Вопросы для подготовки:

- Что вы знаете о модели памяти Java?

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

В Java существуют различные концепции и термины, связанные с параллельным выполнением кода и обеспечением корректности работы программы. Вот объяснения некоторых из них:

Ordering (упорядочивание): Управление порядком выполнения операций в многопоточной среде или при работе с гарантированно упорядоченными структурами данных.

As-if-serial semantics (семантика "как если бы это выполнялось последовательно"): Это принцип, согласно которому результат выполнения программы должен быть таким же, как если бы все операции выполнялись последовательно, даже если фактически происходит параллельное выполнение.

Sequential consistency (последовательная согласованность): Гарантирует, что все потоки видят один и тот же порядок операций, как если бы они выполнялись последовательно в одном потоке.

Visibility (видимость): Обеспечивает, что изменения, сделанные одним потоком в разделяемых переменных, будут видны другим потокам. Без правильного обеспечения видимости возможны ошибки синхронизации и непредсказуемые результаты.

Atomicity (атомарность): Гарантирует, что операция выполняется как неделимая единица и не может быть прервана или разделена на части. Атомарные операции обеспечивают согласованность данных в многопоточной среде.

Happens-before (происходит-до): Устанавливает отношение порядка между операциями в коде. Если операция A происходит-до операции B, то B видит все изменения, внесенные A.

Mutual exclusion (взаимное исключение): Механизм, позволяющий гарантировать, что только один поток может выполнять критическую секцию кода в определенный момент времени. Это обеспечивает консистентное состояние при доступе к разделяемым ресурсам.

Safe publication (безопасная публикация): Методика обеспечения корректной и безопасной видимости объектов в многопоточной среде. Безопасная публикация гарантирует, что другие потоки будут видеть правильное и полностью инициализированное состояние объекта.

Эти концепции и термины являются основными для понимания и управления параллельным выполнением кода в Java и помогают гарантировать правильность и надежность программ.

- В чём разница между конкуренцией и параллелизмом?

В контексте многопоточности в Java, конкуренция (concurrency) и параллелизм (parallelism) являются двумя разными концепциями, связанными с одновременным выполнением задач. Вот их определения и различия:

Конкуренция (Concurrency): Конкуренция означает, что несколько задач выполняются одновременно, но не обязательно одновременно на физическом уровне (на разных процессорах или ядрах). Задачи могут быть переключены между собой, чтобы дать иллюзию одновременного выполнения. В многопоточном приложении с конкуренцией потоки могут исполняться параллельно, если доступны ресурсы процессора, но также могут и переключаться по времени.

Параллелизм (Parallelism): Параллелизм означает фактическое одновременное выполнение нескольких задач на разных физических ресурсах, таких как множество процессоров или ядер в многоядерной системе. При использовании параллелизма, задачи действительно выполняются одновременно и могут значительно увеличить производительность приложения.

Основное отличие между конкуренцией и параллелизмом заключается в том, что конкуренция описывает способность системы обрабатывать множество задач одновременно, независимо от физического параллелизма, в то время как параллелизм предполагает реальное одновременное выполнение задач на разных физических ресурсах.

- 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.

- Каким образом можно создать поток?

Для создания нового потока мы можем создать новый класс, либо наследуя его от класса Thread, либо реализуя в классе интерфейс Runnable.

- 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.

- 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.

- 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.

- В каких состояниях может находиться поток?

Поток может находиться в одном из следующих состояний:

New - объект класса Thread создан, но еще не запущен. Он еще не является потоком выполнения и естественно не выполняется.

Runnable - поток готов к выполнению, но планировщик еще не выбрал его.

Running – поток выполняется.

Waiting/blocked/sleeping - поток блокирован или поток ждет окончания работы другого потока.

Dead - поток завершен. Будет выброшено исключение при попытке вызвать метод start() для dead потока.

Существуют перечисление Thread.State, содержащее значения возможных состояний потока: NEW, RUNNABLE, BLOCKED, WAITING, TIMED\_WAITING, TERMINATED.

Получить текущее значение состояния потока можно вызовом метода getState() класса Thread.

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

wait(): освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()

notify(): продолжает работу потока, у которого ранее был вызван метод wait()

notifyAll(): возобновляет работу всех потоков, у которых ранее был вызван метод wait()

- 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.

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

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

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

- 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).

- 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)

- 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 ресурсов, которые удерживаются другими потоками.

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

цикличного ожидания: поток ждёт 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 определённом 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.

- Для чего используется ключевое слово volatile?

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!

- 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.

- 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.

- Чем отличаются два интерфейса Runnable и Callable?

Интерфейс Runnable появиля 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.

- 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, 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.

- 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».

- 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 исполнится в начале, проиграл гонку и первым исполняется другой поток: поведение кода изменяется, из-за чего возникают недетерменированные ошибки.

- Какие есть способы 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

- Как остановить поток?

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.

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.

- 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?

- 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, определённый 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.

- Что такое Executor?

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.

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:

- Какие типы Executor'ов вы знаете?

- В чём заключается различие между методами submit() и execute() у Executor?

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.

- 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.

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

-Xss

- 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().

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.

- 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.

- 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 эксклюзивеной.

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

- Что вы знаете о 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.

- Что такое Semaphore?

Semaphore - это новый тип синхронизатора: семафор со счётчиком, реализующий шаблон синхронизации Семафор. Доступ управляется с помощью счётчика: изначальное значение счётчика задаётся в конструкторе при создании синхронизатора, когда поток заходит в заданный блок кода, то значение счётчика уменьшается на единицу, когда поток его покидает, то увеличивается. Если значение счётчика 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.

- Как создать потокобезопасный Singleton?

Static field

public class Singleton {

public static final Singleton INSTANCE = new Singleton();

}

Enum

public enum Singleton {

INSTANCE;

}

Synchronized Accessor

public class Singleton {

private static Singleton instance;

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

Double Checked Locking & volatile

public class Singleton {

private static volatile Singleton instance;

public static Singleton getInstance() {

Singleton localInstance = instance;

if (localInstance == null) {

synchronized (Singleton.class) {

localInstance = instance;

if (localInstance == null) {

instance = localInstance = new Singleton();

}

}

}

return localInstance;

}

}

On Demand Holder Idiom

public class Singleton {

public static class SingletonHolder {

public static final Singleton HOLDER\_INSTANCE = new Singleton();

}

public static Singleton getInstance() {

return SingletonHolder.HOLDER\_INSTANCE;

}

}