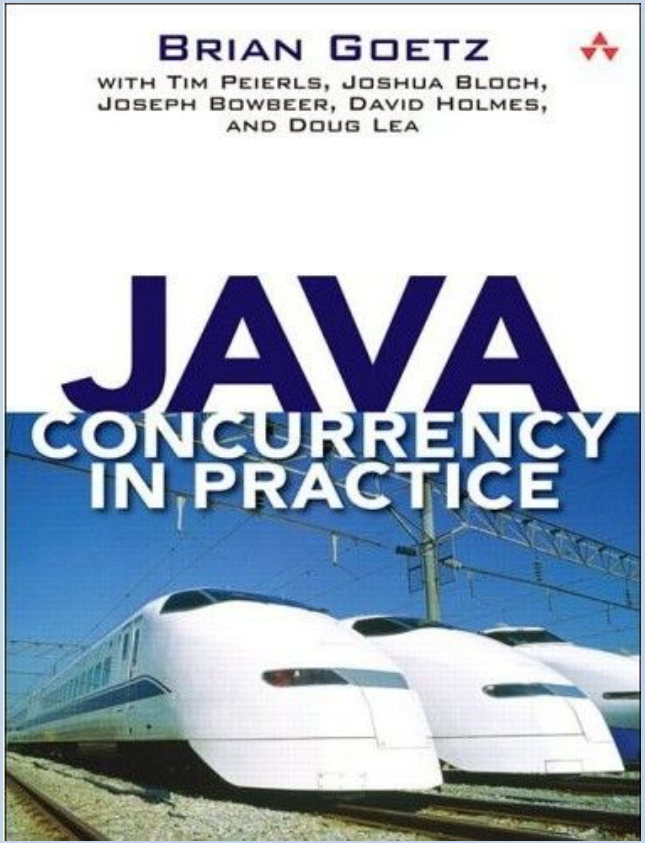
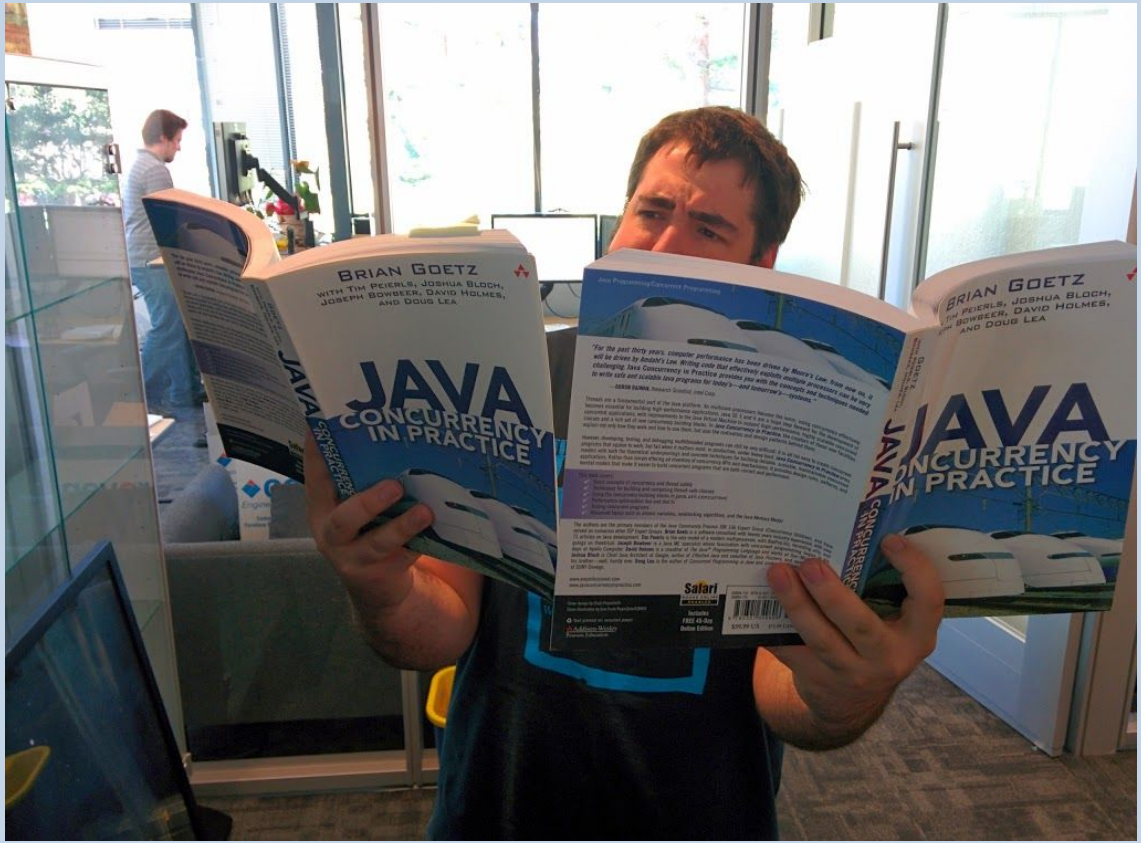

# Параллельное программирование

# Параллельное программирование



# The correct way to read "Java: Concurrency in Practice."



# Введение

*— Папа, а что такое многозадачность в Windows 95?*
*— Подожди, сынок, сейчас дискету доформатирую и покажу.*

Бородатый анекдот

Эволюция многопоточности:

1. Нет ОС, исполняется только одна программа, доступ ко всем ресурсам
2. ОС, запуск нескольких программ сразу:
    a. Более оптимальное использование ресурсов
    b. “Честная” параллелизация нескольких программ
    c. Одна программа - одна сфера ответственности (этакий “SOA vs монолит” на одном компьютере)
3. Разделение процессов на потоки

# Введение

**Процесс** - отдельная изолированная программа. Включает в себя:

- исходный исполняемый код
- собственная область виртуальной памяти, содержащая
  - исполняемый код
  - данные
  - стек вызовов
- дескрипторы ресурсов ОС: файлы, сокеты и пр.
- атрибуты доступа, разрешения на совершение действий в ОС
- состояние процессора (контекста) в ОС:
  - содержимое регистров
  - преобразование виртуальной памяти в физическую
  - и пр.

**Поток** - составная часть процесса, наименьшая единица обработки, которая может быть назначена ядром ОС. В отличие от процессов, потоки:

- являются частью процесса, заимствуют его исполняемый код, данные, ресурсы, доступы
- имеют собственный стек вызовов, содержимое регистров и локальные переменные

# Введение

Преимущества потоков:

- Простота при моделировании параллельной работы
- Более эффективное использование ресурсов (меньше накладных расходов при переключении контекста)
- Естественная единица параллелизации на многопроцессорных системах
- Возможность продолжать выполнение программы при блокирующих операциях (особенно актуально для GUI-программ)
- Простота реализации асинхронных взаимодействий (нет необходимости управлять всем в одном процессе)

# Введение

Недостатки потоков:

- Угрозы потокобезопасности (Thread Safety, корректность выполнения многопоточного кода): Race conditions
- Угрозы работоспособности (“liveness”): deadlock, starvation, livelock
- Накладные расходы при синхронизации и переключении контекстов

# Потокобезопасность

**Потокобезопасность (Thread Safety)** - это корректное выполнения кода в разных потоках вне зависимости от особенностей их выполнения в многопоточной среде и без дополнительной синхронизации со стороны вызываемого кода.

Главный враг - **разделяемое изменяемое состояние (shared mutable state)** - данные или ресурс.

Если несколько потоков могут получить доступ к разделяемому изменяемому состоянию без необходимой синхронизации, то ваша программа уже сломана! Рано или поздно она обязательно выдаст неправильный результат.

От этого можно защититься тремя способами:

1. Не предоставлять доступ к такому состоянию сразу нескольким потокам
2. Сделать состояние неизменяемым (immutable)
3. При каждом доступе к состоянию производить синхронизацию

Как правило, все эти вещи должны учитываться еще на этапе проектирования!

# Потокобезопасность

**Состояние гонки (race condition)** - ошибка проектирования, при которой работа программы зависит от того, в каком порядке выполняются части кода.

```
class UnsafeCounter {
  private var count = 0
  def getCount = count
  def increment(): Unit = { count += 1 }
}

class UnsafeCounterWithCaching {
  private val counter = new AtomicLong()
  private val counterCache = new AtomicReference[Long]()

  def getCount = counterCache.get()

  def increment(): Unit = {
    val newValue = counter.incrementAndGet()
    counterCache.set(newValue)
  }
}
```

# Потокобезопасность

Способы обеспечения потокобезопасности классов:

1. Отсутствие состояния. При выполнении такого кода все переменные являются локальными, а класс - всегда потокобезопасным.
2. Атомарные изменения. Атомарное изменение либо выполняется целиком, либо не выполняется вовсе с точки зрения других потоков. См. `java.util.concurrent.atomic`
3. Блокировки. Выручают там, где надо производить сразу несколько изменений атомарно.

# Потокобезопасность

**Встроенные блокировки (intrinsic locks, monitor locks):**

Java предоставляет встроенный механизм синхронизации по произвольному объекту - блок `synchronized`. Любой код, выполненный внутри `obj.synchronized{}`, будет синхронизирован по объекту `obj`, то есть ни один другой поток не сможет в это время получить lock на этот объект:

```
def increment(): Unit = this.synchronized {
   // Do non-thread-safe stuff
 }
```

Блокировка ассоциируется с потоком, поэтому поток может несколько раз входить в блоки, защищенные одним и тем же объектом.
Тем не менее, взятая блокировка по объекту не ограничивает действия, выполняемые с самим объектом другими потоками (вызов его методов, работа с его состоянием), вне блоков `synchronized`.

# Потокобезопасность

При защите мутабельного состояния блокировками такие блокировки необходимо добавить **везде**, где происходит доступ к этому состоянию (в том числе и его чтение)!

А если защищаемое состояние включает в себя несколько переменных, которые должны изменяться согласованно, то изменение любой переменной должно быть защищено одной и той же блокировкой.

# Потокобезопасность

Использование блокировок сильно влияет на **производительность**. Отсюда чем меньше времени процесс проводит в критической секции - тем больше процессов будет выполняться за единицу времени. Поэтому размер критической секции должен быть минимальным, но при этом достаточным для обеспечения thread safety.

Тем более не стоит включать долгие операции в критические секции (особенно работу с диском, БД или сетью).

# Совместное использование объектов

В однопоточных приложениях с этим все просто: сделали изменение и оно сразу стало видно в коде ниже. Однако, для многопоточного кода это не так.

При работе с несколькими потоками, выполняющимися на разных процессорах/ядрах с собственными кешами **проблема видимости** становится очень важной: как быстро один поток/процессор увидит изменения, сделанные в другом потоке/процессоре? Кто и как контролирует это?

Возможные проблемы из-за ошибок при работе с видимостью изменений:

1. Устаревшие данные (stale data)
2. Неатомарные операции с 64-битными типами данных (double, long), не объявленными как volatile

# Совместное использование объектов

```
object VisibilityProblem extends App {
//  @volatile // Uncomment this to fix the problem
  private var number = 0

  private class ReaderThread extends Thread {
    override def run(): Unit = {
      var lastValue = number
      while (true) {
        if (lastValue != number) {
          lastValue = number
          println(s"Got change to $lastValue")
        }
//        Thread.sleep(1) // Or uncomment this
      }
    }
  }

  private class WriterThread extends Thread {
    override def run(): Unit = {
      for (_ <- 1 to 5) {
        println(s"Incremented number to $number")
        number += 1
        Thread.sleep(100)
      }
    }
  }

  new ReaderThread().start()
  new WriterThread().start()
}
```

# Совместное использование объектов

Самый простой способ устранить проблему видимости переменной - объявить ее как volatile (аннотация `@volatile`). Это подскажет компилятору, что значения данной переменной нельзя кешировать, а надо постоянно брать из памяти и записывать так же сразу в нее.

Использование блоков `synchronized`, а так же volatile-переменных автоматически гарантирует видимость всех изменений, сделанных до выхода из блока/до записи в volatile-переменную. Например:

1. Поток А присваивает переменной `foo` значение 42
2. Поток А входит в блок `synchronized` по переменной M
3. Поток А присваивает переменной `bar` значение 13
4. Поток А выходит из блока `synchronized`
5. Поток Б входит в блок `synchronized` по переменной M
6. Начиная с этого момента, поток Б гарантированно видит все изменения переменных `foo` и `bar`

# Совместное использование объектов

Важной проблемой потокобезопасности является **утечка данных** (escaped objects/variables). Утечка возникает в случаях, когда ссылка на мутабельные данные покидает свой исходный объект, из-за чего нарушается инкапсуляция этого объекта и возникает угроза несанкционированного изменения его внутреннего состояния.

```
class Prison {
  private val prisonerList = Array(0)
  def getPrisoners: Array[Int] = prisonerList
}

object Mallory extends App {
  val prison = new Prison
  println(prison.getPrisoners.toList) // List(0)
  prison.getPrisoners.update(0, 42)
  println(prison.getPrisoners.toList) // List(42)
}
```

# Совместное использование объектов

Другие специальные случаи побега ссылок:

- Утечка ссылок на класс в процессе инициализации:

```
object LeakedNonInitializedClass {
  @volatile var sharedLink: MyClass = _
  class MyClass {
    sharedLink = this
    // do some initialization
  }
}
```

- Утечка указателя на внешний класс для вложенных классов (особенно если конструктор еще не завершил свою работу):

```
class LeakedClass {
  Application.registerListener(
    new Listener {
      override def handleEvent(e: Event): Unit = {
        // reference to LeakedClass is leaked
      }
    }
  )
}
```

# Совместное использование объектов

Простейшие способы обхода проблемы видимости и совместного использования объектов:

1. **Ограничение текущим потоком (thread confinement):** не потокобезопасная переменная используется только в одном (текущем) потоке
2. **ThreadLocal-переменные.** Одноименный контейнер позволяет сохранить ссылки на экземпляры объектов таким образом, что каждый поток будет иметь собственную ссылку, не пересекающуюся с другими потоками. Это - простой способ привязать переменную только к одному потоку.
3. **Иммутабельность (immutability):** если объект является неизменяемым (иммутабельным), то он автоматически обладает свойством потокобезопасности. При этом во время своего создания он не обязан быть неизменяемым (при условии, что он станет опубликованным для других объектов уже будучи полностью сконструированным).
    a. Иммутабельный объект внутри себя может иметь мутабельные объекты, если при этом он не предоставляют наружу ссылки на них или методы для их изменения

# Совместное использование объектов

Правильная инициализация объектов очень важна, ибо даже валидный объект при условии неправильной инициализации может быть источником ошибок. Основные проблемы:

- Публикация ссылок на объект до завершения его создания
- Устаревшие ссылки на неопубликованный объект

```
class Holder() {
  var n: Int = _
  Application.register(this)
  n = 42
  def assertSanity() = if (n != n) throw new AssertionError("This statement is false.")
}

class Application {
  var holder: Holder = _
  def initialize (): Unit = {
    holder = new Holder()
  }
}
```

# Совместное использование объектов

Пример безопасной инициализации с двойной проверкой:

```
class SafeSingletonFactory {
  @volatile
  private var instance: MySingleton = null

  def get: MySingleton = {
    if (instance == null) { // check 1
      synchronized {
        if (instance == null) { // check 2
          instance = new MySingleton
        }
      }
    }
    instance
  }
}
```

# Угрозы работоспособности (liveness hazards)

Потокобезопасность обычно конфликтует с работоспособностью: улучшение одного ведет к ухудшению другого.

Всего есть 3 основные угрозы для работоспособности приложений:

1. Блокировки (deadlocks) - потоки заблокировались и не могут выполняться вообще
2. Голодание (starvation) - бесконечный отказ от продвижения вперед из-за низкого приоритета
3. Livelock - зацикливание с возвратом назад из-за взаимного конфликта (например, попытка пройти в узкий проем с двух сторон одновременно)

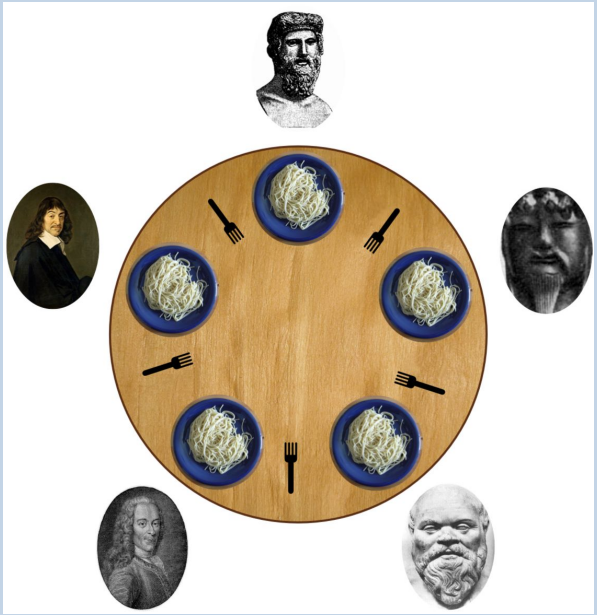
Как правило, все эти проблемы не проявляются сразу, а обнаруживают себя только в самое неподходящее время - в пиковые нагрузки.

# Угрозы работоспособности (liveness hazards)

Классический пример **блокировки** - “обедающие философы”.
Дедлоком здесь будет вариант, когда каждый философ взял по одной вилке и ждет освобождения второй.

Как избегать блокировок:

1. Ввести четкий порядок взятия ресурсов:
   сначала более крупные блокировки, потом мелкие;
   одинаковые по размеру - брать по возрастанию номера ресурса.
2. Использовать внешнего арбитра, который может предвидеть и предотвращать блокировки
3. Минимизировать длительность критических секций, особенно не делать внешние вызовы из них
4. В случаях, когда идет борьба за ограниченный пул (например, пул соединений к БД) не создавать зависимость выполнения одного элемента от другого



# Угрозы работоспособности (liveness hazards)

Еще один вариант избегания локов: брать лок с таймаутом, откатом и переповтором (залогировать ошибки также не помешает).

В случае обнаружения локов на уже запущенном приложении очень полезно снять thread dump, который покажет список всех потоков и их стектрейсы. Так же утилита снятия дампа попробует самостоятельно найти какие-то дедлоки.

# Угрозы работоспособности (liveness hazards)

**Голодание (starvation)** - это бесконечный отказ от продвижения вперед из-за низкого приоритета.

Самый простой пример для любой ОС - поток с повышенным приоритетом съедает весь CPU, фактически останавливая потоки с меньшим приоритетом.
JVM позволяет менять приоритеты у потоков, однако это приводит к платформозависимости и повышенному риску starvation-а.

Отдельным случаем является снижение производительности из-за фонового процесса, отъедающего основные ресурсы. В этом случае изменение (понижение) приоритета его потоков имеет смысл.

# Угрозы работоспособности (liveness hazards)

**Livelock** - зацикливание с возвратом назад из-за взаимного конфликта (например, попытка пройти в узкий проем с двух сторон одновременно).

Где с этим можно столкнуться:

1. При обработке очереди входящих сообщений, когда во время обработки возникает ошибка, сообщение откатывается, кладется обратно в начало очереди и снова берется на обработку. И так до бесконечности. Решается четким разделением ошибок на восстановимые и невосстановимые или внедрением счетчика повторений (желательно с экспоненциальным увеличением интервала повторов)

2. Борьба за общий набор ресурсов с откатом и одинаковым интервалом переповтора. Решается введением порядка взятия локов, а так же рандомизацией интервала переповтора.

# Как лучше всего готовить многопоточность?

# Как лучше всего готовить многопоточность?

Готовить лучше всего из готовых блоков.
К счастью, Java предоставляет нам широкий выбор.

1. Синхронизированные коллекции (все методы являются synchronized):
   a. Vector
   b. HashTable
   c. java.util.Collections.synchronizedXXX()

Проблемы:
- любая комбинация действий является неатомарной и требует дополнительной синхронизации
- итерирование небезопасно в случае возможных параллельных изменений; итераторы являются fail-fast (бросают ConcurrentModificationException в случае обнаружения изменений)
  - можно безопасно итерироваться по копии коллекции
- надо помнить про скрытое итерирование (например, в toString)

# Как лучше всего готовить многопоточность?

2. Параллельные (concurrent) коллекции:
    a. Map => ConcurrentHashMap
    b. List => CopyOnWriteArrayList
    c. Queue / Deque => ConcurrentLinkedQueue / ConcurrentLinkedDeque
    d. SortedMap / SortedSet => ConcurrentSkipListMap / ConcurrentSkipListSet

Основные свойства:

- Параллельное использование (особенно если много чтений)
- Атомарные операции: put-if-absent, replace-if-equal, remove-if-equal
- При итерировании нет fail-fast с ConcurrentModificationException, вместо этого weakly consistent
- size и isEmpty вычисляются приблизительно
- Queue / BlockingQueue / PriorityBlockingQueue / SynchronousQueue позволяют реализовать Producer-consumer шаблон в различных вариациях
- Deque позволяют организовать work stealing

# Как лучше всего готовить многопоточность?

3. Примитивы синхронизации:
    a. Latch - приостановить выполнение потоков до возникновения события
        i. `CountDownLatch`
        ii. `FutureTask`
    b. Семафоры (мьютексы) - ограничить количество одновременно используемых ресурсов
        i. `Semaphore`
    c. Барьеры - синхронизировать выполнение потоков на определенном этапе
        i. `CyclicBarrier`
    d. Явные блокировки
        i. `ReentrantLock` - включая `tryLock(timeout)`, не забываем отпускать только внутри блока `finally`
        ii. `ReadWriteLock`

# Работа с пулами потоков

**Task** - абстракция (асинхронно) выполняемой задачи. Концепция тасков позволяет абстрагировать код выполняемой задачи от деталей её выполнения.

Идеальный таск обладает следующими свойствами:

- независимость от состояния, результатов и сайд-эффектов других тасков
- небольшой размер относительно доступных вычислительных мощностей

Примеры:

- запрос веб-сервера, email или file сервера
- небольшой изолированный кусок общего вычисления

# Работа с пулами потоков

Последовательное исполнение тасков:

```
object SingleThreadWebServer {
  def handleRequest(connection: Socket) = ???
  def main(args: Array[String]): Unit = {
    val socket = new ServerSocket(80)
    while (true) {
      val connection = socket.accept
      handleRequest(connection)
    }
  }
}
```

- в каждый момент времени исполняется только один запрос
- остальные запросы простаивают
- при блокирующих операциях ресурсы тратятся впустую (CPU/memory/disk)
- долгая реакция сервера на входящие запросы (responsiveness)

# Работа с пулами потоков

Исполнение тасков в отдельных тредах:

```
object ThreadPerTaskWebServer {
  def handleRequest(connection: Socket) = ???
  def main(args: Array[String]): Unit = {
    val socket = new ServerSocket(80)
    while (true) {
      val connection = socket.accept
      val task = new Runnable() {
        override def run(): Unit = handleRequest(connection)
      }
      new Thread(task).start()
    }
  }
}
```

- мгновенное принятие запросов на исполнение
- параллельная обработка запросов (высокая пропускная способность)
- необходима потокобезопасность
- нет верхнего ограничения на количество потоков

# Работа с пулами потоков

Исполнение тасков с применением Executor-ов:

```
object TaskExecutionWebServer {
  def handleRequest(connection: Socket) = ???
  def main(args: Array[String]): Unit = {
    val executer = Executors.newFixedThreadPool(100)
    val socket = new ServerSocket(80)
    while (true) {
      val connection = socket.accept
      val task = new Runnable() {
        override def run(): Unit = handleRequest(connection)
      }
      executer.execute(task)
    }
  }
}
```

- все плюшки многопоточной обработки
- гибкая настройка пула потоков исполнения запросов
- встроенный мониторинг состояния пула и очереди задач

# Работа с пулами потоков

Executor-ы предоставляют возможность гибкой настройки исполнения тасков:

- в каких потоках задачи будут исполняться
- в каком порядке они будут исполняться? (FIFO, LIFO, priority queue)
- сколько тасков будет исполняться одновременно
- какова длина очереди тасков на исполнение
- что должно происходить при переполнении очереди
- что необходимо сделать перед/после выполнения таска

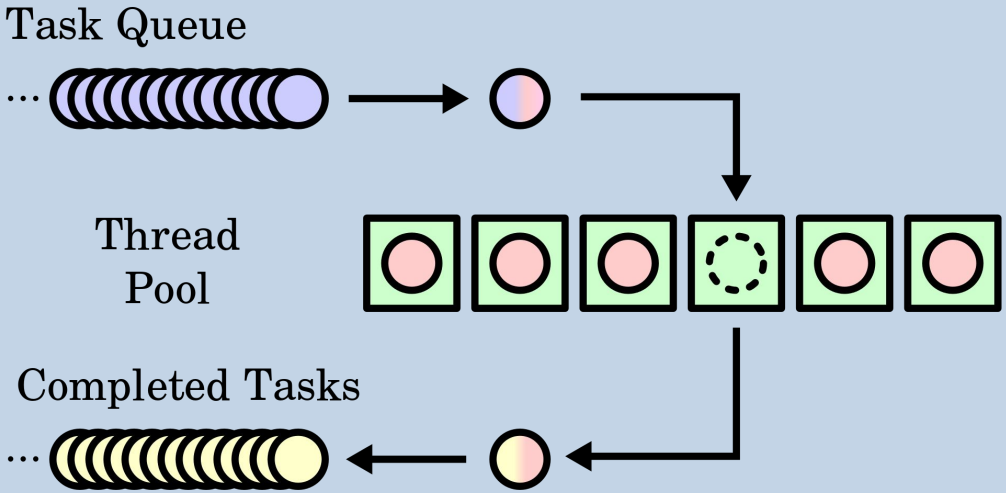
Таким образом, executor-ы позволяют гибко управлять следующими свойствами приложения:

- потребление ресурсов (CPU/memory/disk/network/DB)
- отзывчивость (responsiveness)
- пропускная способность (throughput)
- скорость выполнения отдельных запросов (performance)

# Работа с пулами потоков

Executor-ы активно используют **пулы потоков** - абстракция над множеством потоков, предназначенных для выполнения отдельных задач.

Основное достоинство пулов потоков - экономия ресурсов и времени на создании/уничтожении потоков при выполнении отдельных задач.



# Работа с пулами потоков

Основные виды пулов потоков, предоставляемые фабричными методами Executors:

- newFixedThreadPool - фиксированный размер пула
- newCachedThreadPool - гибко расширяемый размер с постепенным завершением свободных тредов
- newSingleThreadExecutor - однопоточный пул
- newScheduledThreadPool - пул с поддержкой отложенного и периодического запуска задач
- newWorkStealingPool - пул с поддержкой work stealing, когда у каждого треда есть собственная очередь, при опустошении которой тред может забирать задачи из чужих очередей

# Работа с пулами потоков

JVM работает до тех пор, пока есть хотя бы один не-daemon поток. Отсюда возникает проблема завершения потоков, а вместе с ними - и пулов потоков.

Жизненный цикл Executor-а (ExecutorService) имеет 3 состояния:

- Running - Executor работает и выполняет входящие таски
- Shutting down - Executor выполняет graceful shutdown: больше не принимает новые таски, однако завершает все текущие (включая те, что еще не запустились)
- Terminated - Executor полностью завершил работу

```
trait ExecutorService extends Executor {
  def shutdown(): Unit
  def shutdownNow: Nothing
  def isShutdown: Boolean
  def isTerminated: Boolean
  def awaitTermination(timeout: Long, unit: TimeUnit): Boolean
  //  ...
}
```

# Работа с пулами потоков

Остановка и завершение потока.

JVM не предоставляет механизмов безопасной остановки/завершения потоков.
Есть только контракт, следуя которому можно выстроить соответствующий процесс.
(ранее были методы Thread.stop и Thread.suspend, однако теперь они deprecated)

Зачем вообще может понадобиться остановить таску:

- Пользователь запросил отмену выполняемого действия
- Выполнение действия превысило заданный временной лимит
- Отпала необходимость в результате вычисления: найдено альтернативное решение, обнаружена ошибка в соседних тасках
- Завершение приложения/сервиса

# Работа с пулами потоков

```
public class Thread {
    public void interrupt() { ... }
    public boolean isInterrupted() { ... }
    public static boolean interrupted() { ... }
    ...
}
```

Принцип работы прерывания потока:

- инициатор вызывает метод interrupt(), который устанавливает флаг isInterrupted()
- если поток заблокирован (Thread.sleep, Object.wait etc.), он автоматически разблокируется, флаг isInterrupted очищается и кидается InterruptedException
- если же поток не был заблокирован, то выполняемый код должен сам следить за флагом isInterrupted и, если он установлен, инициировать завершение работы с правильной очисткой ресурсов

# Работа с пулами потоков

Таким образом, если выполняющийся таск поймает прерывание, он должен правильно передать это прерывание выполняющему его потоку (выбросить исключение InterruptedException или установить флаг isInterrupted (если он его случайно очистил).

Однако, не все стандартные блокирующиеся методы прерываются:

- Синхронный IO в java.io: read/write в InputStream/OutputStream
- Intrinsic-блокировки (альтернатива - метод Lock.lockInterruptibly())

Также некоторые другие методы бросают собственные специфические исключения.

# Работа с пулами потоков

Остановка JVM.

JVM останавливается штатно в случаях, если:

- все не-daemon потоки завершили свою работу
- был вызван System.exit() или Runtime.exit()
- был получен сигнал SIGINT (Ctrl+C)

JVM останавливается экстренно в случаях, если:

- был вызван Runtime.halt()
- был получен сигнал SIGKILL

# Работа с пулами потоков

При штатной остановке:

1. Запускаются все хуки, зарегистрированные вызовом Runtime.addShutdownHook()
    - хуки запускаются в отдельных тредах, их порядок выполнения не определен
    - если необходимо учесть порядок завершения, лучше это делать в одном хуке
    - необходима предельная осторожность: нет никаких гарантий относительно состояния любого сервиса, он может быть как рабочим, так и завершенным
2. После выполнения хуков JVM останавливается; если остались незавершенные не-daemon-потоки, они просто завершаются.
3. Daemon-потоки так же просто завершаются, без вызова finally-блоков и разворачивания стека => они подходят только для служебных задач, выполняемых в оперативной памяти.

# Подводные камни при работе с пулами потоков

Необходимо помнить о следующих возможных проблемах:

- Неявная зависимость между тасками
  - Thread Starvation Deadlock
- Недостаточная потокобезопасность тасков, не проявляющаяся на однопоточных пулах
- Чувствительность к времени выполнения
- Чрезмерное использование ThreadLocal-переменных, передача в них состояния между тасками

# Подводные камни при работе с пулами потоков

**Thread Starvation Deadlock** - блокировка, возникающая в случае, если таска А ожидает результата выполнения таски Б, в то время как таска Б не может начать выполняться из-за переполненного пула.

Такой же отдельным подвидом такой блокировки является блокировка из-за ожидания освобождения других ресурсов, например пула БД (размер которого может быть сильно меньше размера первого пула).

**Долго выполняющиеся таски** (особенно если они часто блокируются на внешних ресурсах) могут создать проблему отзывчивости пула потоков и сильно затормозить его производительность. В таких случаях рекомендуется блокироваться с таймаутом и в случае его срабатывания освобождать поток для других тасков.

# Подводные камни при работе с пулами потоков

**Оптимальный размер пула** зависит как от выполняемых задач, так и от свойств системы, в которой они выполняются:

- Для вычислительно-сложных задач (нужно много CPU) рекомендуется брать (Number_of_CPU +1) потоков
- Для задач с ограниченным I/O можно рассчитать количество потоков как: Number_of_CPU * CPU_Utilization * (1 + Wait_Time / Compute_Time)
- При ограниченном ресурсе количество потоков можно рассчитать как: Total_Available_Resources / Resources_per_Task
- Если таски используют другие пулы потоков, то размеры пулов взаимозависимы

# Подводные камни при работе с пулами потоков

**Конфигурация пулов потоков.**

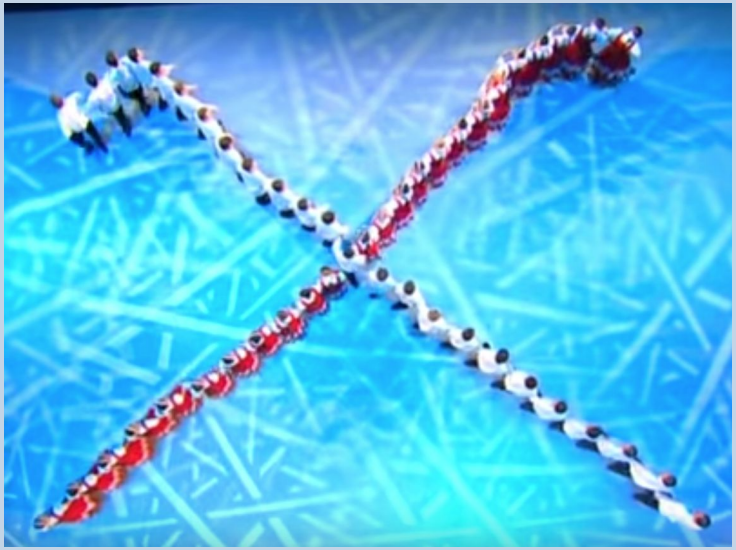
Пулы потоков имеют следующие основные параметры:

- corePoolSize - целевой размер пула
- maximumPoolSize - максимальное число активных потоков
- keepAliveTime - ограничение длительности бездействия потока до его остановки

Несколько важных моментов:

- изначально пулы пусты, потоки создаются только после первого таска (если не задан принудительный старт)
- если у очереди тасков задан размер, то потоки сверх corePoolSize создаются только в случае, когда очередь тасков переполнена
- при переполнении очереди и потоков в дело вступает RejectedExecutionHandler, возможные имплементации: AbortPolicy, CallerRunsPolicy, DiscardPolicy, DiscardOldestPolicy.

# Неблокирующая синхронизация



Проблемы с традиционными блокировками: приостановка выполнения потока и, как минимум, двойная смена контекста.

Однако, если нам не нужно иметь длинную критическую секцию, а достаточно всего лишь надежно менять одно единственное значение, на помощь приходят атомики - переменные с атомарными операциями (а значит - потокобезопасными).

Один из видов деления блокировок:

- **пессимистичные** - защищаемся от худшего (“придут гномы и все разрушат”)
- **оптимистичные** - выполняем действие в надежде, что никто нам не помешает (“вместо того, чтобы просить разрешения, проще сделать и попросить прощение”)

# Неблокирующая синхронизация

```
trait AtomicInteger extends Number {
  def get: Int
  def set(newValue: Int): Unit
  def getAndSet(newValue: Int): Int
  def compareAndSet(expect: Int, update: Int): Boolean
  def getAndIncrement: Int
  def getAndDecrement: Int
  def getAndAdd(delta: Int): Int
  def incrementAndGet: Int
  def decrementAndGet: Int
  def addAndGet(delta: Int): Int
  def getAndUpdate(updateFunction: IntUnaryOperator): Int
  def updateAndGet(updateFunction: IntUnaryOperator): Int
  def getAndAccumulate(x: Int, accumulatorFunction: IntBinaryOperator): Int
  def accumulateAndGet(x: Int, accumulatorFunction: IntBinaryOperator): Int
}
```

# Неблокирующая синхронизация

**CAS (compare-and-swap)** - основная инструкция для атомарных изменений. CAS(A, B) говорит, что надо изменить текущее значение переменной на B только если там сейчас записано A. И все это делается атомарно.

На современных процессорах CAS выполняется одной инструкцией ЦП.

При использовании оптимистичных блокировок в худшем случае как минимум один поток сможет произвести свои изменения, остальные же будут пробовать еще раз. При этом классической блокировки потока не будет, поток будет самостоятельно определять, что же делать дальше:

- попробовать еще раз
- выполнить какое-либо действие по восстановлению
- ничего не делать (игнорировать)

# Ускоряем приложения

Ускорение приложения неизбежно увеличивает его сложность, а следовательно - и вероятность ошибок в потокобезопасности и блокировках.

Отсюда первый принцип ускорения приложений: сначала добейтесь корректности работы, лишь потом - ускоряйте.
И только если ускорение действительно необходимо исходя из ваших требований!

Производительность может быть измерена различными способами:

- Как быстро: время обработки, задержка
- Как много: емкость, пропускная способность

**Масштабируемость** - способность увеличения емкости или пропускной способности приложения при добавлении дополнительных ресурсов (CPU, память, диск, сеть).

# Ускоряем приложения

Оптимизация **для увеличения производительности** зачастую отличается от оптимизации **для масштабируемости**: более быстрый алгоритм, кеширование vs. способы распараллеливания алгоритма.

Практически любая оптимизация - это улучшение одного за счет ухудшения другого:

- производительность за счет потребляемой памяти (и наоборот)
- пропускная способность за счет дублирования и взаимной синхронизации
- практически всегда страдает простота и надежность

Еще раз: **Прежде, чем ускорять что-либо, необходимо быть уверенным, что это действительно нужно!** Погоня за производительностью - одна из самых распространенных причин ошибок в многопоточности.

А уж если решили ускорять - делайте это осознанно! Сначала измеряйте, потом меняйте, потом измеряйте еще раз!

# Ускоряем приложения

Задачи бывают строго последовательные и параллелизуемые.

Ускорение при помощи многопоточности можно получить только для параллелизуемых задач. Отсюда был выведен **закон Амдала** (**Amdahl'sLaw**):

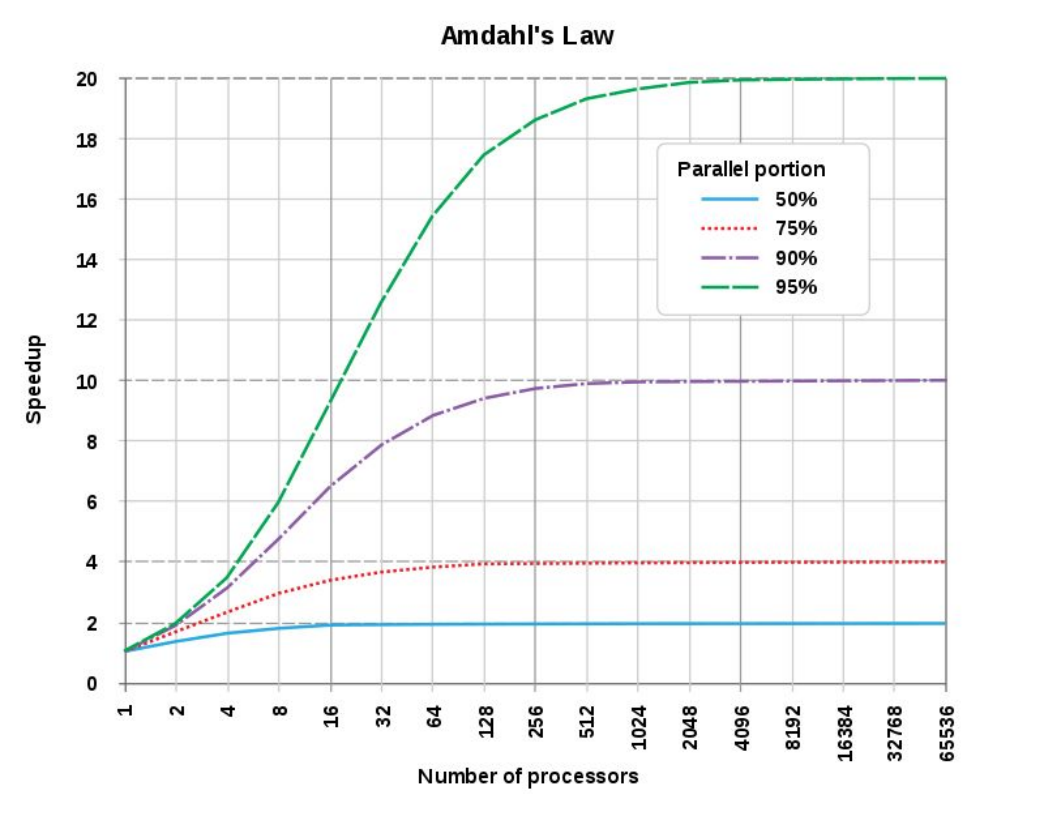
Пусть у нас есть N процессоров и F - доля строго последовательных вычислений в нашей программе. Тогда мы можем добиться следующего максимального ускорения:

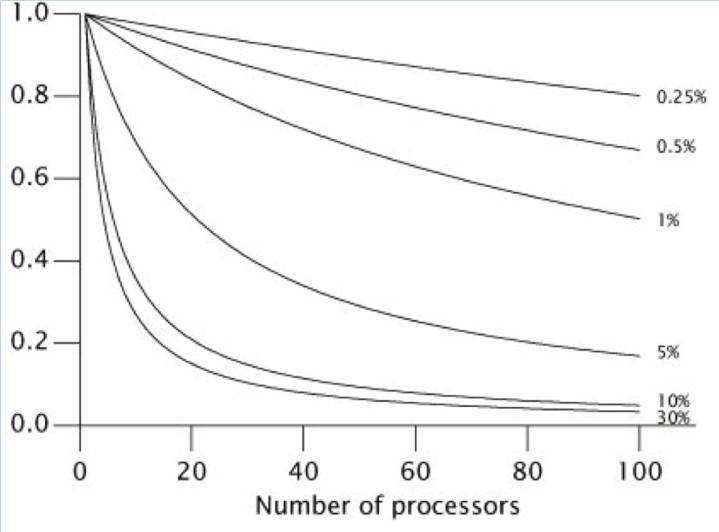
$$Speedup \leq \frac{1}{F + \frac{1 - F}{N}}$$

Примеры:

- F = 0.5, максимальное ускорение = 2 (для N = 2, S = 1.33; N = 4, S = 1.6)
- F = 0.1, максимальное ускорение = 10 (для N = 2, S = 1.81; N = 4, S = 3.07)

# Ускоряем приложения




**Слева** - ускорение, **справа** - утилизация одного процессора в зависимости от количества процессоров и степени параллелизации программы.

# Ускоряем приложения

Отсюда два важных вывода:

- нельзя распараллеливать бесконечно
- степень ускорения сильно зависит от того, какую часть вычислений можно выполнять параллельно

При этом у любой задачи есть вычисления, выполняемые последовательно!
Если вы их не нашли, значит плохо искали!

Чаще всего это - входящая очередь задач и обработка результата их выполнения.

# Ускоряем приложения

Расплата за многопоточность:

- переключение контекста (включая прогрев кешей для данных)
  - грубая оценка стоимости переключения контекста: 5К-10К тактов ЦП
- блокировки увеличивают количество переключений и не дают полностью использовать выделенный от ОС квант времени
- блокировки увеличивают межпоточное взаимодействие

JVM умеет оптимизировать некоторые случаи блокировок. Например:

- последовательная синхронизация на одном объекте
- синхронизация на объекте, ссылка на который не покидала поток

# Ускоряем приложения

В первую очередь, блокировки опасны тогда, когда из-за них блокируются потоки (contended locks). Вероятность потока быть заблокированным зависит от длительности использования используемых им блокировок.

Эту длительность можно уменьшить следующими способами:

- уменьшить длительность нахождения в критической секции
- уменьшить частоту входа в критическую секцию
  - декомпозиция блокировки на более мелкие составляющие
  - lock striping (например, как в ConcurrentHashMap)
  - избавление от hot fields путем кеширования и отказа от consistency
- заменить блокировку на другие инструменты синхронизации с лучшими характеристиками многопоточности (атомики, считающие семафоры и пр.)

# Ускоряем приложения

Пример оптимизации в условиях многопоточности - логирование.

Проблемы наивной реализации:

- общий расшаренный ресурс
- вывод часто и небольшими частями
- IO с большим временем отклика

Решение:

- все логирующие потоки сливают сообщения в один служебный поток
- служебный поток буферизует получаемые данные
- периодически происходит запись на диск большими кусками
- при остановке приложения буфер полностью пишется на диск

# Пара слов о тестировании

Основная проблема в тестировании многопоточных программ (по сравнению с тестированием однопоточных) - недетерминизм при работе, увеличивающий количество потенциальных комбинаций, которые надо предусмотреть и проверить.

Корректность и производительность, в целом, можно тестировать так же, но надо принимать во внимание некоторые особенности:

- вероятность возникновения ошибки сильно меньше => больше прогонов, сами прогоны должны быть дольше
- тест многопоточного класса - сам по себе многопоточный, в нем легко ошибиться
- проще всего выделить инварианты и проверять их после серии действий
- если применяются блокировки, нужно установить соответствующие таймауты-детекторы блокировки и предусмотреть механизм разблокирования
- нельзя смотреть на Thread.getState, так как он может не отражать реальное состояние
- Больше CPU и ядер, хороших и разных!
- Желательно контролировать отсутствие утечек памяти

# Пара слов о тестировании

**Тестирование производительности** - подразделяется на 2 основных блока:

- **Тестирование пропускной способности** - стоит брать среднее/медианное/перцентильное значение за период времени, так результат будет стабильней
- **Тестирование времени ответа** - стоит брать небольшие пачки запросов, чтобы нивелировать неточность системного таймера

# Пара слов о тестировании

Избегаем типичные ошибки (pitfalls):

- Внезапный Garbage Collection.
  Либо тестируем без его применения,
  либо с несколькими вызовами
- Динамическая компиляция. Может внезапно ускорить какой-то кусок => используем предварительный прогрев
- Нерелевантные сценарии. Мешают компилятору правильно оптимизировать.
- Искаженный профиль нагрузки. Нереалистичное рапределение нагрузки по коду приводит к тому, что оптимизируется не то, что “будет болеть” на бою.
- “Мертвый код” в сценариях. От которого JIT успешно избавляется. Результат вычислений должен как-то использоваться и не быть легко предсказуемым.

# Демо!

Демонстрация:

`lectures.concurrent.ConcurrentApplication`

`lectures.concurrent.ScalaMeterExample`

# Домашнее задание

Помогите вашему дяде:
`lectures.concurrent.BillionaireCounters`

Реализовать многопоточного поискового робота:
`lectures.concurrent.WebCrawler`