Многопоточность

[*Методичка по всем разделам, автор неизвестен (:*](https://docs.google.com/spreadsheets/d/1owGOr08-YSywk3KVIAdMLX8_tZyvPm9UJWubEeOcgD8/htmlview#)

*вопросы-ответы Хилькевич Игорь 15.04.2020-*

Вопросы из “all question”

1. Java Memory Model (JMM).

2. Потокобезопасность, конкуренция, параллелизм, корпоративная и вытесняющая многозадачность.

3. Ordering, as-if-serial semantics, sequental consistency, visibility,

atomicity, happens-before, mutual exclusion, safe publication.

4. Процесс vs. поток. Зеленые потоки.

5. Способы создания потока в java.

6. Thread vs. Runnable (start() vs. run()).

7. Как принудительно запустить поток? (никак)

8. Монитор в java.

9. Понятие синхронизации. Способы синхронизации в java.

10. Состояния потока.

11. private mutex.

12. wait(), notify(), notyfyAll()

13. sleep() vs. yield()

14. Thread.join()

15. Race condition, deadlock, livelock, starvation.

16. Thread.holdsLock()

17. volatile.

18. Atomic переменные.

19. Приоритет потока (от 1 до 10, default 5).

20. Потоки демоны.

21. Runnable vs. Callable.

22. FutureTask.

23. CyclicBarrier vs. CountDownLatch.

24. Thread.stop() vs interrupt(). interrupted() vs. isInterrupted().

25. Исключение в потоке.

26. Пулл потоков.

27. Executor, Executors.

28. Fork/Join framework.

29. Semaphore.

30. ThreadLocal-переменная.

31. synchronized vs. ReentrantLock.

32. ReedWriteLock.

33. Блокирующий метод.

34. Busy spin.

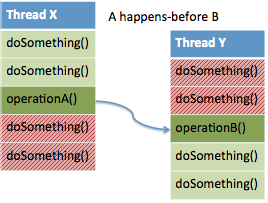
**Ответы**

1. Java Memory Model (JMM).

**Расскажите о модели памяти Java?**

Модель памяти Java (Java Memory Model, JMM) описывает поведение потоков в среде исполнения Java. Это часть семантики языка Java, набор правил, описывающий выполнение многопоточных программ и правил, по которым потоки могут взаимодействовать друг с другом посредством основной памяти.   
Program order - правило порядка выполнения программы.(Program order гарантирует, что в отдельных потоках оптимизация переупорядочения, введенная компилятором, не может привести к результатам, отличным от того, что произошло бы, если бы программа выполнялась послендовательно.)

Формально модель памяти определяет набор действий межпоточного взаимодействия (эти действия включают в себя, в частности, чтение и запись переменной, захват и освобождений монитора, чтение и запись volatile переменной, запуск нового потока), а также модель памяти определяет отношение между этими действиями -happens-before - абстракции обозначающей, что если операция A связана отношением happens-before с операцией B, то весь код следуемый за операцией B, выполняемый в одном потоке, видит все изменения, сделанные другим потоком, до операции A.



* Существует несколько основных правил для отношения happens-before:
* В рамках одного потока любая операция идущая в program order является - happens-before по отношению к любой операции следующей за ней в исходном коде;
* Освобождение монитора (unlock) happens-before захват того же монитора (lock);
* Выход из synchronized блока/метода happens-before вход в synchronized блок/метод на том же мониторе;
* Запись volatile поля happens-before чтение того же самого volatile поля;
* Завершение метода run() экземпляра класса Thread happens-before выход из метода join() или возвращение false методом isAlive() экземпляром того же потока;
* Вызов метода start() экземпляра класса Thread happens-before начало метода run() экземпляра того же потока;
* Завершение конструктора happens-before начало метода finalize() этого класса;
* Вызов метода interrupt() на потоке happens-before обнаружению потоком факта, что данный метод был вызван либо путем выбрасывания исключения InterruptedException, либо с помощью методов isInterrupted() или interrupted().
* Связь happens-before транзитивна, т.е. если X happens-before Y, а Y happens-before Z, то X happens-before Z.
* Освобождение/захват монитора и запись/чтение в volatile переменную связаны отношением happens-before, только если операции проводятся над одним и тем же экземпляром объекта.
* В отношении happens-before участвуют только два потока, о поведении остальных потоков ничего сказать нельзя, пока в каждом из них не наступит отношение happens-before с другим потоком.

Можно выделить несколько основных областей, имеющих отношение к модели памяти:

* Видимость (visibility). Один поток может в какой-то момент временно сохранить значение некоторых полей не в основную память, а в регистры или локальный кэш процессора, таким образом второй поток, выполняемый на другом процессоре, читая из основной памяти, может не увидеть последних изменений поля. И наоборот, если поток на протяжении какого-то времени работает с регистрами и локальными кэшами, читая данные оттуда, он может сразу не увидеть изменений, сделанных другим потоком в основную память.

К вопросу видимости имеют отношение следующие ключевые слова языка Java: synchronized, volatile, final.

С точки зрения Java все переменные (за исключением локальных переменных, объявленных внутри метода) хранятся в главной памяти, которая доступна всем потокам, кроме этого, каждый поток имеет локальную—рабочую—память, где он хранит копии переменных, с которыми он работает, и при выполнении программы поток работает только с этими копиями. Надо отметить, что это описание не требование к реализации, а всего лишь модель, которая объясняет поведение программы, так, в качестве локальной памяти не обязательно выступает кэш память, это могут быть регистры процессора или потоки могут вообще не иметь локальной памяти.

При входе в synchronized метод или блок поток обновляет содержимое локальной памяти, а при выходе из synchronized метода или блока поток записывает изменения, сделанные в локальной памяти, в главную. Такое поведение synchronized методов и блоков следует из правил для отношения «происходит раньше»: так как все операции с памятью происходят раньше освобождения монитора и освобождение монитора происходит раньше захвата монитора, то все операции с памятью, которые были сделаны потоком до выхода из synchronized блока должны быть видны любому потоку, который входит в synchronized блок для того же самого монитора. Очень важно, что это правило работает только в том случае, если потоки синхронизируются, используя один и тот же монитор!

Что касается volatile переменных, то запись таких переменных производится в основную память, минуя локальную. и чтение volatile переменной производится также из основной памяти, то есть значение переменной не может сохраняться в регистрах или локальной памяти потока и операция чтения этой переменной гарантированно вернёт последнее записанное в неё значение.

Также модель памяти определяет дополнительную семантику ключевого слова final, имеющую отношение к видимости: после того как объект был корректно создан, любой поток может видеть значения его final полей без дополнительной синхронизации. «Корректно создан» означает, что ссылка на создающийся объект не должна использоваться до тех пор, пока не завершился конструктор объекта. Наличие такой семантики для ключевого слова final позволяет создание неизменяемых (immutable) объектов, содержащих только final поля, такие объекты могут свободно передаваться между потоками без обеспечения синхронизации при передаче.

Есть одна проблема, связанная с final полями: реализация разрешает менять значения таких полей после создания объекта (это может быть сделано, например, с использованием механизма reflection). Если значение final поля—константа, чьё значение известно на момент компиляции, изменения такого поля могут не иметь эффекта, так-как обращения к этой переменной могли быть заменены компилятором на константу. Также спецификация разрешает другие оптимизации, связанные с final полями, например, операции чтения final переменной могут быть переупорядочены с операциями, которые потенциально могут изменить такую переменную. Так что рекомендуется изменять final поля объекта только внутри конструктора, в противном случае поведение не специфицировано.

* Reordering (переупорядочивание). Для увеличения производительности процессор/компилятор могут переставлять местами некоторые инструкции/операции. Вернее, с точки зрения потока, наблюдающего за выполнением операций в другом потоке, операции могут быть выполнены не в том порядке, в котором они идут в исходном коде. Тот же эффект может наблюдаться, когда один поток кладет результаты первой операции в регистр или локальный кэш, а результат второй операции попадает непосредственно в основную память. Тогда второй поток, обращаясь к основной памяти может сначала увидеть результат второй операции, и только потом первой, когда все регистры или кэши синхронизируются с основной памятью. Еще одна причина reordering, может заключаться в том, что процессор может решить поменять порядок выполнения операций, если, например, сочтет что такая последовательность выполнится быстрее.

Вопрос reordering также регулируется набором правил для отношения «происходит раньше» и у этих правил есть следствие, касающееся порядка операций, используемое на практике: операции чтения и записи volatile переменных не могут быть переупорядочены с операциями чтения и записи других volatile и не-volatile переменных. Это следствие делает возможным использование volatile переменной как флага, сигнализирующем об окончании какого-либо действия. В остальном правила, касающиеся порядка выполнения операций, гарантируют упорядоченность операций для конкретного набора случаев (таких как, например, захват и освобождение монитора), во всех остальных случаях оставляя компилятору и процессору полную свободу для оптимизаций.

**В чем заключаются различия между cтеком (stack) и кучей (heap) с точки зрения многопоточности?**

Cтек – участок памяти, тесно связанный с потоками. У каждого потока есть свой стек, которые хранит локальные переменные, параметры методов и стек вызовов. Переменная, хранящаяся в стеке одного потока, не видна для другого.

Куча – общий участок памяти, который делится между всеми потоками. Объекты, неважно локальные или любого другого уровня, создаются в куче. Для улучшения производительности, поток обычно кэширует значения из кучи в свой стек, в этом случае для того, чтобы указать потоку, что переменную следует читать из кучи используется ключевое слово volatile.

**Как поделиться данными между двумя потоками?**

Данными между потоками возможно делиться, используя общий объект или параллельные структуры данных, например BlockingQueue.

**Какой параметр запуска JVM используется для контроля размера стека потока?**

-Xss

**Как получить дамп потока?**

Среды исполнения Java на основе HotSpot генерируют только дамп в формате HPROF. В распоряжении разработчика имеется несколько интерактивных методов генерации дампов и один метод генерации дампов на основе событий.

Интерактивные методы:

* Использование Ctrl+Break: если для исполняющегося приложения установлена опция командной строки -XX:+HeapDumpOnCtrlBreak, то дамп формата HPROF генерируется вместе с дампом потока при наступлении события Ctrl+Break или SIGQUIT (обычно генерируется с помощью kill -3), которое инициируется посредством консоли. Эта опция может быть недоступна в некоторых версиях. В этом случае можно попытаться использовать следующую опцию: -Xrunhprof:format=b,file=heapdump.hprof
* Использование инструмента jmap: утилита jmap, поставляемая в составе каталога /bin/ комплекта JDK, позволяет запрашивать дамп HPROF из исполняющегося процесса.
* Использование операционной системы: Для создания файла ядра можно воспользоваться неразрушающей командой gcore или разрушающими командами kill -6 или kill -11. Затем извлечь дамп кучи из файла ядра с помощью утилиты jmap.
* Использование инструмента JConsole. Операция dumpHeap предоставляется в JConsole как MBean-компонент HotSpotDiagnostic. Эта операция запрашивает генерацию дампа в формате HPROF.

Метод на основе событий:

Событие OutOfMemoryError: Если для исполняющегося приложения установлена опция командной строки -XX:+HeapDumpOnOutOfMemoryError, то при возникновении ошибки OutOfMemoryError генерируется дамп формата HPROF. Это идеальный метод для «production» систем, поскольку он практически обязателен для диагностирования проблем памяти и не сопровождается постоянными накладными расходами с точки зрения производительности. В старых выпусках сред исполнения Java на базе HotSpot для этого события не устанавливается предельное количество дампов кучи в пересчете на одну JVM; в более новых выпусках допускается не более одного дампа кучи для этого события на каждый запуск JVM.

2. Потокобезопасность, конкуренция, параллелизм, корпоративная и

вытесняющая многозадачность.

**Что такое «потокобезопасность»?**

Потокобезопасность – свойство объекта или кода, которое гарантирует, что при исполнении или использовании несколькими потоками, код будет вести себя, как

предполагается. Например потокобезопасный счётчик не пропустит ни один счёт, даже если один и тот же экземпляр этого счётчика будет использоваться несколькими потоками.

**В чём разница между «конкуренцией» и «параллелизмом»?**

Конкуренция — это способ одновременного решения множества задач. 5

Признаки:

* Наличие нескольких потоков управления (например Thread в Java, корутина в Kotlin), если поток управления один, то конкурентного выполнения быть не может
* Недетерминированный(Независимый друг от друга) результат выполнения. Результат зависит от случайных событий, реализации и того как была проведена синхронизация. Даже если каждый поток полностью детерминированный, итоговый результат будет недетерминированным

Параллелизм — это способ выполнения разных частей одной задачи.

Признаки:

* Необязательно имеет несколько потоков управления
* Может приводить к детерминированному результату, так, например, результат умножения каждого элемента массива на число, не изменится, если умножать его по частям параллельно.

Что такое «кооперативная многозадачность»? Какой тип многозадачности использует Java? Чем обусловлен этот выбор?

Кооперативная многозадачность - это способ деления процессорного времени между потоками, при котором каждый поток обязан отдавать управление следующему добровольно.

Преимущества такого подхода - простота реализации, меньшие накладные расходы на переключение контекста.

Недостатки - если один поток завис или ведет себя некорректно, то зависает целиком вся система и другие потоки никогда не получат управление.

Java использует вытесняющую многозадачность, при которой решение о переключении между потоками процесса принимает операционная система.

В отличие от кооперативной многозадачности управление операционной системе передается вне зависимости от состояния работающих приложений, благодаря чему, отдельные зависшие потоки процесса, как правило, не «подвешивают» всю систему целиком. За счёт регулярного переключения между задачами также улучшается отзывчивость приложения и повышается оперативность освобождения ресурсов, которые больше не используются.

В реализации вытесняющая многозадачность отличается от кооперативной, в частности, тем, что требует обработки системного прерывания от аппаратного таймера.

3. Ordering, as-if-serial semanticsas-if-serial semanticsas-if-serial semanticsas-if-serial semantics, sequental consistency, visibility,

atomicity, happens-before, mutual exclusion, safe publication.

**Что такое ordering, as-if-serial semantics, sequential consistency, visibility, atomicity, happens-before, mutual exclusion, safe publication?**

ordering механизм, который определяет, когда один поток может увидеть out-of-order (неверный) порядок исполнения инструкций другого потока. CPU для для повышения производительности может переупорядочивать процессорные инструкции и выполнять их в произвольном порядке до тех пор пока для потока внутри не будет видно никаких отличий. Гарантия предоставляемая этим механизмом называется as-if-serial semantics.

sequential consistency - то же что и as-if-serial semantics, гарантия того, что в рамках одного потока побочные эффекты от всех операций будут такие, как будто все операции выполняются последовательно.

visibility определяет, когда действия в одном потоке становятся видны из другого потока.

happens-before - логическое ограничение на порядок выполнения инструкций программы. Если указывается, что запись в переменную и последующее ее чтение связаны через эту зависимость, то как бы при выполнении не переупорядочивались инструкции, в момент чтения все связанные с процессом записи результаты уже зафиксированы и видны.

atomicity — атомарность операций. Атомарная операция выглядит единой и неделимой командой процессора, которая может быть или уже выполненной или ещё невыполненной.

mutual exclusion (взаимоисключающая блокировка, семафор с одним состоянием) - механизм, гарантирующий потоку исключительный доступ к ресурсу. Используется для предотвращения одновременного доступа к общему ресурсу. В каждый момент времени таким ресурсом может владеть только один поток. Простейший пример: synchronized(obj) { … }.

safe publication? - показ объектов другим потокам из текущего, не нарушая ограничений visibility. Способы такой публикации в Java:

static{} инициализатор;

volatile переменные;

atomic переменные;

сохранение в разделяемой переменной, корректно защищенной с использованием synchronized(), синхронизаторов или других конструкций, создающих read/write memory barrier;

final переменные в разделяемом объекте, который был корректно проинициализирован.

4. Процесс vs. поток. Зелёные потоки.

**Чем отличается процесс от потока?**

Процесс — экземпляр программы во время выполнения, независимый объект, которому выделены системные ресурсы (например, процессорное время и память). Каждый процесс выполняется в отдельном адресном пространстве: один процесс не может получить доступ к переменным и структурам данных другого. Если процесс хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие. Это могут быть конвейеры, файлы, каналы связи между компьютерами и многое другое.

Для каждого процесса ОС создает так называемое «виртуальное адресное пространство», к которому процесс имеет прямой доступ. Это пространство принадлежит процессу, содержит только его данные и находится в полном его распоряжении. Операционная система же отвечает за то, как виртуальное пространство процесса проецируется на физическую память.

Поток(thread) — определенный способ выполнения процесса, определяющий последовательность исполнения кода в процессе. Потоки всегда создаются в контексте какого-либо процесса, и вся их жизнь проходит только в его границах. Потоки могут исполнять один и тот же код и манипулировать одними и теми же данными, а также совместно использовать описатели объектов ядра, поскольку таблица описателей создается не в отдельных потоках, а в процессах. Так как потоки расходуют существенно меньше ресурсов, чем процессы, в процессе выполнения работы выгоднее создавать дополнительные потоки и избегать создания новых процессов.

**Что такое «зеленые потоки» и есть ли они в Java?**

Зелёные (легковесные) потоки(green threads) - потоки эмулируемые виртуальной машиной или средой исполнения. Создание зелёного потока не подразумевает под собой создание реального потока ОС.

* Виртуальная машина Java берёт на себя заботу о переключении между разными green threads, а сама машина работает как один поток ОС. Это даёт несколько преимуществ. Потоки ОС относительно дороги в большинстве POSIX-систем. Кроме того, переключение между native threads гораздо медленнее, чем между green threads.

Это всё означает, что в некоторых ситуациях green threads гораздо выгоднее, чем native threads. Система может поддерживать гораздо большее количество green threads, чем потоков OС. Например, гораздо практичнее запускать новый green thread для нового HTTP-соединения к веб-серверу, вместо создания нового native thread.

Однако есть и недостатки. Самый большой заключается в том, что вы не можете исполнять два потока одновременно. Поскольку существует только один native thread, только он и вызывается планировщиком ОС. Даже если у вас несколько процессоров и несколько green threads, только один процессор может вызывать green thread. И всё потому, что с точки зрения планировщика заданий ОС всё это выглядит одним потоком.

Начиная с версии 1.2 Java поддерживает native threads, и с тех пор они используются по умолчанию.

5. Способы создания потока в java.

**Каким образом можно создать поток?**

* Создать потомка класса Thread и переопределить его метод run();
* Создать объект класса Thread, передав ему в конструкторе экземпляр класса, реализующего интерфейс Runnable. Эти интерфейс содержит метод run(), который будет выполняться в новом потоке. Поток закончит выполнение, когда завершится его метод run().
* Вызвать метод submit() у экземпляра класса реализующего интерфейс ExecutorService, передав ему в качестве параметра экземпляр класса реализующего интерфейс Runnable или Callable (содержит метод call(), в котором описывается логика выполнения).

6. Thread vs. RunnableThread vs. Runnable (start() vs. run()).

**Чем различаются Thread и Runnable?**

Thread - это класс, некоторая надстройка над физическим потоком.

Runnable - это интерфейс, представляющий абстракцию над выполняемой задачей.

Помимо того, что Runnable помогает разрешить проблему множественного наследования, несомненный плюс от его использования состоит в том, что он позволяет логически отделить логику выполнения задачи от непосредственного управления потоком.

**В чём заключается разница между методами start() и run()?**

Несмотря на то, что start() вызывает метод run() внутри себя, это не то же самое, что просто вызов run(). Если run() вызывается как обычный метод, то он вызывается в том же потоке и никакой новый поток не запускается, как это происходит, в случае, когда вы вызываете метод start().

7. Как принудительно запустить поток? (никак)

**Как принудительно запустить поток?**

Никак. В Java не существует абсолютно никакого способа принудительного запуска потока. Это контролируется JVM и Java не предоставляет никакого API для управления этим процессом.

8. Монитор в java.

**Что такое «монитор» в Java?**

Монитор, мьютекс (mutex) – это средство обеспечения контроля за доступом к ресурсу. У монитора может быть максимум один владелец в каждый текущий момент времени. Следовательно, если кто-то использует ресурс и захватил монитор для обеспечения единоличного доступа, то другой, желающий использовать тот же ресурс, должен подождать освобождения монитора, захватить его и только потом начать использовать ресурс.

Удобно представлять монитор как id захватившего его объекта. Если этот id равен 0 – ресурс свободен. Если не 0 – ресурс занят. Можно встать в очередь и ждать его освобождения.

В Java у каждого экземпляра объекта есть монитор, который контролируется непосредственно виртуальной машиной. Используется он так: любой нестатический synchronized-метод при своем вызове прежде всего пытается захватить монитор того объекта, у которого он вызван (на который он может сослаться как на this). Если это удалось – метод исполняется. Если нет – поток останавливается и ждет, пока монитор будет отпущен.

9. Понятие синхронизации. Способы синхронизации в java.Понятие синхронизации

**Дайте определение понятию «синхронизация».**

Синхронизация это процесс, который позволяет выполнять потоки параллельно.

В Java все объекты имеют одну блокировку, благодаря которой только один поток одновременно может получить доступ к критическому коду в объекте. Такая синхронизация помогает предотвратить повреждение состояния объекта. Если поток получил блокировку, ни один другой поток не может войти в синхронизированный код, пока блокировка не будет снята. Когда поток, владеющий блокировкой, выходит из синхронизированного кода, блокировка снимается. Теперь другой поток может получить блокировку объекта и выполнить синхронизированный код. Если поток пытается получить блокировку объекта, когда другой поток владеет блокировкой, поток переходит в состояние Блокировки до тех пор, пока блокировка не снимется.

**Какие существуют способы синхронизации в Java?**

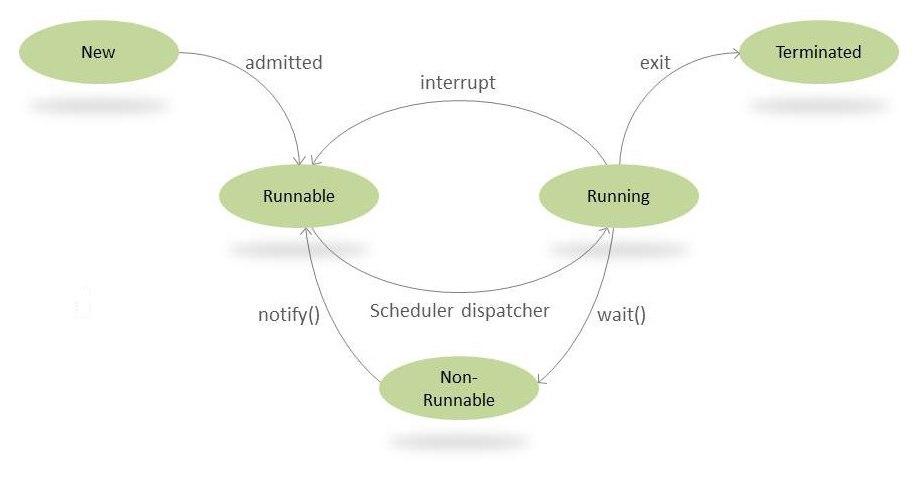
Системная синхронизация с использованием wait()/notify(). Поток, который ждет выполнения каких-либо условий, вызывает у этого объекта метод wait(), предварительно захватив его монитор. На этом его работа приостанавливается. Другой поток может вызвать на этом же самом объекте метод notify() (опять же, предварительно захватив монитор объекта), в результате чего, ждущий на объекте поток «просыпается» и продолжает свое выполнение. В обоих случаях монитор надо захватывать в явном виде, через synchronized-блок, потому как методы wait()/notify() не синхронизированы!

Системная синхронизация с использованием join(). Метод join(), вызванный у экземпляра класса Thread, позволяет текущему потоку остановиться до того момента, как поток, связанный с этим экземпляром, закончит работу.

Использование классов из пакета java.util.concurrent, который предоставляет набор классов для организации межпоточного взаимодействия. Примеры таких классов - Lock, Semaphore и пр.. Концепция данного подхода заключается в использовании атомарных операций и переменных.

10. Состояния потока.

**В каких состояниях может находиться поток?**



Потоки могут находиться в одном из следующих 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. Когда поток находится в этом состоянии, он считается живым.
* Работающий (Running). Поток переходит из состояния Работоспособный в состояние Работающий, когда Планировщик потоков выбирает его как работающий в данный момент.
* Живой, но не работоспособный (Alive, but not runnable). Поток может быть живым, но не работоспособным по нескольким причинам:
* Ожидание (Waiting). Поток переходит в состояние Ожидания, вызывая метод wait(). Вызов notify() или notifyAll() может перевести поток из состояния Ожидания в состояние Работоспособный.
* Сон (Sleeping). Метод sleep() переводит поток в состояние Сна на заданный промежуток времени в миллисекундах.
* Блокировка (Blocked). Поток может перейти в это состояние, в ожидании ресурса, такого как ввод/вывод или из-за блокировки другого объекта. В этом случае поток переходит в состояние Работоспособный, когда ресурс становится доступен.
* Мёртвый (Dead). Поток считается мертвым, когда его метод run() полностью выполнен. Мертвый поток не может перейти ни в какое другое состояние, даже если для него вызван метод start().

11. private mutex.

**Можно ли создавать новые экземпляры класса, пока выполняется static synchronized метод?**

Да, можно создавать новые экземпляры класса, так как статические поля не принадлежат к экземплярам класса.

**Зачем может быть нужен private мьютекс?**

Объект для синхронизации делается private, чтобы сторонний код не мог на него синхронизироваться и случайно получить взаимную блокировку.

**На каком объекте происходит синхронизация при вызове static synchronized метода?**

У синхронизированного статического метода нет доступа к this, но есть доступ к объекту класса Class, он присутствует в единственном экземпляре и именно он выступает в качестве монитора для синхронизации статических методов. Таким образом, следующая конструкция:1

public class SomeClass {

public static synchronized void someMethod() {

//code

}

}

эквивалентна такой:

public class SomeClass {Вызвать метод submit() у экземпляра класса реализующего интерфейс ExecutorService,

public static void someMethod(){

synchronized(SomeClass.class){

//code

}

}

}

12. wait(), notify(), notyfyAll()

**Как работают методы wait() и notify()/notifyAll()?**

Эти методы определены у класса Object и предназначены для взаимодействия потоков между собой при межпоточной синхронизации.

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

Когда вызван метод wait(), поток освобождает блокировку на объекте и переходит из состояния Работающий (Running) в состояние Ожидания (Waiting). Метод notify() подаёт сигнал одному из потоков, ожидающих на объекте, чтобы перейти в состояние Работоспособный (Runnable). При этом невозможно определить, какой из ожидающих потоков должен стать работоспособным. Метод notifyAll() заставляет все ожидающие потоки для объекта вернуться в состояние Работоспособный (Runnable). Если ни один поток не находится в ожидании на методе wait(), то при вызове notify() или notifyAll() ничего не происходит.

Поток может вызвать методы wait() или notify() для определённого объекта, только если он в данный момент имеет блокировку на этот объект. wait(), notify() и notifyAll() должны вызываться только из синхронизированного кода.

**В чем разница между notify() и notifyAll()?**

Дело в том, что «висеть» на методе wait() одного монитора могут сразу несколько потоков. При вызове notify() только один из них выходит из wait() и пытается захватить монитор, а затем продолжает работу со следующего после wait() оператора. Какой из них выйдет - заранее неизвестно. А при вызове notifyAll(), все висящие на wait() потоки выходят из wait(), и все они пытаются захватить монитор. Понятно, что в любой момент времени монитор может быть захвачен только одним потоком, а остальные ждут своей очереди. Порядок очереди определяется планировщиком потоков Java.

**Почему методы wait() и notify() вызываются только в синхронизированном блоке?**

Монитор надо захватывать в явном виде (через synchronized-блок), потому что методы wait() и notify() не синхронизированы.

**Чем отличается работа метода wait() с параметром и без параметра?**

wait() без параметров освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()/notifyAll(),й

с параметрами заставит поток ожидать заданное количество времени или вызова notify()/notifyAll().

13. sleep() vs. yield()

**Чем отличаются методы Thread.sleep() и Thread.yield()?**

Метод yield() служит причиной того, что поток переходит из состояния работающий (running) в состояние работоспособный (runnable), давая возможность другим потокам активизироваться. Но следующий выбранный для запуска поток может и не быть другим.

Метод sleep() вызывает засыпание текущего потока на заданное время, состояние изменяется с работающий (running) на ожидающий (waiting).

**Что значит «усыпить» поток?**

Это значит приостановить его на определенный промежуток времени, вызвав в ходе его выполнения статический метод Thread.sleep() передав в качестве параметра необходимое количество времени в миллисекундах. До истечения этого времени поток может быть выведен из состояния ожидания вызовом interrupt() с выбрасыванием InterruptedException.

14. Thread.join()

**Как работает метод Thread.join()?**

Когда поток вызывает join() для другого потока, текущий работающий поток будет ждать, пока другой поток, к которому он присоединяется, не будет завершён:

void join()

void join(long millis)

void join(long millis, int nanos)

15. сRace condition

**Что такое deadlock?**

Взаимная блокировка (deadlock) - явление при котором все потоки находятся в режиме ожидания. Происходит, когда достигаются состояния:

взаимного исключения: по крайней мере один ресурс занят в режиме неделимости и следовательно только один поток может использовать ресурс в любой данный момент времени.

удержания и ожидания: поток удерживает как минимум один ресурс и запрашивает дополнительные ресурсы, которые удерживаются другими потоками.

отсутствия предочистки: операционная система не переназначает ресурсы: если они уже заняты, они должны отдаваться удерживающим потокам сразу же.

цикличного ожидания: поток ждет освобождения ресурса другим потоком, который в свою очередь ждёт освобождения ресурса, заблокированного первым потоком.

Простейший способ избежать взаимной блокировки – не допускать цикличного ожидания. Этого можно достичь, получая мониторы разделяемых ресурсов в определенном порядке и освобождая их в обратном порядке.

**Что такое livelock?**

livelock – тип взаимной блокировки, при котором несколько потоков выполняют бесполезную работу, попадая в зацикленность при попытке получения каких-либо ресурсов. При этом их состояния постоянно изменяются в зависимости друг от друга. Фактической ошибки не возникает, но КПД системы падает до 0. Часто возникает в результате попыток предотвращения deadlock.

Реальный пример livelock, – когда два человека встречаются в узком коридоре и каждый, пытаясь быть вежливым, отходит в сторону, и так они бесконечно двигаются из стороны в сторону, абсолютно не продвигаясь в нужном им направлении.

**Что такое race condition?**

Состояние гонки (race condition) - ошибка проектирования многопоточной системы или приложения, при которой эта работа напрямую зависит от того, в каком порядке выполняются потоки. Состояние гонки возникает когда поток, который должен исполниться в начале, проиграл гонку и первым исполняется другой поток: поведение кода изменяется, из-за чего возникают недетерменированные ошибки.

**Существует ли способ решения проблемы race condition?**

Распространённые способы решения:

Использование локальной копии — копирование разделяемой переменной в локальную переменную потока. Этот способ работает только тогда, когда переменная одна и копирование производится атомарно (за одну машинную команду), использование volatile.

Синхронизация - операции над разделяемым ресурсом происходят в синхронизированном блоке (при использовании ключевого слова synchronized).

Комбинирование методов - вышеперечисленные способы можно комбинировать, копируя «опасные» переменные в синхронизированном блоке. С одной стороны, это снимает ограничение на атомарность, с другой — позволяет избавиться от слишком больших синхронизированных блоков.

Очевидных способов выявления и исправления состояний гонки не существует. Лучший способ избавиться от гонок — правильное проектирование многозадачной системы.

16. Thread.holdsLock()

**Как проверить, удерживает ли поток монитор определённого ресурса?**

Метод Thread.holdsLock(lock) возвращает true, когда текущий поток удерживает монитор у определённого объекта.

17. volatile.

**Для чего используется ключевое слово volatile, synchronized, transient, native?**

volatile - этот модификатор вынуждает потоки отключить оптимизацию доступа и использовать единственный экземпляр переменной. Если переменная примитивного типа – этого будет достаточно для обеспечения потокобезопасности. Если же переменная является ссылкой на объект – синхронизировано будет исключительно значение этой ссылки. Все же данные, содержащиеся в объекте, синхронизированы не будут!

synchronized - это зарезервированное слово позволяет добиваться синхронизации в помеченных им методах или блоках кода.

Ключевые слова transient и native к многопоточности никакого отношения не имеют, первое используется для указания полей класса, которые не нужно сериализовать, а второе - сигнализирует о том, что метод реализован в платформо-зависимом коде.

18. Atomic переменные.

**В чём различия между volatile и Atomic переменными?**

volatile принуждает использовать единственный экземпляр переменной, но не гарантирует атомарность. Например, операция count++ не станет атомарной просто потому что count объявлена volatile. C другой стороны class AtomicInteger предоставляет атомарный метод для выполнения таких комплексных операций атомарно, например getAndIncrement() – атомарная замена оператора инкремента, его можно использовать, чтобы атомарно увеличить текущее значение на один. Похожим образом сконструированы атомарные версии и для других типов данных.

**В чём заключаются различия между java.util.concurrent.Atomic\*.compareAndSwap() и java.util.concurrent.Atomic\*.weakCompareAndSwap()?**

weakCompareAndSwap() не создает memory barrier и не дает гарантии happens-before;

weakCompareAndSwap() сильно зависит от кэша/CPU, и может возвращать false без видимых причин;

weakCompareAndSwap(), более легкая, но поддерживаемая далеко не всеми архитектурами и не всегда эффективная операция.

19. Приоритет потока (от 1 до 10, default 5).

**Что значит «приоритет потока»?**

Приоритеты потоков используются планировщиком потоков для принятия решений о том, когда какому из потоков будет разрешено работать. Теоретически высокоприоритетные потоки получают больше времени процессора, чем низкоприоритетные. Практически объем времени процессора, который получает поток, часто зависит от нескольких факторов помимо его приоритета.

Чтобы установить приоритет потока, используется метод класса Thread: final void setPriority(int level). Значение level изменяется в пределах от Thread.MIN\_PRIORITY = 1 до Thread.MAX\_PRIORITY = 10. Приоритет по умолчанию - Thread.NORM\_PRlORITY = 5.

Получить текущее значение приоритета потока можно вызвав метод: final int getPriority() у экземпляра класса Thread.

20. Потоки демоны.

**Что такое «потоки-демоны»?**

Потоки-демоны работают в фоновом режиме вместе с программой, но не являются неотъемлемой частью программы. Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения и его деятельность заключается в обслуживании основных потоков приложения, то такой процесс может быть запущен как поток-демон с помощью метода setDaemon(boolean value), вызванного у потока до его запуска. Метод boolean isDaemon() позволяет определить, является ли указанный поток демоном или нет. Базовое свойство потоков-демонов заключается в возможности основного потока приложения завершить выполнение потока-демона (в отличие от обычных потоков) с окончанием кода метода main(), не обращая внимания на то, что поток-демон еще работает.

**Можно ли сделать основной поток программы демоном?**

Нет. Потоки-демоны позволяют описывать фоновые процессы, которые нужны только для обслуживания основных потоков выполнения и не могут существовать без них.

21. Runnable vs. Callable.

**Чем отличаются два интерфейса Runnable и Callable?**

Интерфейс Runnable появился в Java 1.0, а интерфейс Callable был введен в Java 5.0 в составе библиотеки java.util.concurrent;

Классы, реализующие интерфейс Runnable для выполнения задачи должны реализовывать метод run(). Классы, реализующие интерфейс Callable - метод call();

Метод Runnable.run() не возвращает никакого значения, Callable.call() возвращает объект Future, который может содержать результат вычислений;

Метод run() не может выбрасывать проверяемые исключения, в то время как метод call() может.

22. FutureTask.

**Что такое FutureTask?**

FutureTask представляет собой отменяемое асинхронное вычисление в параллельном Java приложении. Этот класс предоставляет базовую реализацию Future, с методами для запуска и остановки вычисления, методами для запроса состояния вычисления и извлечения результатов. Результат может быть получен только когда вычисление завершено, FutureTask метод получения будет заблокирован, если вычисление ещё не завершено. Объекты FutureTask могут быть использованы для обёртки объектов Callable и Runnable. Так как FutureTask реализует Runnable, его можно передать в Executor на выполнение.

23. CyclicBarrier vs. CountDownLatch.

**В чем заключаются различия между CyclicBarrier и CountDownLatch?**

CountDownLatch (замок с обратным отсчетом) предоставляет возможность любому количеству потоков в блоке кода ожидать до тех пор, пока не завершится определенное количество операций, выполняющихся в других потоках, перед тем как они будут «отпущены», чтобы продолжить свою деятельность. В конструктор CountDownLatch(int count) обязательно передается количество операций, которое должно быть выполнено, чтобы замок «отпустил» заблокированные потоки.

Примером CountDownLatch из жизни может служить сбор экскурсионной группы: пока не наберется определенное количество человек, экскурсия не начнется.

CyclicBarrier реализует шаблон синхронизации «Барьер». Циклический барьер является точкой синхронизации, в которой указанное количество параллельных потоков встречается и блокируется. Как только все потоки прибыли, выполняется опционное действие (или не выполняется, если барьер был инициализирован без него), и, после того, как оно выполнено, барьер ломается и ожидающие потоки «освобождаются». В конструкторы барьера CyclicBarrier(int parties) и CyclicBarrier(int parties, Runnable barrierAction) обязательно передается количество сторон, которые должны «встретиться», и, опционально, действие, которое должно произойти, когда стороны встретились, но перед тем когда они будут «отпущены».

CyclicBarrier является альтернативой метода join(), который «собирает» потоки только после того, как они выполнились.

CyclicBarrier похож на CountDownLatch, но главное различие между ними в том, что использовать «замок» можно лишь единожды - после того, как его счётчик достигнет нуля, а «барьер» можно использовать неоднократно, даже после того, как он «сломается».

24. Thread.stop() vs interrupt(). interrupted() vs. isInterrupted().

**Как остановить поток?**

На данный момент в Java принят уведомительный порядок остановки потока (хотя JDK 1.0 и имеет несколько управляющих выполнением потока методов, например stop(), suspend() и resume() - в следующих версиях JDK все они были помечены как deprecated из-за потенциальных угроз взаимной блокировки).

Для корректной остановки потока можно использовать метод класса Thread - interrupt(). Этот метод выставляет некоторый внутренний флаг-статус прерывания. В дальнейшем состояние этого флага можно проверить с помощью метода isInterrupted() или Thread.interrupted() (для текущего потока). Метод interrupt() также способен вывести поток из состояния ожидания или спячки. Т.е. если у потока были вызваны методы sleep() или wait() – текущее состояние прервется и будет выброшено исключение InterruptedException. Флаг в этом случае не выставляется.

Схема действия при этом получается следующей:

Реализовать поток.

В потоке периодически проводить проверку статуса прерывания через вызов isInterrupted().

Если состояние флага изменилось или было выброшено исключение во время ожидания/спячки, следовательно поток пытаются остановить извне.

Принять решение – продолжить работу (если по каким-то причинам остановиться невозможно) или освободить заблокированные потоком ресурсы и закончить выполнение.

Возможная проблема, которая присутствует в этом подходе – блокировки на потоковом вводе-выводе. Если поток заблокирован на чтении данных - вызов interrupt() из этого состояния его не выведет. Решения тут различаются в зависимости от типа источника данных. Если чтение идет из файла – долговременная блокировка крайне маловероятна и тогда можно просто дождаться выхода из метода read(). Если же чтение каким-то образом связано с сетью – стоит использовать неблокирующий ввод-вывод из Java NIO.

Второй вариант реализации метода остановки (а также и приостановки) – сделать собственный аналог interrupt(). Т.е. объявить в классе потока флаги – на остановку и/или приостановку и выставлять их путем вызова заранее определённых методов извне. Методика действия при этом остаётся прежней – проверять установку флагов и принимать решения при их изменении. Недостатки такого подхода. Во-первых, потоки в состоянии ожидания таким способом не «оживить». Во-вторых, выставление флага одним потоком совсем не означает, что второй поток тут же его увидит. Для увеличения производительности виртуальная машина использует кеш данных потока, в результате чего обновление переменной у второго потока может произойти через неопределенный промежуток времени (хотя допустимым решением будет объявить переменную-флаг как volatile).

**Почему не рекомендуется использовать метод Thread.stop()?**

При принудительной остановке (приостановке) потока, stop() прерывает поток в недетерменированном месте выполнения, в результате становится совершенно непонятно, что делать с принадлежащими ему ресурсами. Поток может открыть сетевое соединение - что в таком случае делать с данными, которые еще не вычитаны? Где гарантия, что после дальнейшего запуска потока (в случае приостановки) он сможет их дочитать? Если поток блокировал разделяемый ресурс, то как снять эту блокировку и не переведёт ли принудительное снятие к нарушению консистентности системы? То же самое можно расширить и на случай соединения с базой данных: если поток остановят посередине транзакции, то кто ее будет закрывать? Кто и как будет разблокировать ресурсы?

**В чем разница между interrupted() и isInterrupted()?**

Механизм прерывания работы потока в Java реализован с использованием внутреннего флага, известного как статус прерывания. Прерывание потока вызовом Thread.interrupt() устанавливает этот флаг. Методы Thread.interrupted() и isInterrupted() позволяют проверить, является ли поток прерванным.

Когда прерванный поток проверяет статус прерывания, вызывая статический метод Thread.interrupted(), статус прерывания сбрасывается.

Нестатический метод isInterrupted() используется одним потоком для проверки статуса прерывания у другого потока, не изменяя флаг прерывания.

25. Исключение в потоке.

**Что происходит, когда в потоке выбрасывается исключение?**

Если исключение не поймано – поток «умирает» (переходит в состяние мёртв (dead)).

Если установлен обработчик непойманных исключений, то он возьмёт управление на себя. Thread.UncaughtExceptionHandler – интерфейс, определенный как вложенный интерфейс для других обработчиков, вызываемых, когда поток внезапно останавливается из-за непойманного исключения. В случае, если поток собирается остановиться из-за непойманного исключения, JVM проверяет его на наличие UncaughtExceptionHandler, используя Thread.getUncaughtExceptionHandler(), и если такой обработчик найдет, то вызовет у него метод uncaughtException(), передав этот поток и исключение в виде аргументов.

26. Пулл потоков.

**Что такое «пул потоков»?**

Создание потока является затратной по времени и ресурсам операцией. Количество потоков, которое может быть запущено в рамках одного процесса также ограниченно. Чтобы избежать этих проблем и в целом управлять множеством потоков более эффективно в Java был реализован механизм пула потоков (thread pool), который создаётся во время запуска приложения и в дальнейшем потоки для обработки запросов берутся и переиспользуются уже из него. Таким образом, появляется возможность не терять потоки, сбалансировать приложение по количеству потоков и частоте их создания.

**Какого размера должен быть пул потоков?**

Настраивая размер пула потоков, важно избежать двух ошибок: слишком мало потоков (очередь на выполнение будет расти, потребляя много памяти) или слишком много потоков (замедление работы всей систему из-за частых переключений контекста).

Оптимальный размер пула потоков зависит от количества доступных процессоров и природы задач в рабочей очереди. На N-процессорной системе для рабочей очереди, которая будет выполнять исключительно задачи с ограничением по скорости вычислений, можно достигнуть максимального использования CPU с пулом потоков, в котором содержится N или N+1 поток. Для задач, которые могут ждать осуществления I/O (ввода - вывода) - например, задачи, считывающей HTTP-запрос из сокета – может понадобиться увеличение размера пула свыше количества доступных процессоров, потому, что не все потоки будут работать все время. Используя профилирование, можно оценить отношение времени ожидания (WT) ко времени обработки (ST) для типичного запроса. Если назвать это соотношение WT/ST, то для N-процессорной системе понадобится примерно N\*(1 + WT/ST) потоков для полной загруженности процессоров.

Использование процессора – не единственный фактор, важный при настройке размера пула потоков. По мере возрастания пула потоков, можно столкнуться с ограничениями планировщика, доступной памяти, или других системных ресурсов, таких, как количество сокетов, дескрипторы открытого файла, или каналы связи базы данных.

**Что будет, если очередь пула потоков уже заполнена, но подается новая задача?**

Если очередь пула потоков заполнилась, то поданная задача будет «отклонена». Например - метод submit() у ThreadPoolExecutor выкидывает RejectedExecutionException, после которого вызывается RejectedExecutionHandler.

**В чём заключается различие между методами submit() и execute() у пула потоков?**

Оба метода являются способами подачи задачи в пул потоков, но между ними есть небольшая разница.

execute(Runnable command) определён в интерфейсе Executor и выполняет поданную задачу и ничего не возвращает.

submit() – перегруженный метод, определённый в интерфейсе ExecutorService. Способен принимать задачи типов Runnable и Callable и возвращать объект Future, который можно использовать для контроля и управления процессом выполнения, получения его результата.

27. Executor, Executors.

Начиная с Java 1.5 Java API предоставляет фреймворк Executor, который позволяет создавать различные типы пула потоков:

Executor - упрощенный интерфейс пула, содержит один метод для передачи задачи на выполнение;

ExecutorService - расширенный интерфейс пула, с возможностью завершения всех потоков;

AbstractExecutorService - базовый класс пула, реализующий интерфейс ExecutorService;

Executors - фабрика объектов связанных с пулом потоков, в том числе позволяет создать основные типы пулов;

ThreadPoolExecutor - пул потоков с гибкой настройкой, может служить базовым классом для нестандартных пулов;

ForkJoinPool - пул для выполнения задач типа ForkJoinTask;

... и другие.

Методы Executors для создания пулов:

newCachedThreadPool() - если есть свободный поток, то задача выполняется в нем, иначе добавляется новый поток в пул. Потоки не используемые больше минуты завершаются и удаляются из кэша. Размер пула неограничен. Предназначен для выполнения множество небольших асинхронных задач;

newCachedThreadPool(ThreadFactory threadFactory) - аналогично предыдущему, но с собственной фабрикой потоков;

newFixedThreadPool(int nThreads) - создает пул на указанное число потоков. Если новые задачи добавлены, когда все потоки активны, то они будут сохранены в очереди для выполнения позже. Если один из потоков завершился из-за ошибки, на его место будет запущен другой поток. Потоки живут до тех пор, пока пул не будет закрыт явно методом shutdown().

newFixedThreadPool(int nThreads, ThreadFactory threadFactory) - аналогично предыдущему, но с собственной фабрикой потоков;

newSingleThreadScheduledExecutor() - однопотоковый пул с возможностью выполнять задачу через указанное время или выполнять периодически. Если поток был завершен из-за каких-либо ошибок, то для выполнения следующей задачи будет создан новый поток.

newSingleThreadScheduledExecutor(ThreadFactory threadFactory) - аналогично предыдущему, но с собственной фабрикой потоков;

newScheduledThreadPool(int corePoolSize) - пул для выполнения задач через указанное время или переодически;

newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory) - аналогично предыдущему, но с собственной фабрикой потоков;

unconfigurableExecutorService(ExecutorService executor) - обертка на пул, запрещающая изменять его конфигурацию;

28. Fork/Join framework.

**Что такое «фреймворк Fork/Join»?**

Фреймворк Fork/Join, представленный в JDK 7, - это набор классов и интерфейсов позволяющих использовать преимущества многопроцессорной архитектуры современных компьютеров. Он разработан для выполнения задач, которые можно рекурсивно разбить на маленькие подзадачи, которые можно решать параллельно.

Этап Fork: большая задача разделяется на несколько меньших подзадач, которые в свою очередь также разбиваются на меньшие. И так до тех пор, пока задача не становится тривиальной и решаемой последовательным способом.

Этап Join: далее (опционально) идёт процесс «свёртки» - решения подзадач некоторым образом объединяются пока не получится решение всей задачи.

Решение всех подзадач (в т.ч. и само разбиение на подзадачи) происходит параллельно.

Для решения некоторых задач этап Join не требуется. Например, для параллельного QuickSort — массив рекурсивно делится на всё меньшие и меньшие диапазоны, пока не вырождается в тривиальный случай из 1 элемента. Хотя в некотором смысле Join будет необходим и тут, т.к. всё равно остаётся необходимость дождаться пока не закончится выполнение всех подзадач.

Ещё одно замечательное преимущество этого фреймворка заключается в том, что он использует work-stealing алгоритм: потоки, которые завершили выполнение собственных подзадач, могут «украсть» подзадачи у других потоков, которые всё ещё заняты.

29. Semaphore.

**Что такое Semaphore?**

Semaphore – это новый тип синхронизатора: семафор со счётчиком, реализующий шаблон синхронизации Семафор. Доступ управляется с помощью счётчика: изначальное значение счетчика задается в конструкторе при создании синхронизатора, когда поток заходит в заданный блок кода, то значение счетчика уменьшается на единицу, когда поток его покидает, то увеличивается. Если значение счетчика равно нулю, то текущий поток блокируется, пока кто-нибудь не выйдет из защищаемого блока. Semaphore используется для защиты дорогих ресурсов, которые доступны в ограниченном количестве, например подключение к базе данных в пуле.

30. ThreadLocal-переменная.

**Что такое ThreadLocal-переменная?**

ThreadLocal - класс, позволяющий имея одну переменную, иметь различное её значение для каждого из потоков.

У каждого потока - т.е. экземпляра класса Thread - есть ассоциированная с ним таблица ThreadLocal-переменных. Ключами таблицы являются cсылки на объекты класса ThreadLocal, а значениями - ссылки на объекты, «захваченные» ThreadLocal-переменными, т.е. ThreadLocal-переменные отличаются от обычных переменных тем, что у каждого потока свой собственный, индивидуально инициализируемый экземпляр переменной. Доступ к значению можно получить через методы get() или set().

Например, если мы объявим ThreadLocal-переменную: ThreadLocal<Object> locals = new ThreadLocal<Object>();. А затем, в потоке, сделаем locals.set(myObject), то ключом таблицы будет ссылка на объект locals, а значением - ссылка на объект myObject. При этом для другого потока существует возможность «положить» внутрь locals другое значение.

Следует обратить внимание, что ThreadLocal изолирует именно ссылки на объекты, а не сами объекты. Если изолированные внутри потоков ссылки ведут на один и тот же объект, то возможны коллизии.

Так же важно отметить, что т.к. ThreadLocal-переменные изолированы в потоках, то инициализация такой переменной должна происходить в том же потоке, в котором она будет использоваться. Ошибкой является инициализация такой переменной (вызов метода set()) в главном потоке приложения, потому как в данном случае значение, переданное в методе set(), будет «захвачено» для главного потока, и при вызове метода get() в целевом потоке будет возвращен null.

31. synchronized vs. ReentrantLock.

**Назовите различия между synchronized и ReentrantLock?**

В Java 5 появился интерфейс Lock предоставляющий возможности более эффективного и тонкого контроля блокировки ресурсов. ReentrantLock – распространённая реализация Lock, которая предоставляет Lock с таким же базовым поведением и семантикой, как у synchronized, но расширенными возможностями, такими как опрос о блокировании (lock polling), ожидание блокирования заданной длительности и прерываемое ожидание блокировки. Кроме того, он предлагает гораздо более высокую эффективность функционирования в условиях жесткой состязательности.

Что понимается под блокировкой с повторным входом (reentrant)? Просто то, что есть подсчет сбора данных, связанный с блокировкой, и если поток, который удерживает блокировку, снова ее получает, данные отражают увеличение, и тогда для реального разблокирования нужно два раза снять блокировку. Это аналогично семантике synchronized; если поток входит в синхронный блок, защищенный монитором, который уже принадлежит потоку, потоку будет разрешено дальнейшее функционирование, и блокировка не будет снята, когда поток выйдет из второго (или последующего) блока synchronized, она будет снята только когда он выйдет из первого блока synchronized, в который он вошел под защитой монитора.

Lock lock = new ReentrantLock();

lock.lock();

try {

// update object state

}

finally {

lock.unlock();

}

Реализация ReentrantLock гораздо более масштабируема в условиях состязательности, чем реализация synchronized. Это значит, что когда много потоков соперничают за право получения блокировки, общая пропускная способность обычно лучше у ReentrantLock, чем у synchronized. JVM требуется меньше времени на установление очередности потоков и больше времени на непосредственно выполнение.

У ReentrantLock (как и у других реализаций Lock) блокировка должна обязательно сниматься в finally блоке (иначе, если бы защищенный код выбросил исключение, блокировка не была бы снята). Используя синхронизацию, JVM гарантирует, что блокировка автоматически снимаются.

Резюмируя можно сказать, что когда состязания за блокировку нет либо оно очень мало, то synchronized возможно будет быстрее. Если присутствует заметное состязание за доступ к ресурсу, то скорее всего ReentrantLock даст некое преимущество.

32. ReedWriteLock.

**Что такое ReadWriteLock?**

ReadWriteLock – это интерфейс расширяющий базовый интерфейс Lock. Используется для улучшения производительности в многопоточном процессе и оперирует парой связанных блокировок (одна - для операций чтения, другая - для записи). Блокировка чтения может удерживаться одновременно несколькими читающими потоками, до тех пор пока не появится записывающий. Блокировка записи является эксклюзивной.

Существует реализующий интерфейс ReadWriteLock класс ReentrantReadWriteLock, который поддерживает до 65535 блокировок записи и до стольки же блокировок чтения.

ReadWriteLock rwLock = new ReentrantReadWriteLock();

Lock rLock = rwLock.readLock();

Lock wLock = rwLock.writeLock();

wLock.lock();

try {

// exclusive write

} finally {

wLock.unlock();

}

rLock.lock();

try {

// shared reading

} finally {

rLock.unlock();

}

33. Блокирующий метод.

**Что такое «блокирующий метод»?**

Блокирующий метод – метод, который блокируется, до тех пор, пока задание не выполнится, например метод accept() у ServerSocket блокируется в ожидании подключения клиента. Здесь блокирование означает, что контроль не вернётся к вызывающему методу до тех пор, пока не выполнится задание. Так же существуют асинхронные или неблокирующиеся методы, которые могут завершится до выполнения задачи.

34. Busy spin.

**Что такое busy spin?**

busy spin – это техника, которую программисты используют, чтобы заставить поток ожидать при определённом условии. В отличие от традиционных методов wait(), sleep() или yield(), которые подразумевают уступку процессорного времени, этот метод вместо уступки выполняет пустой цикл. Это необходимо, для того, чтобы сохранить кэш процессора, т.к. в многоядерных системах, существует вероятность, что приостановленный поток продолжит свое выполнение уже на другом ядре, а это повлечет за собой перестройку состояния процессорного кэша, которая является достаточно затратной процедурой.