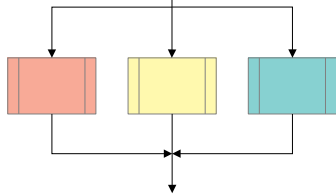




# Многопоточность. Теория

### • Thread

- нить, резьба
- поток
  - ◊ поток выполнения
  - ◊ параллельно



### • Stream

- поток, ручей
- поток
  - ◊ поток данных
  - ◊ последовательно



Сначала, как обычно, немного терминологии. В обычных словарях «thread» переводится как «нить», и этот термин в последнее время часто употребляется. Однако, еще с прошлого тысячелетия, особенно в учебниках и литературе, для термина «multithreading» устоялся перевод «многопоточность», поэтому и «thread» традиционно называется потоком.

И это уже третий случай употребления слова «поток»:

- 1) поток ввода-вывода, он же I/O stream, он же поток байтов и символов, он же InputStream и OutputStream.
- 2) поток данных, он же data stream (в Stream API), он же стрим, он же объект конвейерной обработки
- 3) поток исполнения, он же Thread, он же нить, он же легковесный процесс

В общем случае можно считать, что Thread — это параллельный поток исполнения кода, а Stream — поток элементов данных, которые следуют друг за другом.

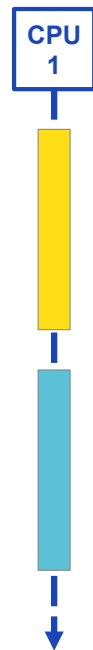
- Один процессор
- Одна задача

CPU  
1



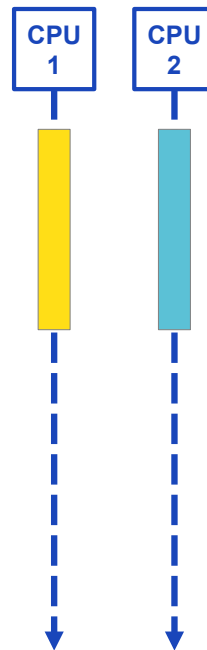
Для того, чтобы разобраться, как могут программы исполняться параллельно, чем параллельность отличается от конкурентности, и что такое многозадачность, рассмотрим упрощенные примеры. Начнем с самого простого случая, когда одна задача выполняется одним процессором. Пунктирная ось, направленная вниз — это время. Цветным прямоугольником обозначается период, когда устройство (а именно процессор №1), занято выполнением задачи.

- Один процессор
- Две задачи
- Последовательное исполнение
- Простое управление



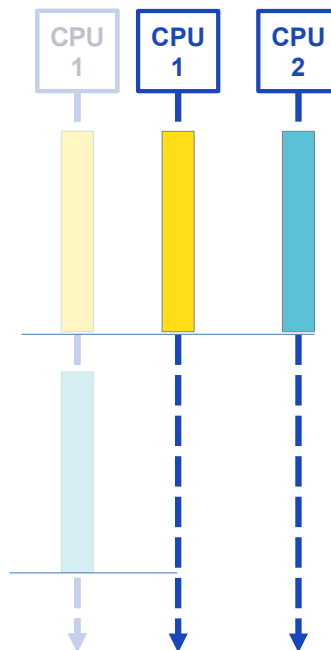
Добавим вторую задачу. Она начинает выполняться после того, как будет выполнена первая. Такой способ исполнения называется последовательным. В этом случае управление задачами довольно простое, и оно заключается только в том, чтобы запускать на исполнение очередную задачу после того, как предыдущая полностью завершится.

- Два процессора
- Две задачи
- Параллельное исполнение
- Более сложное управление



Добавим второй процессор, который может заняться исполнением второй задачи, пока первый процессор занят исполнением первой задачи. Задачи не зависят друг от друга, поэтому могут выполняться одновременно. Такой вид исполнения называется параллельным. При этом управление становится несколько сложнее, так как необходимо распределять задачи между процессорами.

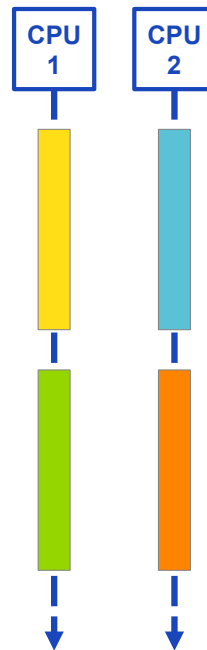
- Два процессора
- Две задачи
- Параллельное исполнение
- Более сложное управление



Очевидно, что время выполнения двух задач параллельно двумя процессорами намного меньше, чем время выполнения этих же задач одним процессором последовательно.

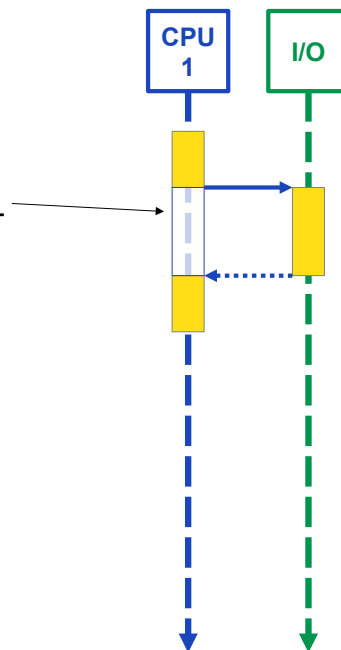
Кстати, если по каким-то причинам параллельное исполнение нельзя применить, например, вторая задача должна использовать результат, полученный после исполнения первой задачи, то мы вынуждены выполнять их последовательно, даже при наличии второго процессора, который в этом случае будет бесполезно простаивать.

- Много процессоров
- Много задач
- Параллельное исполнение
- Высокая производительность



Но, если задачи удастся распараллелить, то можно получить значительный прирост производительности при использовании большого количества процессоров. При этом каждый процессор свои задачи исполняет последовательно.

- Один процессор
- Одна задача
- Ввод-вывод
- Процессор простаивает

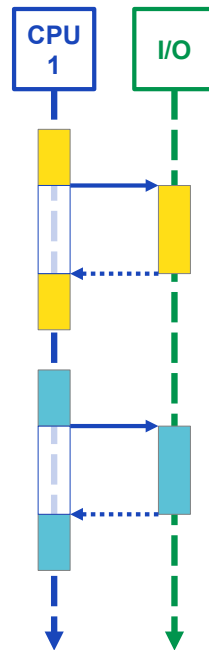


Рассмотрим более интересный случай. Допустим, что нужно произвести обмен данными с каким-то внешним устройством. Также допустим, что у этого устройства есть контроллер, который способен, получив команду, читать или записывать данные напрямую в память, без участия процессора.

Сначала рассмотрим случай с одним процессором и одной задачей. При этом, в середине исполнения задачи управление передается контроллеру, а процессор в это время ничего не делает — ждет окончания обмена данными.

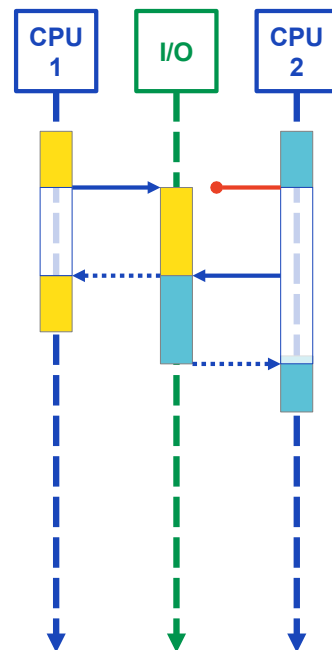


- Один процессор
- Две задачи
- Ввод-вывод
- Процессор простаивает



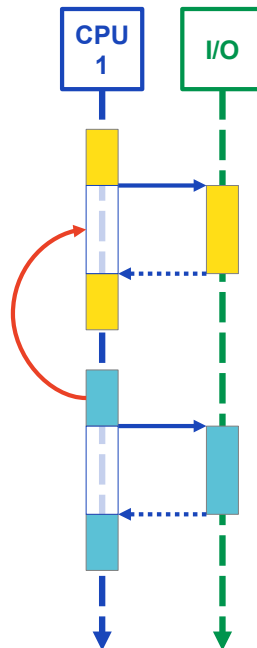
Добавим вторую задачу, которая начнет выполняться после окончания выполнения первой задачи. Во время выполнения второй задачи процессор также ожидает завершения операции ввода-вывода.

- Два процессора
- Две задачи
- Ввод-вывод
- Процессор простаивает
- Занятость В/У



Добавим второй процессор. В этом случае ситуация немного поменяется по сравнению со случаем, где не было ввода-вывода. Два процессора начинают параллельно исполнять свои задачи, но, если первый процессор уже отдал команду контроллеру на обмен данными с внешним устройством, то второй процессор не может этого сделать, так как контроллер в этот момент уже занят и не может обработать запрос. Поэтому второму процессору приходится ждать, когда внешнее устройство освободится и начнет обмен данными. При этом, время выполнения двух задач двумя процессорами становится больше, чем было для задач без ввода-вывода, так как появилось узкое место — единственный контроллер, который может задерживать выполнение задач в некоторые моменты времени.

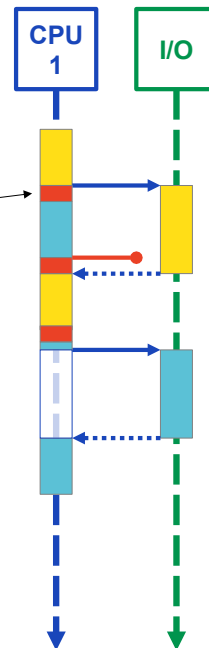
- Один процессор
- Две задачи
- Ввод-вывод
- Процессор **простаивает!**



Вернемся к случаю с одним процессором и двумя задачами. Можно ли здесь попробовать сократить время выполнения? Видно, что в те моменты, когда работает процессор, контроллер ввода-вывода ждет команды на обмен данными. А пока контроллер передает данные, процессор ждет завершения операции ввода-вывода.

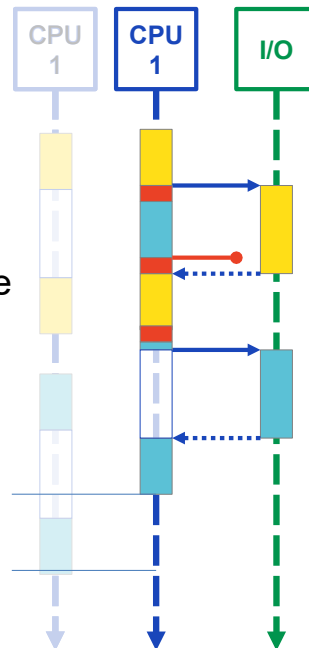
А что, если попробовать занять процессор во время простоя? Можно занять его началом выполнения второй задачи.

- Один процессор
- Две задачи
- Ввод-вывод
- Переключение контекста
- Конкурентное исполнение
- Более сложное управление



После того, как контроллер ввода-вывода получит от процессора команду на обмен данными, можно сохранить состояние процессора (счетчик команд, состояние стека, состояние памяти) и загрузить необходимые данные для второй задачи, то есть выполнить переключение контекста задач. После чего процессор начинает выполнять вторую задачу. Однако, операция ввода-вывода для второй задачи не запустится, так как контроллер пока занят и не может обработать запрос. Поэтому имеет смысл сохранить контекст второй задачи и загрузить контекст первой. После чего можно продолжить выполнение первой задачи, получить от контроллера извещение об окончании ввода-вывода и завершить первую задачу. Далее восстановить контекст второй задачи и продолжить ее выполнение.

- Один процессор
- Две задачи
- Ввод-вывод
- Переключение контекста
- Конкурентное исполнение
- Более сложное управление



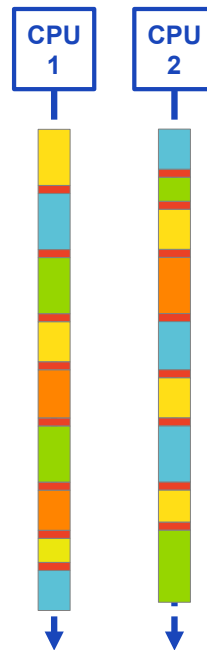
При этом, часть времени будет потрачена на переключение контекста, но все равно общее время выполнения двух задач будет меньше, чем при последовательном исполнении. Такой вид исполнения называется конкурентным (или одновременным). Исполнитель один, но он не ждет полного завершения задачи, а может в свободное время переключаться на другие задачи. При этом требуется еще более сложное управление, так как при переключении между задачами должен сохраняться контекст задач, чтобы их можно было продолжить с момента переключения.

- Один процессор
- Много задач
- Многозадачность
- Конкурентное исполнение



Таким образом, можно выполнять и много задач, при этом не обязательно всегда ждать начала операции ввода-вывода, можно переключать задачи через определенные промежутки времени. При конкурентном исполнении даже на одном процессоре, несмотря на то, что в один момент времени выполняется только одна задача, все задачи постепенно выполняются и продвигаются к своему завершению. У каждой задачи есть некий прогресс. Способность одновременно выполнять много задач называется многозадачностью.

- Много процессоров
- Много задач
- Многозадачность
- Конкурентное исполнение



Если у нас много процессоров, мы можем этот же принцип распространить на все процессоры. Таким образом, можно использовать абстракцию многозадачности. Неважно, сколько процессоров имеется — один или много. Неважно, как именно организован процесс переключения между задачами. Неважно, в каком порядке задачи будут выполняться. Для нас важно, что если у нас есть много задач, все они будут выполняться постепенно и условно одновременно — то есть многозадачно. Желательно при этом стремиться к уменьшению времени, затрачиваемому на переключение контекста.



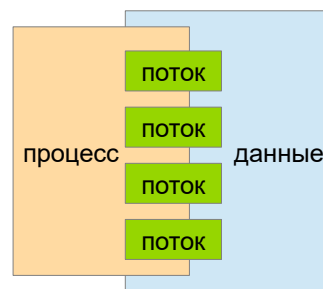
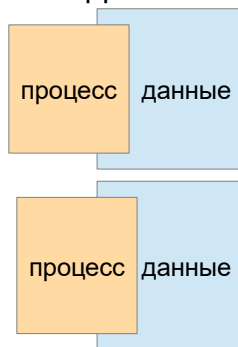
## Стратегии переключения задач

- Кооперативная многозадачность
  - Добровольное переключение в удобный момент
  - Если задача не делится ресурсами — другие страдают
- Вытесняющая многозадачность
  - Задачи переключает диспетчер
  - Переключение в произвольные моменты
  - Необходимо сохранения состояния
  - Необходимо согласование доступа

Наиболее часто используются две стратегии переключения задач. Кооперативная многозадачность когда задача сама решает, в какой момент она приостановит выполнение и даст шанс другим задачам. Плюс в том, что задача это делает в удобный для нее момент, который предусмотрел программист. Недостаток подхода в том, что если задача не делится ресурсами, то другие задачи ничего не могут поделать. Вытесняющая многозадачность подразумевает, что переключением управляет отдельный диспетчер, который переключает задачи по определенному алгоритму, либо в моменты, когда задача ожидает освобождения ресурса, либо выделяет каждой задаче определенный квант времени, а когда он заканчивается, переключает задачи. В этом случае при написании программ надо учитывать возможность переключения в любой момент времени, даже когда какое-то действие не завершилось полностью.



- Процесс
  - отдельное приложение
  - свои ресурсы и память
  - долгое переключение контекста
- Многозадачность
- Поток (thread)
  - работает внутри процесса
  - общие ресурсы и память
  - быстрое переключение контекста
- Многопоточность



Многозадачностью традиционно называется управление процессами со стороны операционной системы. Процесс — это запущенное отдельное приложение, которому операционная система выделяет определенный набор ресурсов, в том числе некоторую область памяти. Обмен данными между процессами довольно сложен и требует специальных средств межпроцессного взаимодействия. На уровне отдельного приложения можно говорить о многопоточности. Поток (thread) — это более мелкая единица исполнения. В одном процессе можно запустить несколько потоков, при этом они имеют общий доступ к памяти и ресурсам. Переключение контекста между потоками занимает значительно меньше времени, чем между процессами. Новый поток может быть запущен внутри процесса.

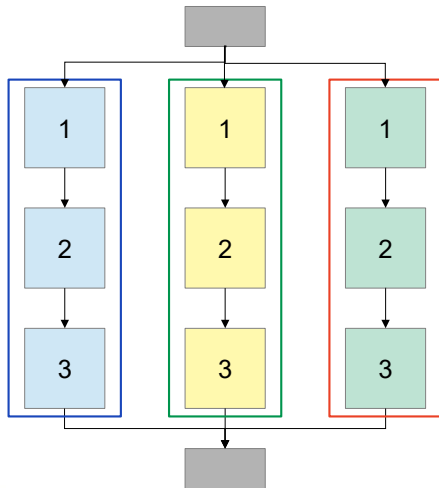


## Средства для обеспечения параллельной обработки

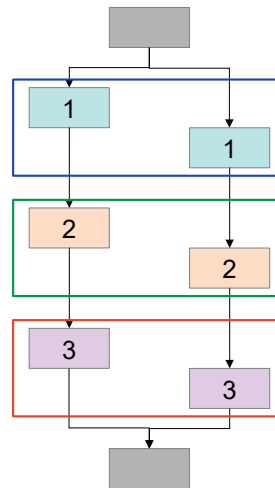
- Аппаратные
  - Многопроцессорность
  - Многоядерность
  - Многопоточность — переключение контекста на уровне процессора
    - Временная (Temporal) — в один момент времени один поток
    - Одновременная (Simultaneous) — в один момент времени несколько потоков
- Программные
  - Многозадачность
  - Многопоточность на уровне ядра ОС
  - Многопоточность на пользовательском уровне
    - Green Threads — потоки, управляемые виртуальной машиной

Для поддержки параллельной обработки используются аппаратные и программные средства. На аппаратном уровне может быть много процессоров, много ядер в каждом процессоре, можно организовать аппаратную многопоточность в процессоре. С точки зрения ОС эти элементы воспринимаются как отдельные процессоры. На программном уровне используется многозадачность, многопоточность на уровне ядра ОС, когда потоки создаются функциями ядра операционной системы. Также многопоточность может быть реализована в пользовательском пространстве, например, виртуальной машиной и другими средствами без обращения к функциям ядра ОС. Сейчас в реализации JVM Hotspot поток виртуальной машины соответствует потоку ядра ОС, однако некоторое время назад были варианты с использованием green threads, которыми управляла виртуальная машина.

- распараллеливание
- обработчик выполняет все этапы одной задачи



- конвейерная обработка
- обработчик выполняет один этап всех задач



Ну и в конце рассмотрим два варианта организации многопоточной обработки нескольких задач. В одном случае (распараллеливание) каждой задаче выделяется свой обработчик, который полностью выполняет эту задачу от начала до конца. В другом случае (конвейеризация), каждый обработчик выполняет один из этапов всех задач, после чего передает результат выполнения следующему обработчику, а сам принимается за этот же этап очередной задачи.

# Многопоточность. Java

- Потоки в JVM
  - системные потоки:
    - ◊ основной поток виртуальной машины
    - ◊ сборщик мусора
    - ◊ поток периодических задач
    - ◊ поток динамической компиляции
  - прикладные потоки
    - ◊ основной поток (main)
    - ◊ создаваемые программным путем

При старте JVM создаются системные потоки (основной поток виртуальной машины, сборщик мусора, поток периодических задач, поток динамической компиляции). После загрузки и инициализации основного класса, запускается основной поток программы (метод main). Если программа не создает собственные потоки (однопоточная), то она так и остается единственным прикладным потоком. Основной поток может создавать дополнительные потоки, тогда программа становится многопоточной.



- class **Thread** implements **Runnable**
- 2 варианта создания потока
  - 1) 

```
public class MyClass extends Thread {  
    public void run() { /* тело потока*/ }  
}  
new MyClass().start();
```
  - 2) 

```
public class MyClass implements Runnable {  
    public void run() { /* тело потока */ }  
}  
new Thread(new MyClass()).start();
```
  - 2λ) 

```
Runnable r = () -> { /* тело потока */ };  
Thread t = new Thread(r); t.start();
```

Для создания и управления потоками в Java имеется класс `Thread` и интерфейс `Runnable`. Класс `Thread` реализует интерфейс `Runnable`.

Соответственно, есть 2 варианта создания потока:

- 1) Наследование от класса `Thread` с реализацией метода `run()`, в котором находится код, выполняемый потоком, далее можно запустить этот поток методом `start()`.
- 2) Реализовать интерфейс `Runnable`, точно так же поместив код, который должен выполнить поток, в метод `run()`. Далее создать экземпляр класса, передать его в конструктор класса `Thread`, и вызвать у получившегося потока метод `start()`
- 3) Так как `Runnable` — функциональный интерфейс, то можно использовать и лямбда-выражение (а также и анонимный класс)



- класс Thread — управление потоками
  - метод start()
  - метод не ждет завершения потока
- интерфейс Runnable — реализация потока
  - метод run()
  - завершается метод — завершается поток
- Что будет, если вызвать метод run() напрямую?

Хотя код, который выполняет поток, находится в методе run(), непосредственно этот метод не выполняют, а для запуска потока используют метод start(). При запуске метода start() поток переходит в состояние готового к выполнению, и дальше он управляется диспетчером, который решает, когда поток начнет реально исполняться, то есть запустится его метод run().

Метод start() не ждет завершения потока, а сразу возвращается, и дальше можно запускать остальные потоки — все они будут выполняться одновременно (конкурентно) с основным.

Когда метод run() завершается — поток также завершается и больше не может быть выполнен. Ну и остается вопрос — что будет, если запустить метод run() напрямую? Ничего страшного, он выполнится, но в основном потоке. При этом никакой многопоточности, конечно, не получится.

- метод `interrupt()`
  - прерывает выполнение некоторых методов (`sleep`, `join`, `wait`) прерыванием `InterruptedException`
    - ◊ прерывание можно поймать и завершить поток (выйти из `run()`)
    - ◊ прерывание можно пробросить
  - устанавливает флаг прерывания потока
    - ◊ флаг можно проверить и завершить поток
    - ◊ флаг можно игнорировать
- проверка флага
  - `Thread.interrupted()` - сбрасывает флаг
  - `isInterrupted()` - не сбрасывает флаг

Иногда бывает нужно прервать выполнение потока. Это делается с помощью метода `interrupt()`. Если поток в этот момент выполняет один из методов `sleep()`, `join()` или `wait()`, то метод `interrupt` вызывает выброс исключения `InterruptedException`, которое можно обработать и, например, выйти из потока. Если же поток выполняет другие действия, то метод `interrupt()` устанавливает флаг прерывания, извещая поток, что хорошо бы ему прерваться. Если поток периодически проверяет флаг, он может завершиться (обычно это и надо), но в каких-то случаях поток может просто игнорировать запрос на прерывание. Проверить флаг можно либо с помощью статического метода `Thread.interrupted()`, который сбрасывает флаг после проверки, либо вызвав для конкретного потока метод `isInterrupted()`, который не меняет состояние флага.





## Приостановка и ожидание завершения потока

- `Thread.sleep(1000); // спать 1000 мс`
- `t.join(); // ожидать завершения t`
- и другие полезные методы
  - `Thread.currentThread()`
  - `getId()`
  - `getName()`
  - `isAlive()`
  - `getPriority()`
  - `getThreadGroup()`

25

Можно приостановить выполнение потока на определенное время с помощью метода `sleep()`, которому передается время сна в мс. И еще можно дождаться завершения выполнения некоторого потока, вызвав у него метод `join()`. Также в классе `Thread` есть много других полезных методов, о которых можно прочитать в документации. С их помощью можно получить ссылку на текущий исполняющийся поток, получить идентификатор или имя потока, узнать, живой ли поток, получить приоритет потока и узнать группу, к которой поток принадлежит.

```
public class ThreadTest {  
    public static void main(String[] args) {  
        Runnable r = () -> {  
            String name = Thread.currentThread().getName();  
            System.out.println(name + " started");  
            try {  
                Thread.sleep(500 + (long)(100 * Math.random()));  
            } catch (InterruptedException e) { return; }  
            System.out.println(name + " finished");  
        };  
        for (int i = 0; i < 10; i++) {  
            (new Thread(r)).start();  
        }  
    }  
}
```

В данном примере определяется код потока, который сначала пишет сообщение, что он запустился, потом спит 500-600 миллисекунд, выводит сообщение, что поток завершился, и заканчивается. Потом в цикле запускается 10 таких потоков одновременно.



## Результат работы

```
Thread-0 started      Thread-8 finished
Thread-2 started      Thread-7 finished
Thread-9 started      Thread-9 finished
Thread-8 started      Thread-5 finished
Thread-6 started      Thread-0 finished
Thread-7 started      Thread-1 finished
Thread-5 started      Thread-2 finished
Thread-1 started      Thread-6 finished
Thread-4 started      Thread-4 finished
Thread-3 started      Thread-3 finished
```

27

Как можно увидеть, при запуске потоки выводят свои сообщения не обязательно в том порядке, в каком они были запущены. И завершаются они тоже в случайном порядке. Более того, этот порядок меняется с каждым запуском программы.

Из-за того, что выполнение многопоточной программы недетерминировано, при работе многопоточных программ могут возникать различные проблемы, и их не очень просто отлаживать.



## Гонки (race condition)

```
class Shared {  
    int counter = 1;  
    void up() { counter++; }  
    void down() { counter--; }  
}
```

```
Shared sh = new Shared();  
new Thread(sh::down).start();  
new Thread(sh::up).start();
```

28

Одной из распространенных проблем многопоточных программ является состояние гонок. Оно возникает, когда к одной и той же общей переменной обращается несколько потоков, и по крайней мере один из них производит запись значения этой переменной. В данном примере есть общая переменная counter, представляющая собой некоторый счетчик, и два метода, один из которых увеличивает значение счетчика, а другой — уменьшает. Методы запускаются из разных потоков. Один поток вызывает метод up(), другой — down(). Пример не обязательно всегда будет работать именно так, но возможность подобного исхода есть, особенно когда такие действия повторяются много раз.

```
class Shared {
    int counter = 1;
    void up() { counter++; }
    void down() { counter--; }
}
```

```
1. load counter
2. add 1 (sub 1)
3. store counter
```

```
Shared sh = new Shared();
new Thread(sh::down).start();
new Thread(sh::up).start();
```

поток 1	counter	поток 2
load counter (1)	1	
	1	load counter (1)
	1	add 1 (2)
	2	store counter (2)
sub 1 (0)	2	
store counter (0)	0	

Проблема может возникнуть из-за того, что переключение между потоками может случиться в любой момент времени, в том числе между выполнением операций. Операции инкремента и декремента выглядят так, как будто в них выполняется одиночное действие. Но на самом деле на уровне байт-кода, инкремент и декремент состоят из последовательности операций — загрузить значение, добавить или отнять единицу, записать значение обратно.

Поэтому, есть вероятность, что сразу после того, как первый поток получит значение счетчика, управление перейдет второму потоку, который выполнит инкремент и запишет значение в счетчик, после чего снова начнет работать первый поток, который, не зная от том, что значение счетчика уже изменилось, выполнит операцию декремента со старым значением и сохранит его. В итоге вместо ожидаемого результата 1 мы увидим 0.

```
class Shared {
  int counter = 1;
  void up() { counter++; }
  void down() { counter--; }
}
```

**КРИТИЧЕСКАЯ СЕКЦИЯ**

1. load counter
2. add 1 (sub 1)
3. store counter

```
Shared sh = new Shared();
new Thread(sh::down).start();
new Thread(sh::up).start();
```

поток 1	counter	поток 2
load counter (1)	1	
	1	load counter (1)
	1	add 1 (2)
	2	store counter (2)
sub 1 (0)	2	
store counter (0)	0	

Часть кода, которая содержит операции чтения или записи общих переменных, называется критической секцией. Для корректной работы желательно обеспечить атомарность операций. То есть, нужно сделать так, чтобы поток, начав выполнять операцию инкремента, не мог быть прерван в середине операции.



```
class Shared {  
    int counter = 0;  
    synchronized void up() { counter++; }  
    synchronized void down() { counter--; }  
}
```

- **Защита критической секции** от выполнения двумя потоками
- Любой объект имеет **встроенную блокировку** (intrinsic lock)
- При **входе** в критическую секцию поток **захватывает** блокировку
- При **выходе** из критической секции поток **освобождает** блокировку
- Блокировка **реентерабельна** — не блокирует себя
- **synchronized** — защита критической секции (метода или блока) по определенному объекту
  - обычный метод — по объекту, у которого вызван метод
  - статический метод — по объекту класса Class (
  - по явно указанному объекту для synchronized блока

Это можно сделать с помощью модификатора `synchronized` для методов `up()` и `down()`. Модификатор не позволяет этому методу выполняться двумя потоками. Как только первый поток начинает выполнять синхронизированный метод, он захватывает блокировку, при выходе из метода поток освобождает блокировку. Выполнять метод может только поток, захвативший блокировку. При этом он же может заходить и в другие синхронизированные методы (блокировка реентерабельна), но потом поток должен освободить блокировку столько раз, сколько он ее захватил. Синхронизировать можно не только обычный метод, но и статический. А также можно синхронизировать небольшой блок кода. Всегда желательно синхронизировать небольшие участки кода, так как пока он выполняется, другие потоки будут ждать своей очереди, и лучше сократить это время ожидания и не блокировать лишнее.

```

public class MyClass {
    Object lock = new Object(),
    public synchronized void add() { }
    public synchronized void rem() { }
    public static synchronized int min() { }
    public static synchronized int max() { }
    public void x() { ... synchronized (lock) { } ... }
    public void y() { ... synchronized (this) { } ... }
}
MyClass m = new MyClass();
m.add(); // один поток начал выполнять m.add()
// другие потоки не могут вызвать m.add(), m.rem()
// и войти в синхронизированный блок внутри метода m.y()

MyClass.min(); // один поток начал выполнять min()
// другие потоки не могут вызвать min(), max()

m.x(); // один поток вошел в блок внутри метода x()
// другие потоки не могут войти в любой блок,
// синхронизированный по объекту lock

```

В данном примере методы `add()` и `rem()` синхронизированы по объекту класса `MyClass`, по нему же синхронизирован блок внутри метода `y()`.

Соответственно, если один поток вызвал у какого-то объекта класса `MyClass` метод `add()`, то другие потоки не смогут вызвать методы `add()` и `rem()` у этого же объекта, а также начать выполнение синхронизированного блока внутри метода `y()`. При этом у других объектов данные методы не блокируются.

Методы `min()` и `max()` блокируются для всего класса `MyClass` (точнее для объекта `Myclass.class`).

Ну и синхронизированный блок внутри метода `x()` блокируется по объекту `lock`. Таким образом можно создавать разные группы блокировок.





## Проблема видимости

- Процессор может сохранять значения переменных в локальном кэше для повышения производительности
- Компилятор и JVM могут оптимизировать порядок выполнения инструкций

```
(new Thread(() -> { while (!done) i++; })).start();  
Thread.sleep(1000);  
done = true; // первый поток остановится через 1 с  
  
a = 0, b = 0;  
x = 0, y = 0;  
m1() { b = 1; x = a; }  
m2() { a = 2; y = b; }  
  
// x = 2; y = 0;  
// x = 0; y = 1;
```

33

Еще одной проблемой многопоточности является проблема видимости переменных. Во-первых, она проявляется при работе разных потоков на разных процессорах, так как процессор для повышения производительности может хранить значения переменных в своем кэше. Ситуация похожа на гонки, так как один процессор может сохранить новое значение, в то время как второй процессор использует старое значение, которое хранится в его кэше. Во-вторых, компилятор и JVM имеет право менять порядок инструкций (также для повышения производительности). В первом примере запускается один поток, который должен завершиться, когда в основном потоке через 1 с установится переменная `done`. Во втором примере методы `m1()` и `m2()` запускаются в разных потоках, и в зависимости от того, какой из них выполнится раньше, можно получить один из двух результатов.

- Процессор может сохранять значения переменных в локальном кэше для повышения производительности
- Компилятор и JVM могут оптимизировать порядок выполнения инструкций

```
(new Thread(() -> { while (!done) i++; })).start();
Thread.sleep(1000);
done = true; // первый поток остановится через 1 с

a = 0, b = 0;
x = 0, y = 0;
m1() { b = 1; x = a; }
m2() { a = 2; y = b; }

// x = 2; y = 1;
```

На самом деле, во втором примере возможен еще один результат. Если произойдет переключение потоков либо после  $b = 1$ , либо после  $a = 2$ , то мы получим в итоге  $x = 1$ ,  $y = 2$ . Проследите, как это может произойти.

- Процессор может сохранять значения переменных в локальном кэше для повышения производительности
- Компилятор и JVM могут оптимизировать порядок выполнения инструкций

```
(new Thread(() -> { while (!done) i++; })).start();
```

```
Thread.sleep(1000);
done = true;
```

```
a = 0, b = 0;
```

```
x = 0, y = 0;
```

```
m1() { b = 1; x = a; }
```

```
m2() { a = 2; y = b; }
```

```
// x = 0; y = 0;
```

```
if (!done)
while (true)
i++;
```

```
x = a; b = 1;
```

```
y = b; a = 2;
```

Но и это еще не все.

В первом примере код может быть оптимизирован, так как значение `done` не меняется в цикле `while` — тогда зачем ее проверять в цикл, если это можно сделать заранее. В результате, поток не остановится через 1 секунду, а будет работать бесконечно.

Во втором примере, если рассматривать методы отдельно друг от друга, то порядок операций в каждом методе не имеет значения. Поэтому в каких-то случаях компилятор может поменять их местами. И тогда при запуске из двух потоков возможен случай, когда и `x`, и `y` будут равны 0.

При оптимизации случаи работы в нескольких потоках не рассматриваются, иначе алгоритмы оптимизации получились бы слишком сложными.

- Модификатор `volatile` — переменная может быть использована не в текущем потоке
- Операции чтения-записи переменной с модификатором `volatile` должны выполняться без использования кэша
- Порядок операций чтения-записи переменной с модификатором `volatile` не должен меняться — должно соблюдаться ограничение «happens-before»
  - **Запись** `volatile` переменной должна выполняться **ПОСЛЕ** предшествующих операций чтения и записи других переменных
  - **Чтение** `volatile` переменной должно выполняться **ДО** последующих операций чтения и записи других переменных

Решить проблему во втором случае можно синхронизацией методов `m1` и `m2`, но это не всегда оправдано, тем более, что это не решает проблему цикла `while` в первом случае.

Есть другое решение — модификатор `volatile`. Он говорит о том, что значение переменной может быть изменено другим потоком, поэтому нельзя кэшировать, и нельзя менять порядок операций с такой переменной.

Более того, в Java накладываются дополнительные ограничения, а именно:

Запись `volatile` переменной должна выполняться после расположенных в коде до этой записи операций чтения и записи других переменных

Чтение `volatile` переменной должно выполняться до расположенных в коде после этого чтения операций чтения и записи других переменных

```

volatile boolean done;
(new Thread(() -> { while (!done) i++; })).start();
Thread.sleep(1000);
done = true; // первый поток остановится через 1 с

volatile a = 0, b = 0;
x = 0, y = 0;
m1() { b = 1; x = a; }
m2() { a = 2; y = b; }

// x = 2; y = 0;
// x = 0; y = 1;
// x = 2; y = 1;

```

В итоге, приведенные примеры можно поправить следующим образом: переменную `done` объявить как `volatile`, тогда цикл `while` не будет оптимизироваться. И также можно объявить переменные `a` и `b` как `volatile`, при этом их порядок нельзя будет менять.



## Класс ThreadLocal

- У каждого потока есть стек, где хранятся локальные переменные и параметры методов
- Примитивные локальные переменные потокобезопасны
- Переменные класса — общие для всех потоков
- ThreadLocal — потокобезопасные переменные

```
private ThreadLocal<String> tl = new ThreadLocal();  
tl.set("local");  
String s = tl.get();
```

38

Каждый поток имеет свой собственный стек, где хранятся локальные переменные и параметры методов. При этом, если они являются примитивными, то другие потоки не могут получить к ним доступ, поэтому такие переменные потоки могут использовать, не сталкиваясь с рассмотренными проблемами общего доступа. Со ссылочными переменными надо быть осторожнее, так как ссылка у потока своя, но сам объект хранится в куче, где к нему могут иметь доступ и другие потоки. Также общими переменными являются поля класса.

Если необходимо использовать переменную так, чтобы в каждом потоке она имела свое значение, то это можно сделать с помощью класса ThreadLocal. Можно создать объект этого класса, хранящий нужное значение, и с помощью методов set() и get() работать с этим значением.

- Вариант 1 — общая переменная и флаг

```

class Block {
    volatile boolean ready;
    int value;

    void put(int i) {
        while (ready);
        synchronized(this) {
            value = i;
            ready = true;
        }
    }
    int get() {
        while (!ready);
        synchronized(this) {
            ready = false;
            return value;
        }
    }
}

Block g = new Block();
Thread t1 = new Thread(() -> {
    g.put(100);
});
Thread t2 = new Thread(() -> {
    System.out.println(g.get());
});
t1.start();
t2.start();
    
```

busy-wait

Иногда нужно обеспечить взаимодействие потоков. Например, в случае, когда один поток генерирует значения, а другие потоки забирают их. Для простоты рассмотрим случай, когда значения передаются по одному. Один поток в методе `put()` при сброшенном флаге `ready` записывает значение в переменную `value` и устанавливает флаг `ready`, что значение готово. Другой поток в методе `get()` ждет, пока флаг не станет равным `true`, читает значение, и сбрасывает флаг, показывая что можно выдавать новое. Запись значение в переменную `value` должно находиться с сихнхронизированном блоке, чтобы исключить гонки, а флаг `ready` должен иметь модификатор `volatile`, чтобы компилятор не оптимизировал циклы с его проверкой. Недостаток данного способа реализации в том, что потоки нагружают процессор, пока ждут нужного значения флага. Такой вариант работы называется `busy-wait`.

- Методы `wait()`, `notify()`, `notifyAll()` вызываются только после захвата блокировки
- `wait()`
  - поток помещается в очередь ожидания (`wait set`) объекта
  - поток освобождает блокировку и ждет:
    - ◊ сигнал `notify`
    - ◊ прерывание
    - ◊ окончание времени ожидания
  - поток получает блокировку и завершает метод `wait()`
- `notify()` - выводит из очереди ожидания один из потоков.
- `notifyAll()` - выводит из очереди ожидания все потоки.

Чтобы зря не грузить процессор, можно использовать методы `wait` и `notify` класса `Object`. Эти методы разрешено вызывать только при наличии захваченной блокировки, то есть внутри синхронизированного метода или блока.

При вызове метода `wait()` поток помещается в очередь ожидания (`wait set`) объекта, у которого вызван метод. Дальше поток освобождает блокировку и начинает ждать. Ждет он, пока у объекта не будет вызван метод `notify`, либо поток не получит прерывание, либо (если метод `wait` был вызван с параметром времени) не истечет указанное время. Дальше поток вместе с другими кандидатами пытается захватить блокировку. Если ему это удастся, он продолжает работу после метода `wait`. Если нет — поток возвращается в состояние ожидания.

Метод `notify` выводит из ожидания один случайный поток, метод `notifyAll` — все потоки, которые ждут, но только один из них войдет в синхронизированный блок.



- Вариант 1 — wait / notify

```

class Block {
    volatile boolean ready;
    int value;

    synchronized void put(int i) {
        while (ready) wait();
        value = i;
        ready = true;
        notifyAll();
    }

    synchronized int get() {
        while (!ready) wait();
        ready = false;
        notifyall();
        return value;
    }
}

Block g = new Block();
Thread t1 = new Thread(() -> {
    g.put(100);
});
Thread t2 = new Thread(() -> {
    System.out.println(g.get());
});

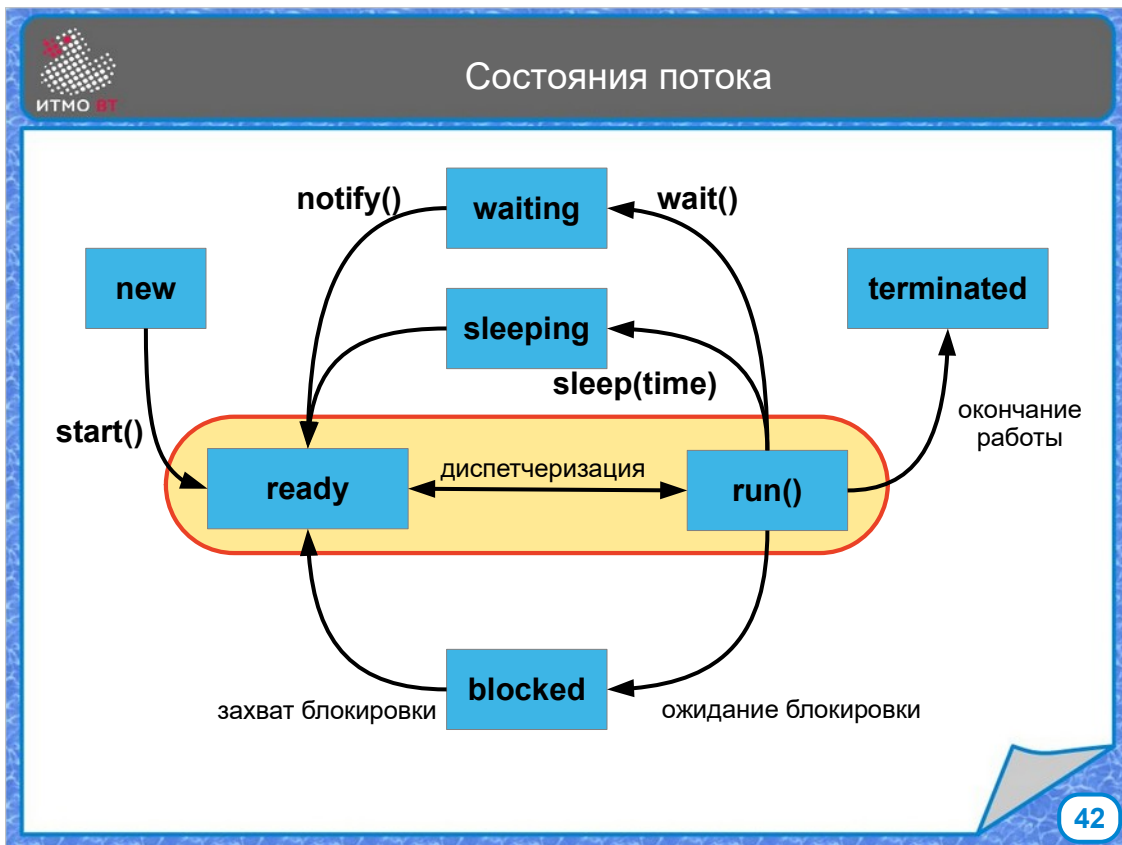
t1.start();
t2.start();
    
```

spin lock

В данном примере показана реализация взаимодействия потоков с помощью wait/notify. Она похожа на первый способ, но теперь процессор не выполняет цикл while, так как поток уходит в состояние ожидания и перестает выполняться. Другой поток выполнит свое действие и поменяет значение флага, после чего вызовет notify(), чтобы вывести ожидающий поток из состояния ожидания.

При работе методов wait и notify могут случаться спонтанные выходы потоков из состояния ожидания (ложные срабатывания), поэтому на всякий случай вызов метода wait() размещен внутри цикла проверки флага (такой способ называется spin lock). Если поток выйдет из ожидания при неверном значении флага, он снова станет ждать.

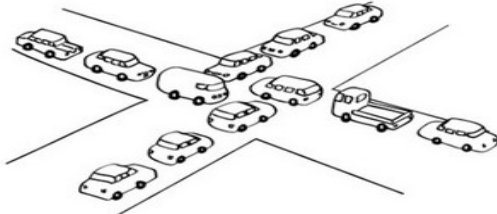
В данном примере для простоты не показан блок try для метода wait().



И наконец, рассмотрим диаграмму состояний потока. Состояние `new` начинается, когда поток создан, но еще не запущен. После вызова метода `start()` поток переходит в состояние готовности к выполнению, из которого диспетчер периодически переводит поток в состояние реального исполнения. Java не различает состояние готовности и выполнения. Количество потоков, которые одновременно могут исполняться, зависит от числа процессоров (ядер) и настроек JVM. Во время исполнения поток может уснуть в результате вызова `sleep()`, уйти ждать после вызова `wait()`, а также заблокироваться, ожидая, пока другой поток покинет синхронизированный блок. Из всех этих состояний поток переходит в состояние готовности, чтобы далее диспетчер перевел его в исполняемое состояние, возобновив работу метода `run()`. После завершения метода `run()` поток переходит в состояние `terminated`.

# Многопоточность java.concurrent.\*

- Взаимная блокировка (deadlock)

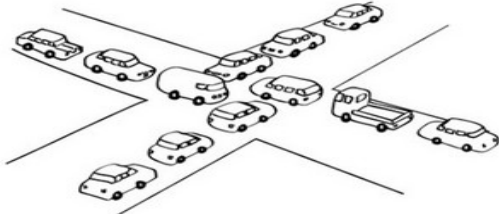


- Обедающие философы



При работе многопоточных программ могут возникать различные проблемы, связанные с организацией взаимодействия потоков. Одной из проблем является взаимная блокировка (deadlock). Такая ситуация возникает, когда потоку для работы требуется одновременно получить доступ к нескольким ресурсам. Один поток блокирует первый ресурс, и ждет освобождения второго ресурса для продолжения работы. А второй ресурс в это время заблокирован другим потоком, который ждет освобождения первого ресурса. В итоге ни один из потоков не может продолжить работу. Пример этой проблемы — пробка на перекрестке. Еще можно рассмотреть ее на примере обедающих философов. За столом сидят 5 философов, на столе 5 тарелок с едой и 5 вилок, по одной справа и слева. После размышления над проблемами мироздания философ берет правую вилку, если она свободна, затем ждет освобождения левой вилки, берет ее и обедает. Потом кладет вилки на стол. В какой-то момент времени может оказаться, что все философы держат правую вилку и ждут освобождения левой вилки, но никто из них не

- Взаимная блокировка (deadlock)



- Ресурсное голодание (starvation)



- Обедаящие философы



- Зацикливание (livelock)



Проблема решается уточнением алгоритма, по которому действуют философы. Например, можно пронумеровать вилки, и запретить брать вилку с меньшим номером, если вилка с большим номером занята. Также можно заставить философов освобождать первую вилку, если у них не получилось взять вторую. Но, при не совсем корректном решении проблемы взаимных блокировок можно встретиться с другой проблемой — livelock или зацикливание. Опять же, может получиться так, что философ, не сумев заполучить левую вилку, кладет правую, но то же самое делают и другие философы. В итоге они будут одновременно брать правую вилку и класть ее обратно. Аналогичная проблема можно наблюдать, когда 2 человека пытаются разойтись в узком коридоре, либо когда двое пытаются пропустить друг друга войти первым в дверь. С одной стороны полной блокировки нет, с другой стороны, не делается никакой полезной работы. Еще одной проблемой может стать ситуация, когда одни потоки постоянно получают доступ к ресурсам, а другие — нет, в итоге часть потоков начинает испытывать ресурсное



## Неизменяемые объекты (Immutable Objects)

- **Неизменяемый объект** — нет проблем многопоточности
  - Убрать сеттеры
  - Все поля — `private final`
  - Все методы — `final`
  - Не сохранять ссылки на изменяемые объекты — сохранять копии объектов

46

Одним из вариантов решения проблем многопоточности является использование неизменяемых объектов (Immutable objects). Основным принципом является то, что объекты, к которым одновременно может обращаться несколько потоков, не могут изменяться после создания. Для этого надо убрать все методы, меняющие переменные объекта, все поля объявляются финальными, все методы тоже, чтобы подкласс не мог переопределить их и изменить значения полей. И при передаче в качестве параметра ссылочного объекта, сохранять его копию, и дальше работать только с ней.



- `java.util.concurrent`
  - интерфейсы `Executor`, `Callable`, `Future`
  - классы `ThreadPoolExecutor`, `ForkJoinPool`
  - классы-синхронизаторы
  - интерфейсы `BlockingQueue`, `TransferQueue`
  - коллекции `ConcurrentX` и `CopyOnWriteArrayX`
- `java.util.concurrent.locks`
  - интерфейсы `Lock`, `Condition`
- `java.util.concurrent.atomic`
  - `AtomicInteger`, `AtomicLong`, `AtomicReference`

Но не всегда можно обойти только неизменяемыми объектами. Есть много алгоритмов решения проблем многопоточности, которые реализованы в классах, входящих в пакет `java.util.concurrent` и его подпакеты. Это интерфейсы и классы, реализующие исполнителей (`Executors`), которые позволяют запускать задачи на параллельное исполнение, пулы потоков, которые дают возможность использовать потоки повторно, синхронизаторы, которые предоставляют готовые способы синхронизировать работу потоков, интерфейсы и классы для очередей задач и потокобезопасных коллекций. Также рассмотрим интерфейсы `Lock` и `Condition` — более удобные способы обеспечить механизмы блокировки доступа, а также ожидания-нотификации. И в пакет `java.util.concurrent.atomic` входят классы для работы с атомарными переменными, реализующие атомарные операции для работы с ними.

- interface **Executor**
- Thread — абстракция потока
- Executor — абстракция исполнителя
- void **execute**(Runnable task)
  - выполнить задачу в какой-то момент времени в будущем
  - возможно сразу
  - возможно в новом потоке

```
Executor executor = ...;
executor.execute(task1);
executor.execute(task2);

(new Thread(task1)).start();
(new Thread(task2)).start();
```

Базовым интерфейсом исполнителей является Executor. Он позволяет запустить дополнительную задачу, как это делает класс Thread. Но если класс Thread является абстракцией потока, то Executor — это абстракция исполнителя задачи. Executor содержит метод execute, который принимает в качестве аргумента объект, реализующий интерфейс Runnable. Метод execute запускает задачу на исполнение в какой-то момент времени в будущем. Задача может запуститься и сразу же после вызова метода execute — это тоже момент в будущем. Как именно запустится задача, зависит от реализации. Она может запуститься в отдельном потоке, но может и в том же. В отличие от класса Thread, при использовании которого для запуска нового потока нужно создавать новый объект класса Thread, при работе с исполнителем не нужно создавать отдельного исполнителя для каждой задачи, один объект исполнителя способен запустить множество задач.



- interface **ExecutorService** extends `Executor`
  - `Future<T> submit(Callable<T> task)`
  - `List<Future<T>> invokeAll(Collection<Callable<T>> tasks)`
  - `void shutdown()`
- interface **Callable<T>**
  - `T call()`
- interface **Future<T>**
  - `T get()`
  - `boolean isDone()`
  - `boolean cancel()`
- interface **ScheduledExecutorService** extends `ExecutorService`
  - `ScheduledFuture schedule(task, delay, unit)`
  - `ScheduledFuture scheduleAtFixedRate(task, initial, period, unit)`
  - `ScheduledFuture scheduleAtFixedDelay(task, initial, delay, unit)`

Интерфейс `ExecutorService` расширяет интерфейс `Executor` добавлением методов: `submit`, который позволяет запланировать к выполнению не только объект `Runnable`, но и объект типа `Callable<T>`. `Callable` — это интерфейс с методом `call()`, который в отличие от `run()` возвращает значение типа `T`.

Метод `submit`, принимая `Callable`, возвращает объект типа `Future` — интерфейса, представляющего будущий результат, который в том числе может быть получен асинхронно. В интерфейсе `Future` есть метод `get()`, ожидающий завершения задачи и возвращающий ее результат, метод `isDone()`, проверяющий готовность результата, и метод `cancel()`, который отменяет выполнение задачи.

Интерфейс `ExecutorService` также определяет методы `invokeAll()`, принимающий коллекцию задач для исполнения и возвращающий список результатов, и метод `shutdown()`, завершающий работу исполняемой службы.

Интерфейс `ScheduledExecutorService` расширяет интерфейс `ExecutorService`, добавляя метод `schedule`, который позволяет запустить задачу с задержкой, а

```
String search(String f, String text) { }  
s = "find";  
text = "very long text";  
ExecutorService service = ...;  
Callable<String> task = () -> search(s, text);  
Future<String> future = service.submit(task);  
// while(!future.isDone()) { ... }  
String searchResult = future.get();
```

Пример работы службы исполнителей: допустим, что есть долго выполняющийся метод поиска строки в тексте. Каким-то образом получаем объект типа `ExecutorService`, создаем объект типа `Callable`, в методе `call()` которого задаем необходимое действие, а именно вызов метода `search()`. И запускаем задачу на выполнение методом `submit`, который вернет объект типа `Future`, в который будет помещен результат выполнения поиска.

Дальше можно либо дождаться результата, вызвав метод `get()`, но при этом текущий поток будет заблокирован вызовом метода `get()`, либо начать выполнять другую задачу, периодически проверяя готовность результата поиска вызовом метода `isDone()`, либо создать еще один поток, который будет ждать готовности результата, и полностью отдать ему обработку полученных данных.

- Создание потока требует ресурсов и времени
- Потоки в пуле повторно используются по мере освобождения
- Постепенная деградация при увеличении нагрузки

Так как создание нового потока требует ресурсов и времени, создавать для выполнения новой задачи отдельный поток, и завершать его после окончания задачи слишком накладно при большом количестве потоков. Имеет смысл постоянно держать некоторое количество готовых потоков в пуле, а после окончания задачи возвращать поток в пул, где он может дожждаться следующей задачи. При этом, если ограничить число одновременно работающих потоков, то система получает полезное свойство постепенной деградации. Если на каждый отдельный запрос система создает поток, и их число не ограничено, то в какой-то момент времени на создание нового потока может не хватить ресурсов, и система перестанет отвечать на запросы. Если количество потоков ограничено, то при возрастании количества запросов система начнет отвечать реже, будет повышаться время ожидания, но полной внезапной остановки работы не произойдет.

- class ThreadPoolExecutor implements ExecutorService
- class ScheduledThreadPoolExecutor implements ScheduledThreadPoolExecutor
- класс Executors — статические методы
  - ExecutorService newSingleThreadExecutor()
  - ScheduledExecutorService newSingleThreadScheduledExecutor()
  - ExecutorService newFixedThreadPool()
  - ExecutorService newCachedThreadPool()
  - ScheduledExecutorService newScheduledThreadPool()

Класс ThreadPoolExecutor является реализацией службы исполнителей на основе пула потоков. Имеется также класс ScheduledThreadPoolExecutor, который позволяет задавать время запуска задач. Также для удобства можно использовать класс Executors, содержащий статические методы для создания пулов потоков с различными свойствами, например, исполнителя задач в одиночном потоке без задержек, либо по расписанию, пул потоков фиксированного размера, либо пул кэширующихся потоков, которые создаются по мере необходимости, но удаляются, если не были использованы в течение определенного времени (60 сек)

- Реализация параллельного программирования
- Стратегия «разделяй и властвуй» (divide and conquer)
- Алгоритм «перехват работы» (work stealing)

```

if (size(task) = small) {
    return result
} else {
    foreach (subtask[i] : task) {
        result[i] = execute(subtask[i])
    }
    return merge(result)
}

```

В 7 версии Java появился фреймворк поддержки параллельного программирования — Fork/Join framework, который оптимизирован для выполнения на многоядерных процессорах. Есть класс задач, разделяемых на отдельные подзадачи, которые можно выполнить параллельно. Можно сказать, что фреймворк реализует стратегию «разделяй и властвуй» применительно к параллельному выполнению. Кроме того, применяется схема с «перехватом работы», когда у каждого исполнителя есть двусторонняя очередь задач, и когда исполнитель заканчивает выполнять все свои задачи, он начинает выполнение задач других исполнителей, забирая их из конца очереди.

Общий алгоритм работы фреймворка Fork/Join выглядит так:

Если полученная задача достаточно мала, то выполнить ее и вернуть результат. Иначе — поделить задачу на подзадачи, отдать их на выполнение, получить результаты подзадач, объединить их в общий результат и вернуть его.

- class ForkJoinPool
  - ForkJoinPool.commonPool()
- class ForkJoinTask
  - ForkJoinTask<V> fork()
  - V join()
  - V invoke(ForkJoinTask<V>)
  - invokeAll(ForkJoinTask... tasks)
  - class RecursiveAction extends ForkJoinTask
    - ◊ abstract void compute()
  - class RecursiveTask extends ForkJoinTask
    - abstract V compute()

Основными классами для работы этого фреймворка являются `ForkJoinPool`, представляющий пул потоков-исполнителей. Объект этого класса можно получить например с помощью статического метода `commonPool()`. Для представления задачи предназначен класс `ForkJoinTask`, основными методами которого являются `fork()`, который возвращает подзадачу, и `join()`, предназначенный для ожидания результата. Однако, для еще более удобной работы можно использовать подклассы — `RecursiveTask` и `RecursiveAction`, один для задач с возвращаемым результатом, другой для задач без него. В этих двух классах необходимо переопределить абстрактный метод `compute()`, в котором и реализовать алгоритм выполнения маленькой части задачи, либо деления ее на части с передачей их на исполнение другим.



## Пример

```
public class DoubleTask {
    final int[] array;
    final int lo, hi;
    final static int SMALL_SIZE = 10;

    DoubleTask(int[] array, int lo, int hi) {
        this.array = array; this.lo = lo, this.hi = hi;
    }

    protected void compute() {
        if ((hi - lo) < SMALL_SIZE) {
            for (int i = lo; i < hi; i++) array[i] *= 2;
        } else {
            int mid = (lo + hi) / 2;
            DoubleTask dt1 = new DoubleTask(array, lo, mid);
            DoubleTask dt2 = new DoubleTask(array, mid, hi);
            invokeAll(dt1, dt2);
        }
    }
}
```

55

В данном примере имеется класс `DoubleTask`, который используется для параллельного удвоения элементов массива. Поля класса — это массив, а также нижний и верхний индексы диапазона, в котором удваиваются элементы.

В методе `compute()` сначала проверяется, сколько элементов нам нужно удвоить. Если это число меньше заданного размера, а именно 10, то это будет сделано сразу же. Если же разница между индексами больше, то обработка первой половины массива становится первой подзадачей, обработка второй половины — второй подзадачей. Далее эти подзадачи запускаются на выполнение методом `invokeAll()`.



## Пример

```
int[] arr = {0, ... , 33554431};  
DoubleTask dt = new DoubleTask(arr, 0, arr.length - 1);  
ForkJoinPool pool = ForkJoinPool.commonPool();  
pool.invoke(dt);
```

56

Пусть имеется массив, который нужно обработать. Создадим задачу `DoubleTask`, передав в качестве параметров ссылку на массив, индексы начального и конечного элемента для обработки данной задачей — 0 и индекс последнего элемента. Создаем пул задач типа `Fork/Join`, и запускаем исходную задачу на выполнение. После окончания метода `invoke` все элементы массива будут удвоены.





- interface Lock — аналог synchronized
  - lock()
  - unlock()
  - tryLock()
  - tryLock(long time)
  - lockInterruptibly()
  - Condition newCondition()
- interface Condition — аналог wait-notify
  - await()
  - signal()
  - signalAll()

Синхронизация действий потоков с помощью блокировок и условий реализована в Java на низком уровне — каждый объект имеет встроенную блокировку, которая захватывается потоком при входе в синхронизированный блок, а также методы `wait` и `notify`, позволяющие входить в состояние ожидания и посылать извещения об изменении некоторого условия.

Однако, часто требуется более гибкие методы решения той же задачи с расширенными возможностями. Для этого в стандартную библиотеку были добавлены интерфейсы `Lock` и `Condition`. `Lock` — это блокировка, аналогичная встроенной блокировке при использовании синхронизированных методов и блоков, методы `lock()` и `unlock()` аналогичны захвату и освобождению блокировки, методы `tryLock` и `lockInterruptibly` позволяют попытаться захватить блокировку с возможностью прервать операцию. А интерфейс `Condition` предоставляет методы `await`, `signal` и `signalAll`, аналогичные `wait` и `notify`, но без привязки к одному конкретному объекту.



- class ReentrantLock implements Lock

```
int value;
Lock lock = new ReentrantLock();

void put(int i) {
    lock.lock();
    try { value = i; }
    finally { lock.unlock(); }
}

int get() {
    lock.lock();
    try { return value; }
    finally { lock.unlock(); }
}
```

В пакете `java.util.concurrent.locks` есть класс `ReentrantLock`, реализующий базовое поведение блокировки, аналогичное модификаторам `synchronized`. Пример показывает реализацию методов `put` и `get`, в которых с помощью метода `lock()` захватывается блокировка. Рекомендуется сразу же после вызова `lock` размещать блок `try` с участком кода, выполнение которого должно быть синхронизировано между потоками, а в блоке `finally` вызывать метод `unlock()`, чтобы обеспечить освобождение блокировки даже в случае возникновения исключений.



- class ReentrantLock implements Lock

```
boolean right = rightFork.tryLock();
try {
    if (right) {
        boolean left = leftFork.tryLock();
        try {
            if (left) {
                eat();
            }
        } finally { leftFork.unlock(); }
    }
} finally { rightFork.unlock(); }
```

В этом примере показан один из вариантов поведения обедающих философов. Сначала делаем попытку взять правую вилку, если она удалась, то пробуем взять левую.

Если и это получилось, то вызывается метод eat() и философ обедает. Если какую-то из вилок взять не удалось, то обе вилки освобождаются.



- interface ReadWriteLock
  - Lock readLock()
    - ◊ возвращает Lock для операций чтения (множественный доступ)
  - Lock writeLock()
    - ◊ возвращает Lock для операций записи (блокирующий доступ)
- class ReentrantReadWriteLock

Кроме одиночной блокировки, можно использовать связанные блокировки, реализующие интерфейс `ReadWriteLock` и класс `ReentrantReadWriteLock`, реализующий этот интерфейс.

В классе содержится пара блокировок — на чтение и на запись, при этом при захваченной блокировке на чтение, другие потоки также могут захватывать эту блокировку, так что множественный доступ к объекту на чтение разрешен. Если какой-то поток пытается захватить блокировку на запись, то он ждет освобождения всех блокировок на чтение. Во время ожидания новые блокировки на чтение не захватываются. При получении блокировки на запись, возможность работы имеет только один поток. Такой способ позволяет повысить производительность, так как общий доступ блокируется только во время операций записи, а чтение может происходить из множества потоков одновременно.



## Интерфейс Condition

```
Lock lock = new ReentrantLock();
Condition notFull = lock.newCondition();
Condition notEmpty = lock.newCondition();
int[] values = new int[100]; int count;

public void put(int i) {
    lock.lock();
    try {
        while(count == values.length) { notFull.await(); }
        values[count++] = i;
        notEmpty.signal();
    } finally { lock.unlock(); }
}

public int get() {
    lock.lock();
    try {
        while(count == 0) { notEmpty.await(); }
        notFull.signal();
        return values[--count];
    } finally { lock.unlock(); }
}
```

61

Данный пример показывает использование интерфейса Condition. Есть массив на 100 значений, также имеется блокировка и 2 условия, связанные с этой блокировкой — NotEmpty (массив не пуст) и NotFull (массив не заполнен).

В методе put происходит захват блокировки, затем проверяется, достигли ли мы конца массива, и, если да, то поток, выполняющий метод put, уходит в состояние ожидания по условию notFull, дожидаясь получения сигнала по тому же условию. Если же в массиве еще есть место, то туда добавляется значение и вызывается сигнал по условию notEmpty, извещающий потоки, ожидающие появления данных. В блоке finally блокировка снимается.

В методе get также захватывается блокировка, происходит проверка на отсутствие данных, и, если их нет, то поток начинает ожидание по условию notEmpty, если же данные есть, то происходит извещение потоков, ожидающих возможность записать данные и возврат нужного значения, после чего блокировка освобождается.

- Semaphore — синхронизатор со счетчиком разрешений. Мьютекс — простейший бинарный семафор.
  - методы `acquire()`, `release()`
- CountdownLatch — синхронизатор с обратным счетчиком ожидает определенное количество событий
  - методы — `await()`, `countDown()`
- CyclicBarrier — синхронизатор, который синхронизирует определенное количество потоков
  - метод `await()`, `reset()`
- Phaser — синхронизатор, ожидающий завершения действий, состоящих из этапов или фаз
  - методы `register()`, `arrive()`, `arriveAndAwaitAdvance()`, `arriveAndDeregister()`
- Exchanger<V> — синхронизатор обмена данными между двумя потоками
  - метод `V exchange(V data)`

Кроме простейшего механизма синхронизации, в пакете `java.util.concurrent` есть классы-синхронизаторы. Класс `Semaphore` позволяет организовать блокировку со счетчиком. Участок кода может выполняться определенным количеством потоков, как только это число достигается, остальные потоки ждут, пока хотя бы один не завершит выполнение этого участка. `CountDownLatch` позволяет нескольким потокам войти в состояние ожидания, а после вызова метода `countDown` определенное число раз, вывести из ожидания сразу все. Класс `CyclicBarrier` позволяет нескольким потокам ждать друг друга. Как только количество ожидающих потоков достигнет заданного значения, они все продолжат работу. Класс `Phaser` делает то же самое, но с возможностью разбить задачу на этапы, и для каждого этапа задать набор потоков, которые должны будут ждать или не ждать, пока другие не завершат данный этап. И класс `Exchanger` позволяет организовать обмен данными между потоками. Как только один поток вызывает метод `exchange`, он ждет, когда другой поток вызовет его же, после чего каждый из них получит переданные другим потоком данные.



- Атомарная операция — операция, выполняющаяся без промежуточных состояний.
- Атомарные операции не вызывают состояние гонок
- операции чтения-записи ссылок и примитивных типов (кроме long и double) — атомарные
- Пакет java.util.concurrent.atomic — атомарные типы

```
class A {  
    AtomicInteger counter = new AtomicInteger(0);  
    public void up() { counter.incrementAndGet(); }  
    public void down() { counter.decrementAndGet(); }  
}
```

Состояние гонок возникает при доступе к общим переменным из разных потоков. С ним можно справиться с помощью блокирования одновременного доступа. Однако, исходная причина гонок — это неатомарность операций доступа к данным.

Атомарная операция выполняется без промежуточных состояний, то есть не может возникнуть ситуации, когда какой-то поток видит незавершенный результат операции, выполняемой другим потоком. Операции записи и чтения примитивных типов атомарны для всех типов, кроме long и double. Другие операции, например, инкремент и декремент, сами по себе не являются атомарными. Для обеспечения атомарности можно использовать классы из пакета

java.util.concurrent.atomic. С помощью этих классов можно обеспечить множественный доступ к общим переменным из разных потоков. Например, вместо неатомарных операций инкремента и декремента с переменной типа int можно использовать методы incrementAndGet и decrementAndGet для объекта класса AtomicInteger.



- AtomicInteger
  - get – аналог чтения volatile переменной
  - set – аналог записи volatile переменной
  - compareAndSet(int expect, int update)

```
int operation(int arg) { }  
int operationAndGet(AtomicInteger x) {  
    int prev, next;  
    do {  
        prev = x.get();  
        next = operation(prev);  
    } while (!x.compareAndSet(prev, next));  
    return next;  
}
```

Методы `get` и `set` классов, представляющих атомарные типы данных, выполняют действия, аналогичные получению и записи значения переменных с модификатором `volatile`. Также все атомарные типы реализуют метод `compareAndSet`, который сравнивает ожидаемое значение с имеющимся, и в случае их совпадения обновляет значение.

Допустим, что надо атомарно выполнить некоторую операцию над аргументом типа `int`, реализованную в методе `operation()`.

Тогда мы можем написать метод `operationAndGet`, в котором в цикле сохраняем старое значение переменной, получаем результат операции, и если старое значение переменной совпадает с текущим значением, то сохраняем новое. При наличии несовпадения повторяем цикл.

При этом операция сравнения и установки нового значения выполняется атомарно, то есть без промежуточных состояний.



- **BlockingQueue / BlockingDeque**
  - `put(E)` — поместить элемент в очередь, ожидая, пока появится свободное место
  - `E take()` — получить элемент из очереди, ожидая, пока в ней появятся элементы
  - для `Deque` — `putFirst`, `putLast`, `takeFirst`, `takeLast`
  - `ArrayBlockingQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`
  - `DelayQueue` — элементы доступны после задержки
  - `SynchronousQueue` — синхронное добавление-получение
- **TransferQueue extends BlockingQueue**
  - `transfer(E)` — дождаться получения элемента
  - `LinkedTransferQueue`

Одним из способов организовать многопоточную обработку является использование очередей. Они используются как для управления задачами и потоками — задачи помещаются в очередь, из которой потом их забирают исполнители. Также можно использовать очереди для передачи данных. Одни потоки помещают данные в очередь, другие потоки забирают их оттуда. В пакете `java.util.concurrent` имеется несколько реализаций подобных очередей. Интерфейс `BlockingQueue` имеет методы `put()` и `take()`, которые блокируются, если не удастся поместить элемент в очередь (например, если она заполнена), либо прочитать элемент из очереди (например, если в ней нет данных). Есть несколько реализаций этого интерфейса — `ArrayBlockingQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`. `DelayQueue` — это очередь, элементы из которой можно получить только после истечения некоторого количества времени после того, как они попали в очередь. `SynchronousQueue` — это очередь, где элементы не хранятся, а поместить туда элемент можно только если в это время другой поток ждем



## Конкурентные коллекции

- Синхронизированные коллекции — используют блокировки
- Конкурентные коллекции — оптимизированные алгоритмы для многопоточной работы
- `ConcurrentMap` / `ConcurrentNavigableMap`
  - атомарные операции — методы `putIfAbsent`, `remove`, `replace`
  - `ConcurrentHashMap`,
  - `ConcurrentSkipListMap`, `ConcurrentSkipListSet`
- `ConcurrentLinkedQueue`
  - потокобезопасная очередь
- `CopyOnWriteArrayList` / `CopyOnWriteArraySet`
  - операции, изменяющие коллекцию, создают новую копию.
  - операции чтения, а также итераторы продолжают работать со старой копией.

66

В классе `Collections` есть методы `synchronizedCollection()` и подобные, которые возвращают синхронизированные коллекции с блокировкой методов добавления и получения элементов. Блокировки делают использование коллекции потокобезопасным, но не обеспечивают высокую производительность. В пакете `java.util.concurrent` есть классы, реализующие конкурентные коллекции, оптимизированные для многопоточного доступа. `ConcurrentMap` обеспечивает атомарные операции записи, удаления и замены элементов, реализации этого интерфейса — `ConcurrentHashMap` и `ConcurrentSkipListMap` (на основе списка с пропусками). `ConcurrentLinkedQueue` — потокобезопасная очередь. И коллекции типа `CopyOnWrite` — это список и множество, реализуют алгоритм изменения коллекции, когда при записи создается новая копия коллекции, при этом во время записи другие потоки имеют доступ на чтение к сохраненному старому варианту, а сразу же после записи ссылка переключается на новую коллекцию.



- `java.nio` — асинхронные каналы — `Future<V>`
- `java.util.stream` — `Splitterator`, `parallelStream()`

Классы и методы, предназначенные для работы многопоточных приложений, используются и в других модулях библиотеки. Например, объекты типа `Future` возвращаются при работе с каналами ввода-вывода в асинхронном режиме. Алгоритмы работы, аналогичные фреймворку `Fork/Join` используются в потоках данных (`Stream API`) для распараллеливания обработки, использования сплитераторов и параллельной сортировки массивов в классе `Arrays`.