JVM поддерживает параллельные вычисления. В 2004 году (Java 5.0) был добавлен пакет java.util.concurrent.

**Что такое параллелизм?**

Под параллелизмом понимается способность программы выполнять несколько задач одновременно, что позволяет эффективно использовать системные ресурсы и повышать производительность.

Параллелизм: управление несколькими задачами одновременно  
Поток: Наименьшая единица обработки  
Синхронизация: Координация доступа нескольких потоков к общим ресурсам

Все начинается с главного (основного) потока, то есть минимально в твоей программе уже есть один выполняемый поток. Основной поток может создавать другие потоки с помощью Callable или Runnable. Создание отличается только возвращаемым результатом, Runnable не возвращает результата и не может выбросить проверяемое исключение. Поэтому у тебя получается хорошая возможность построить эффективную работу с файлами, но это очень опасно и нужно быть аккуратным.

На некоторых платформах запуск новых потоков может замедлить работу приложения. Что может иметь большое значение, если нам критичная производительность приложения.

Для каждого потока создается свой собственный стек в памяти, куда помещаются все локальные переменные и ряд других данных, связанных с выполнением потока.

**Синхронизация потоков в Java**

Коммуникация обеспечивается за счет разделения доступа к объектам. Это весьма эффективно, но в то же время очень легко допустить ошибку при работе. Ошибки бывают двух случаев: thread interference — когда другой поток вмешивается в твой поток, и memory consistency errors — консистентности памяти. Для решения и предотвращения этих ошибок у нас есть разные методы синхронизации.  
мониторы, — это высокоуровневый механизм, позволяющий единовременно только одному потоку выполнять блок кода, защищённый этим же монитором

Синхронизация имеет несколько важных моментов, на которых нужно обратить внимание. Первый момент — это взаимное исключение (mutual exclusion) — только один поток может владеть монитором, таким образом, синхронизация на мониторе подразумевает, что как только один поток входит в synchronized-блок, защищённый монитором, никакой другой поток не может войти в блок, защищенный этим монитором, пока первый поток не выйдет из synchronized-блока. То есть несколько потоков не могут обратится в один блок synchronized одновременно.

Чтение-запись в поле — это атомарная операция, если поле объявлено volatile, либо защищено уникальной блокировкой, получаемой перед любым чтением-записью. Но если ты все-таки столкнулся с ошибкой, то получаешь ошибку о переупорядочивании (изменение порядка следования, reordering). Она проявляется в некорректно синхронизированных многопоточных программах, где один поток может наблюдать эффекты, которые производятся другими потоками.

Эффект взаимного исключения и синхронизации потоков, то есть их корректная работа достигается только вхождением в synchronized-блок или метод, неявно получающий блокировку, или получением блокировки явным образом.

**Synchronizers**

Семафоры, как правило, используются, когда надо ограничить количество потоков при работе с файловой системой. Доступ к файлу или другому общему ресурсу управляется через счетчик. Если его значение больше нуля, доступ разрешен, но в тот же момент времени показания счетчика будут уменьшаться.

В тот момент, когда счетчик вернет ноль, текущий поток будет заблокирован до момента освобождения ресурса другим потоком. Параметр количества разрешений необходимо задавать через конструктор.

CountDownLatch — позволяет нескольким потокам ожидать, пока не завершится определенное количество операций, выполняемых в других потоках. В качестве примера можно представить установку приложения: она не начнется, пока ты не примешь правила пользования, пока не выберешь папку, куда устанавливать новую программу и так далее. Для этого есть специальный метод countDown() — этот метод уменьшает счетчик count down на единицу.

Как только счетчик становится равным нулю, все ожидающие потоки в await продолжат свою работу, а все последующие вызовы await будут проходить без ожиданий. Счетчик count down одноразовый и не может быть сброшен в первоначальное состояние.

CyclicBarrier — используется для синхронизации заданного количества потоков в одной точке. Барьер достигается в тот момент времени, когда N-потоков вызовут метод await(...) и блокируются. После чего счетчик сбрасывается в исходное значение, а ожидающие потоки будут освобождены. Дополнительно, если нужно, существует возможность запуска специального кода до разблокировки потоков и сброса счетчика. Для этого через конструктор передается объект с реализацией интерфейса Runnable.

Класс Phaser позволяет синхронизировать потоки, представляющие отдельную фазу или стадию выполнения общего действия. Phaser определяет объект синхронизации, который ждет, пока не завершится определенная фаза. Затем Phaser переходит к следующей стадии или фазе и снова ожидает ее завершения.

При работе с классом Phaser обычно сначала создается его объект. Далее нам надо зарегистрировать всех участников. Для регистрации для каждого участника вызывается метод register(), либо можно обойтись и без этого метода, передав нужное количество участников в конструктор Phaser.

Затем каждый участник выполняет некоторый набор действий, составляющих фазу. А синхронизатор Phaser ждет, пока все участники не завершат выполнение фазы. Чтобы сообщить синхронизатору, что фаза завершена, участник должен вызвать метод arrive() или arriveAndAwaitAdvance(). После этого синхронизатор переходит к следующей фазе.

**Volatile поля**

Запись работает как освобождение монитора, а чтение — как захват монитора. Доступ осуществляется в отношении по типу “выполняется прежде”. Если разобраться, то все, что будет видно для потока A, когда он обращался к volatile переменной, — это переменная для потока B. То есть вы гарантированно не потеряете ваши изменения из других потоков.

**Атомарные операции**

Алгоритм использует низкоуровневые машинные инструкции, такие как сравнение и замена (CAS, compare-and-swap, что обеспечивает целостность данных и по ним уже существует большое количество исследований).

CAS атомарно обновляет M до B, но только если значение M совпадает с A, в противном случае никаких действий предприниматься не будет.

В первом и втором случае вернут значение М. Это позволяет объединить три шага, а именно — получение значения, сравнение значения и его обновление. И это все превращается в одну операцию на машинном уровне.

В тот момент времени, когда многопоточное приложение обращается к переменной и пытается обновить его и применяется CAS, то один из потоков получит его и сможет обновить его. Но в отличии от блокировок, другие потоки просто получат ошибки о том, что им не удалось обновить значение. Потом они перейдут к дальнейшей работе, а переключение полностью исключено при таком типе работе.

**Locks: блокировка доступа к ресурсам**

**Condition** — применение условий в блокировках позволяет добиться контроля над управлением доступа к потокам. Условие блокировки представлет собой объект интерфейса Condition из пакета java.util.concurrent.locks. Применение объектов **Condition** во многом аналогично использованию методов wait/notify/notifyAll класса Object.