INTERVIEW JAVA DEVELOPER MULTITHREADING

Ответы на вопросы на собеседование Multithreading (часть 1).

Vasyl K 7:23:00 1 Комментарий

- **ЧЕМ ОТЛИЧАЕТСЯ ПРОЦЕСС ОТ ПОТОКА?**

Процесс это некоторая единица операционной системы, которой выделена память и другие ресурсы. Поток это единица исполнения кода. Поток имеет стэк – некоторую свою память для исполнения. Остальная память процесса – общая для всех его потоков. Потоки исполняются на ядрах процессора.
В некоторых OS разница между процессами и потоками сведена к минимуму.

- **КАКИМ ОБРАЗОМ МОЖНО СОЗДАТЬ ПОТОК?**

Есть несколько способов создания и запуска потоков:

- С помощью класса, реализующего Runnable:
  - Создать объект класса Thread.
  - Создать объект класса, реализующего интерфейс Runnable.
  - Вызвать у созданного объекта Thread метод start() (после этого запустится метод run() у переданного объекта, реализующего Runnable).
- С помощью класса, расширяющего Thread:
  - Создать объект класса ClassName extends Thread.
  - Переопределить run() в этом классе (смотрите примере ниже, где передается имя потока 'Second').
- С помощью класса, реализующего java.util.concurrent.Callable:
  - Создать объект класса, реализующего интерфейс Callable.
  - Создать объект ExecutorService с указанием пула потоков.
  - Создать объект Future. Запуск происходит через метод submit(); Сигнатура: <T> Future<T> submit(Callable<T> task).

- **ЧТО ТАКОЕ МОНИТОР?**

Контроль за доступом к объекту-ресурсу обеспечивает понятие монитора. Монитор экземпляра может иметь только одного владельца. При попытке конкурирующего доступа к объекту, чей монитор имеет владельца, желающий заблокировать объект-ресурс поток должен подождать освобождения монитора этого объекта и только после этого завладеть им и начать использование объекта-ресурса.

• КАКИЕ СПОСОБЫ СИНХРОНИЗАЦИИ В JAVA?

Ниже приведены некоторые 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.
- Системная синхронизация с использованием join. Метод join, вызванный у экземпляра класса Thread, позволяет текущему потоку остановиться до того момента, как поток, связаный с этим экземпляром, закончит работу.
- Использование классов из пакета java.util.concurrent, который предоставляет набор классов для организации межпоточного взаимодействия. Примеры таких классов - Lock, семафор (Semaphore), etc. Концепция данного подхода заключается в использовании атомарных операций и переменных.

• КАК РАБОТАЮТ МЕТОДЫ WAIT И NOTIFY/NOTIFYALL?

Эти метеоды предназначены для межпоточной синхронизации, для взаимодействия потоков между собой.

Как работают эти методы. Во-первых они могут вызваны только потоком, который захватил монитор объекта, для которого эти методы вызываются. То есть они вызываются внутри блока synchronized и для объекта, монитор которого этим synchronized захвачен. Если внутри synchronized метода - то для класса, к которому относятся эти методы.

Что делает метод wait(). Метод wait() отдает (освобождает) монитор объекта, так что другие потоки теперь могут его (монитор) захватить, то есть войти в блок synchronized для этого объекта. Затем метод wait() переходит в состояние ожидания, до тех пор пока другой поток не вызывет метод notify() или notifyAll() для этого же объекта. После чего поток, в котором был вызван wait(), пытается снова захватить монитор объекта и когда монитор становится свободным, то есть когда другой поток освобождает его, захватывает монитор и продолжает выполнение со следующего после wait() оператора. Причем у потока вызвашего wait() нет никакого преимущества перед другими потоками, ожидающими захвата того же монитора.

Что делают методы notify(), notifyAll(). Они "пробуждают" поток, ожидающий

методом wait() (если такой есть), и переводят его в состояние ожидания освобождения монитора. Разница между notify() и notifyAll() в том, что notify() пробуждает только один поток , ожидающий методом wait(), какой именно будет пробужден - определить нельзя, а notifyAll() - все такие потоки.

- **ЧЕМ ОТЛИЧАЕТСЯ РАБОТА МЕТОДА WAIT С ПАРАМЕТРОМ И БЕЗ ПАРАМЕТРА?**

Разница методов в следующем:

- final void wait() - метод используется в многопоточной среде, может вызываться только потоком, владеющим объектом синхронизации. При этом объект синхронизации освобождается, а текущий поток переходит в режим ожидания сигнала освобождения объекта синхронизации другим потоком путем вызова метода notify() либо notifyAll().
- final void wait(long time) - аналогично wait() данный метод используется в многопоточной среде, переходит текущий поток в режим ожидания сигнала освобождения объекта синхронизации другим потоком путем вызова метода notify() либо notifyAll(), или ожидание происходит заданное время time, затем выполнение продолжается безусловно.

- **КАК РАБОТАЕТ МЕТОД THREAD.YIELD()? ЧЕМ ОТЛИЧАЮТСЯ МЕТОДЫ THREAD.SLEEP() И THREAD.YIELD()?**

Основные отличия:

- метод yield() - пытается сказать планировщику потоков, что нужно выполнить другой поток, что ожидает в очереди на выполнение. Метод не пытается перевести текущий поток в состояние блокировки, сна или ожидания. Он просто пытается его перевести из состояние "работающий" в состояние "работоспособный". Однако выполнение метода может вообще не произвести никакого эффекта. состояние потока остатается RUNNABLE
- метод sleep() - приостанавливает поток на указанное. состояние меняется на TIMED_WAITING, по истечению - RUNNABLE
- метод wait() - меняет состояние потока на WAITING может быть вызвано только у объекта владеющего блокировкой, в противном случае выкинется исключение IllegalMonitorStateException. при срабатывании метода блокировка отпускается, что позволяет продолжить работу другим потокам ожидающим захватить ту же самую блокировку . в случае wait(int) с аргументом состояние будет TIMED_WAITING.

- **КАК РАБОТАЕТ МЕТОД THREAD.JOIN()?**

Метод join() вызывается для того, чтобы привязать текущий поток в конец потока для которого вызывается метод. То есть второй поток будет в режиме блокировки пока первый поток не выполнится.

- **ЧТО ТАКОЕ DEAD LOCK?**

Это когда один поток A получил блокировку на объект A1, а поток B получил блокировку на объект B1. В то время как поток A пытается получит блокировку на объект B1, а поток B на A1.

- **НА КАКОМ ОБЪЕКТЕ ПРОИСХОДИТ СИНХРОНИЗАЦИЯ ПРИ ВЫЗОВЕ STATIC SYNCHRONIZED МЕТОДА?**

Представьте себе ситуацию что два потока одновременно изменяют состояние какого-то объекта, это недопустимо. Для этого необходимо синхронизировать потоки. Как это сделать? Ключевое слово synchronized позволяет это сделать установив в сигнатуре метода. Или же в методе можно описать блок synchronized, только в качестве параметра необходимо передать объект, который будет блокироватся.
Представьте себе ситуацию когда один поток ждет пока разблокируется объект… а если это ждут несколько потоков? Нет гарантии что тот объект что больше всех ждал снятия блокировки будет выполнятся первым.
Статические синхронизированные методы и нестатические синхронизированные методы не будет блокировать друг друга, никогда. Статические методы блокируются на экземпляре класса Class в то время как нестатические методы блокируются на текущем экземпляре (this). Эти действия не мешают друг другу.
wait() - отказывается от блокировки остальные методы сохраняют блокировку.

- **ДЛЯ ЧЕГО ИСПОЛЬЗУЕТСЯ КЛЮЧЕВОЕ СЛОВО VOLATILE, SYNCHRONIZED, TRANSIENT, NATIVE?**

Краткое описание ключевых слов:
- volatile - указывает на то, что поле синхронизировано для нескольких потоков
- synchronized - указывает на то что метод синхронизированный или же в методе может находится такой блок синхронизации.
- transient - указывает на то, что переменная не подлежит сериализации
- native - говорит о том, что реализация метода написана на другой программной платформе

- **ЧТО ЗНАЧИТ ПРИОРИТЕТ ПОТОКА?**

Приоритет потока - это число от 1 до 10, в зависимости от которого, планировщик потоков выбирает какой поток запускать. Однако полагаться на приоритеты для

предсказуемого выполнения многопоточной программы нельзя!

- **ЧТО ТАКОЕ ПОТОКИ – ДЕМОНЫ В ДЖАВА?**

Это потоки, которое работают в фоновом режиме и не гарантируют что они завершатся. Тоеть если все потоки завершились, то поток демон просто обрывается вместе с закрытием приложения.

- **ЧТО ЗНАЧИТ УСЫПИТЬ ПОТОК?**

Перевести поток в спящее состояние можно с помощью метода sleep(long ms) ms – время в миллисекундах.
При вызове этого метода, поток переходит в спящее состояние, после сна, поток переходит в пул потоков и находится в состоянии "работоспособный", т.е. не гарантируется что после пробуждения он будет сразу выполняться. Также поток не может усыпить другой поток, так как метод sleep – это статический метод! Вы просто усыпите текущий поток и не более того! Также метод sleep() может возбуждать InterruptedException().

- **В КАКИХ СОСТОЯНИЯХ МОЖЕТ БЫТЬ ПОТОК В ДЖАВА? КАК ВООБЩЕ РАБОТАЕТ ПОТОК?**

У нас есть текущий поток, в котором выполняется метод main. Этот поток имеет свой стек и этот стек начинается с вызова метода main.
Далее в методе main мы создаем новый поток, что происходит… создается новый поток и для него выделяется свой стек с первоначальным методом run().
Когда мы запускаем несколько потоков, то мы не можем гарантировать определенный порядок их вызовов. Планированием потоков занимается планировщик потоков JVM, выбирая из пулов потоков поток. Мы даже не можем гарантировать что если первый поток начался выполнятся первым, то он и закончит выполнятся первым, он может закончить выполнятся последним.
Еще такой ньюанс, что поток, который закончил свое выполнение, не может быть повторно запущен! Он находится в состоянии "мертвый", а для запуска потока нового потока, объект должен находится в состоянии "новый".
Потоки имеют такие состояния:

- новый(это когда только создали экземпляр класса Thread)
- живой или работоспособный(переходит в это состояние после запуска метода start(), но это не означает что поток уже работает! Или же он может перейти в это состояние из состояние работающий или блокированный)
- работающий(это когда метод run() начал выполнятся)

- ожидающий (waiting)/Заблокированный (blocked)/Спящий(sleeping). Эти состояния характеризуют поток как не готовый к работе. Я объединил эти состояния т.к. все они имеют общую черту – поток еще жив (alive), но в настоящее время не может быть выполнен. Другими словами поток уже не работает, но он может вернутся в рабочее состояние. Поток может быть заблокирован, это может означать что он ждет освобождение каких-то ресурсов. Поток может спать, если встретился метод sleep(long s) , или же он может ожидать, если встретился метод wait(), он будет ждать пока не вызовится метод notify() или notifyall().
- мертвый(состояние когда метод run() завершил свою работу)

• ЧЕМ ОТЛИЧАЮТСЯ ДВА ИНТЕРФЕЙСА ДЛЯ РЕАЛИЗАЦИИ ЗАДАЧ RUNNABLE И CALLABLE?

Основные различия:

- 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 должны реализовывать метод run() для выполнения задачи. Классы, реализующие интерфейс Callable должны реализовывать метод call() для выполнения задачи.
- Метод Runnable.run() не возвращает никакого значения, его тип void, а метод Callable.call() может возвращать значение типа T. Интерфейс Callable является параметризированным Callable<T> и тип значения, которое будет возвращаться в методе call() задается этим параметром T.
- Метод run() не может бросить проверяемое исключение, в то время как метод call() может бросить проверяемое исключение.

• РАЗЛИЧИЯ МЕЖДУ CYCLICBARRIER И COUNTDOWNLATCH?

Хоть оба эти синхронизаторы позволяют нитям дожидаться друг друга, главное различие между ними в том, что вы не можете заново использовать CountDownLatch после того, как его счётчик достигнет нуля, но вы можете использовать CyclicBarrier снова, даже после того, как барьер сломается.

• ЧТО ТАКОЕ СОСТОЯНИЕ ГОНКИ (RACE CONDITION)?

Состояние гонки – причина трудноуловимых багов. Как сказано в самом названии, состояние гонки возникает из-за гонки между несколькими нитями, если нить, которая должна исполнятся первой, проиграла гонку и исполняется вторая, поведение кода изменяется, из-за чего возникают недетерменированные баги. Это одни из сложнейших к отлавливанию и воспроизведению багов, из-за беспорядочной природы гонок между нитями. Пример состояния гонки – беспорядочное исполнение.

- **КАК ОСТАНОВИТЬ НИТЬ?**

Java предоставляет богатые API для всего, но, по иронии судьбы, не предоставляет удобных способов остановки нити. В JDK 1.0 было несколько управляющих методов, например stop(), suspend() и resume(), которые были помечены как deprecated в будущих релизах из-за потенциальных угроз взаимной блокировки, с тех пор разработчики Java API не предприняли попыток представить стойкий, нитe-безопасный и элегантный способ остановки нитей. Программисты в основном полагаются на факт того, что нить останавливается сама, как только заканчивает выполнять методы run() или call(). Для остановки вручную, программисты пользуются преимуществом volatile boolean переменной и проверяют её значение в каждой итерации, если в методе run() есть циклы, или прерывают нити методом interrupt() для внезапной отмены заданий.

- **ЧТО ПРОИСХОДИТ, КОГДА В НИТИ ПОЯВЛЯЕТСЯ ИСКЛЮЧЕНИЕ?**

Это один из хороших вопросов с подвохом. Простыми словами, если исключение не поймано - нить мерта, если установлен обработчик непойманных исключений, он получит колбек. Thread.UncaughtExceptionHandler – интерфейс, определённый как вложенный интерфейс для обработчиков, вызываемых, когда нить внезапно останавливается из-за непойманного исключения. Когда нить собирается остановится из-за непойманного исключения, JVM проверит её на наличие UncaughtExceptionHandler, используя Thread.getUncaughtExceptionHandler(), и вызовет у обработчика метод uncaughtException(), передав нить и исключение в виде аргументов.