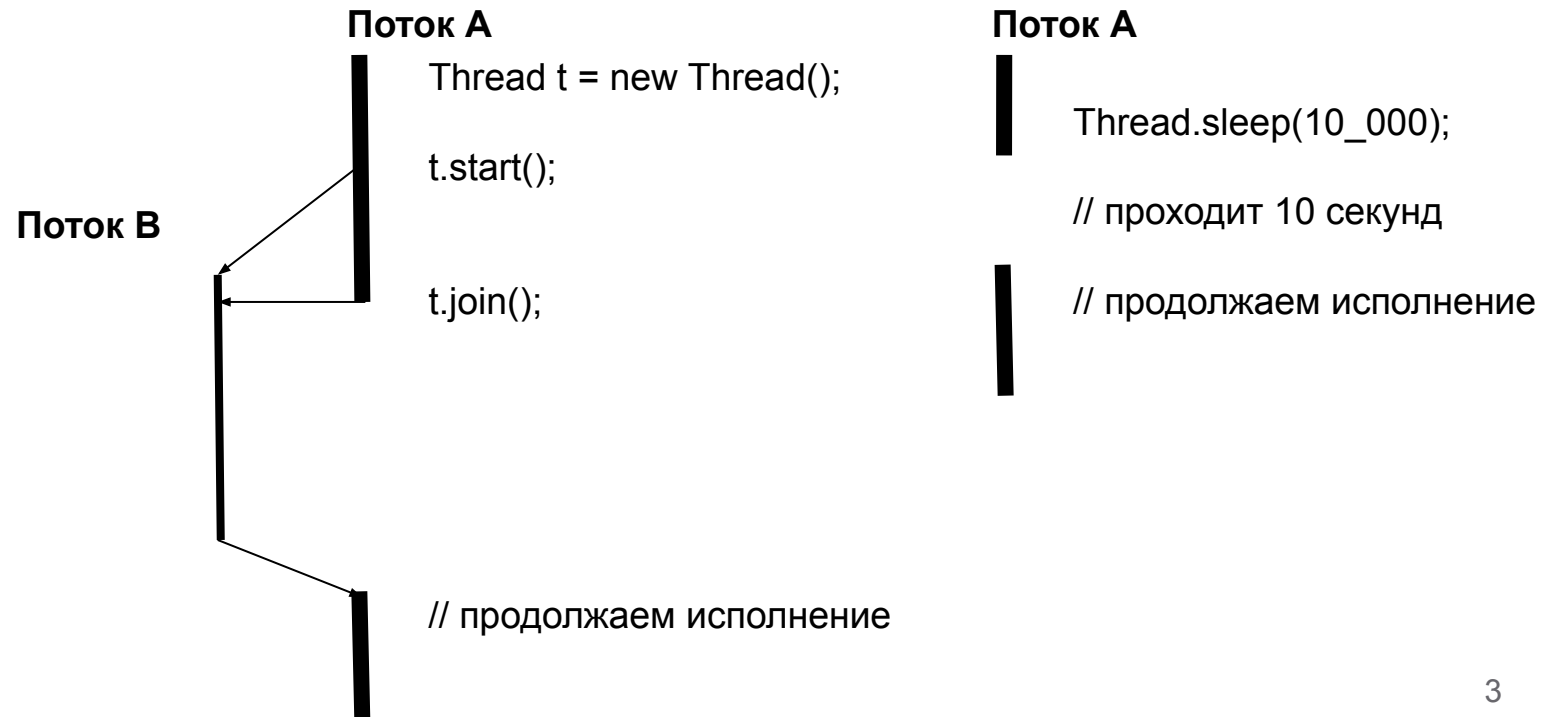
Sleep. Join.

`Thread.sleep(long millis)` - усыпляет текущий поток на заданное время

`void join()` - блокирует текущий поток, до завершения потока метода которого вызван

```
elif_operation == "MIRROR_X":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
elif_operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True

#selection at the end -add ba
mirror_ob.select= 1
modifier_ob.select=1
bpy.context.scene.objects.active
print("Selected" + str(modifier_o
    #mirror_ob.select = 0
    time = bpy.context.select
    bpy.data.objects[mod_ob.name].time
```



Прерывание потока

- Каждый поток имеет связанное с ним булево свойство **isInterrupted**, которое отображает его статус прерывания.
- **Статус** прерывания изначально имеет значение **false**.
- когда поток прерывается каким-либо другим потоком путем вызова **Thread.interrupt()**, то происходит одно из двух.
- Если другой поток выполняет прерываемый метод блокирования низкого уровня, такой как **Thread.sleep()**, **Thread.join()** или **Object.wait()**, то он разблокируется и выдает **InterruptedException**.
- Иначе, **interrupt()** просто устанавливает статус прерывания потока.

Прерывание потока

- Код, действующий в **прерванном потоке**, может позднее **обратиться** к статусу прерывания, чтобы **посмотреть**, был ли запрос на прекращение выполняемого действия.
- Статус прерывания может быть прочитан с помощью **Thread.isInterrupted()**
- И может быть прочитан и сброшен за одну операцию при помощи неудачно названного **Thread.interrupted()**.
- Если реализуется завершаемый поток, то периодически проверяем статус и реагируем. Например, можем выйти из метода потока, тем самым завершив работу потока.

```
public void run() {  
    while (!Thread.currentThread().isInterrupted()) {  
        // выполняем пока поток не прерван  
    }  
}
```


Wait. Notify. NotifyAll.

- **wait()** - освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()
- **notify()** - продолжает работу потока, у которого ранее был вызван метод wait()
- **notifyAll()** - возобновляет работу всех потоков, у которых ранее был вызван метод wait()

Lock

- **Lock** - более гибкий подход по ограничению доступа к ресурсам/блокам нежели при использовании `synchronized`
- **ReentrantLock** - замок на входение, только один поток может войти в защищенный блок.
- **ReadWriteLock** - дополнительный интерфейс для создания read/write замков, такие замки необычайно полезны, когда в системе много операций чтения и мало операций записи.

Lock

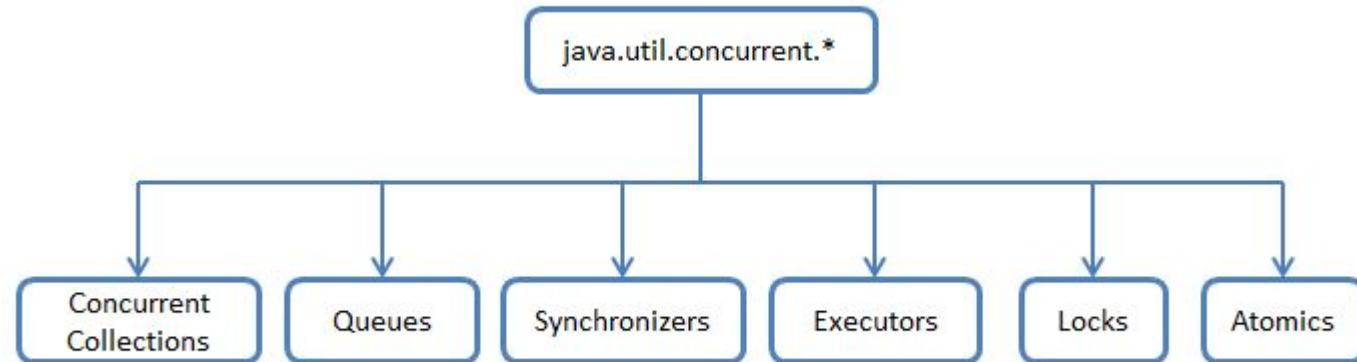
- **void lock()** - ожидает, пока не будет получена блокировка
- **void lockInterruptibly()** - ожидает, пока не будет получена блокировка, если поток не прерван
- **boolean tryLock()** - пытается получить блокировку, если блокировка получена, то возвращает true. Если блокировка не получена, то возвращает false. В отличие от метода lock() не ожидает получения блокировки, если она недоступна
- **void unlock()** - снимает блокировку

Lock. Пример.

```
Lock lock = new ReentrantLock();  
boolean value = false;
```

```
void update() {  
    lock.lock();  
    try {  
        value = true;  
    } finally {  
        lock.unlock();  
    }  
}
```

java.util.concurrent.*



Concurrent Collections

Concurrent Collections — набор коллекций, более эффективно работающие в многопоточной среде нежели стандартные универсальные коллекции из `java.util` пакета.

Вместо базового wrappers `Collections.synchronizedList` с блокированием доступа ко всей коллекции используются блокировки по сегментам данных или же оптимизируется работа для параллельного чтения данных по wait-free алгоритмам.

Concurrent Collections

CopyOnWriteArrayList<E> - потокобезопасный аналог ArrayList, реализованный с CopyOnWrite алгоритмом.

ConcurrentHashMap<K, V> - в отличие от Hashtable и блоков synchronized на HashMap, данные представлены в виде сегментов, разбитых по hash'ам ключей. В результате, для доступ к данным лочится по сегментам, а не по одному объекту.

ConcurrentLinkedQueue<E> - в реализации используется wait-free алгоритм от Michael & Scott, адаптированный для работы с garbage collector'ом. Этот алгоритм довольно эффективен и, что самое важное, очень быстр, т.к. построен на CAS.

Synchronizers

Semaphore - семафоры чаще всего используются для ограничения количества потоков при работе с аппаратными ресурсами или файловой системой. Доступ к общему ресурсу управляется с помощью счетчика. Если он больше нуля, то доступ разрешается, а значение счетчика уменьшается. Если счетчик равен нулю, то текущий поток блокируется, пока другой поток не освободит ресурс.

CountDownLatch - позволяет одному или нескольким потокам ожидать до тех пор, пока не завершится определенное количество операций, выполняющихся в других потоках.

Synchronizers

CyclicBarrier - используется для синхронизации заданного количества потоков в одной точке. Барьер достигается когда N-потоков вызовут метод `await(...)` и заблокируются. После чего счетчик сбрасывается в исходное значение, а ожидающие потоки освобождаются.

Exchanger<V> - используется для обмена объектами между двумя потоками.

Executor Services

Executor - базовый интерфейс для классов, реализующих запуск **Runnable** задач. Тем самым обеспечивается развязка между добавлением задачи и способом её запуска.

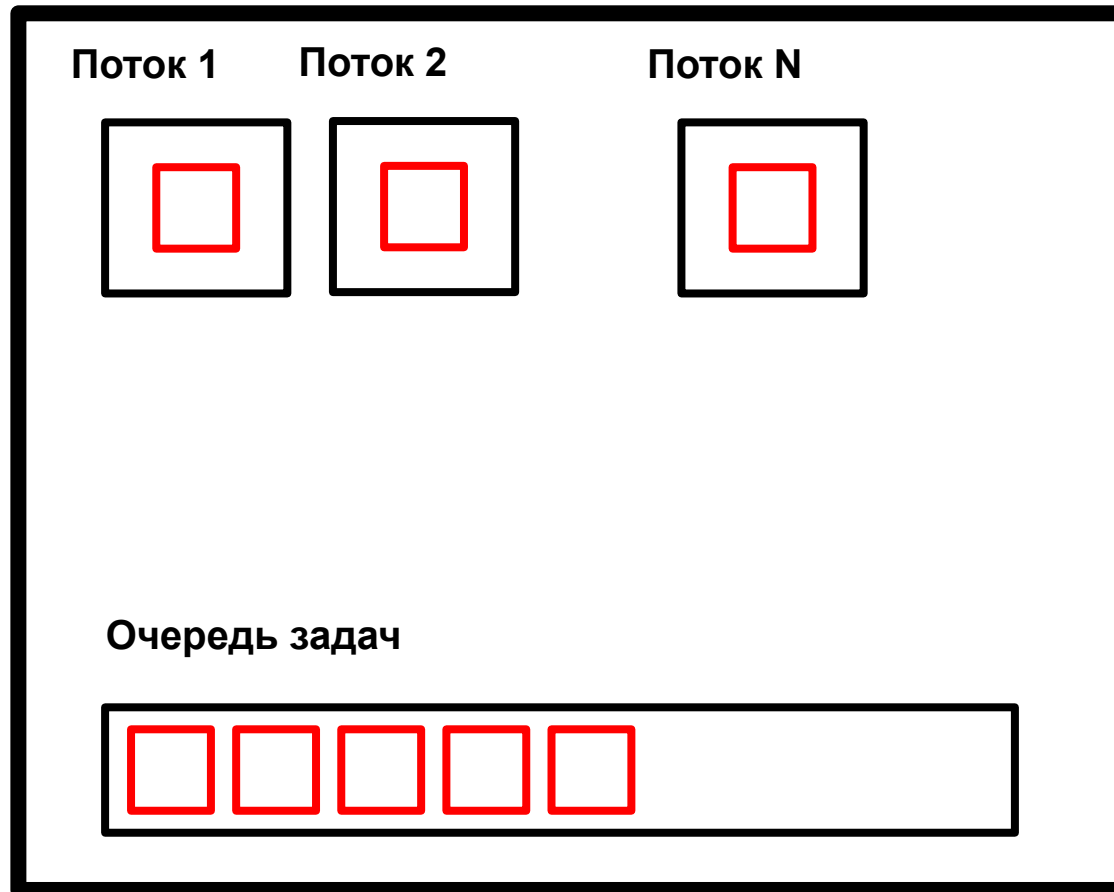
ExecutorService - интерфейс, который описывает сервис для запуска **Runnable** или **Callable** задач. Методы **submit** на вход принимают задачу в виде **Callable** или **Runnable**, а в качестве возвращаемого значения идет **Future**, через который можно получить результат.

Executors - класс-фабрика для создания **ThreadPoolExecutor**, **ScheduledThreadPoolExecutor**.

ThreadPoolExecutor - используется для запуска асинхронных задач в пуле потоков.

Thread pool

Thread pool



Future

Future<V> - замечательный интерфейс для получения результатов работы асинхронной операции.

```
elif _operation == "MIRROR_X":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
elif _operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True

#selection at the end -add ba
mirror_ob.select= 1
modifier_ob.select=1
bpy.context.scene.objects.active
print("Selected" + str(modifier_o
    #mirror_ob.select = 0
    time = bpy.context.select
    time = bpy.context.select
```


Callable

Callable<V> - расширенный аналог интерфейса **Runnable** для асинхронных операций. Позволяет возвращать типизированное значение и кидать **checked exception**. Несмотря на то, что в этом интерфейсе отсутствует метод **run()**, многие классы **java.util.concurrent** поддерживают его наряду с **Runnable**.

КРОК

ИНТЕГРИРУЕМ
БУДУЩЕЕ