

MULTITHREADING

- закон амдала
- реализация многопоточности в java
- создание потоков
- состояния потока
- sleep и yield
- приоритеты потоков
- демоны
- ожидание завершения потока
- обработка исключений

Большинство проблем решается с помощью последовательных программ.

Однако в некоторых случаях бывает удобно и более быстро выполнять некоторые часть программы параллельно.

Например – web сервер, который может обрабатывать несколько запросов от пользователей параллельно.

Если мы хотим увеличить скорость работы программы – разбиваем программу на куски выполняем каждый кусок на отдельном процессоре.

Если у нас многоядерная и многопроцессорная машина – такое разбиение существенно увеличит пропускную способность.

Если мы хотим увеличить скорость работы программы – разбиваем программу на куски выполняем каждый кусок на отдельном процессоре.

Если у нас многоядерная и многопроцессорная машина – такое разбиение существенно увеличит пропускную способность.

Но не только – **на одноядерной машине производительность так же может быть увеличена.**

Если мы хотим увеличить скорость работы программы – разбиваем программу на куски выполняем каждый кусок на отдельном процессоре.

Если у нас многоядерная и многопроцессорная машина – такое разбиение существенно увеличит пропускную способность.

Но не только – **на одноядерной машине производительность так же может быть увеличена.**

Достигается когда поток блокируется на I/O операциях и процессор может в это время выполнять код другого потока, не блокируя программу в целом.

Возникает проблема взаимодействия между потоками в рамках работы с ОБЩИМИ данными.

Проблема синхронизации потоков.

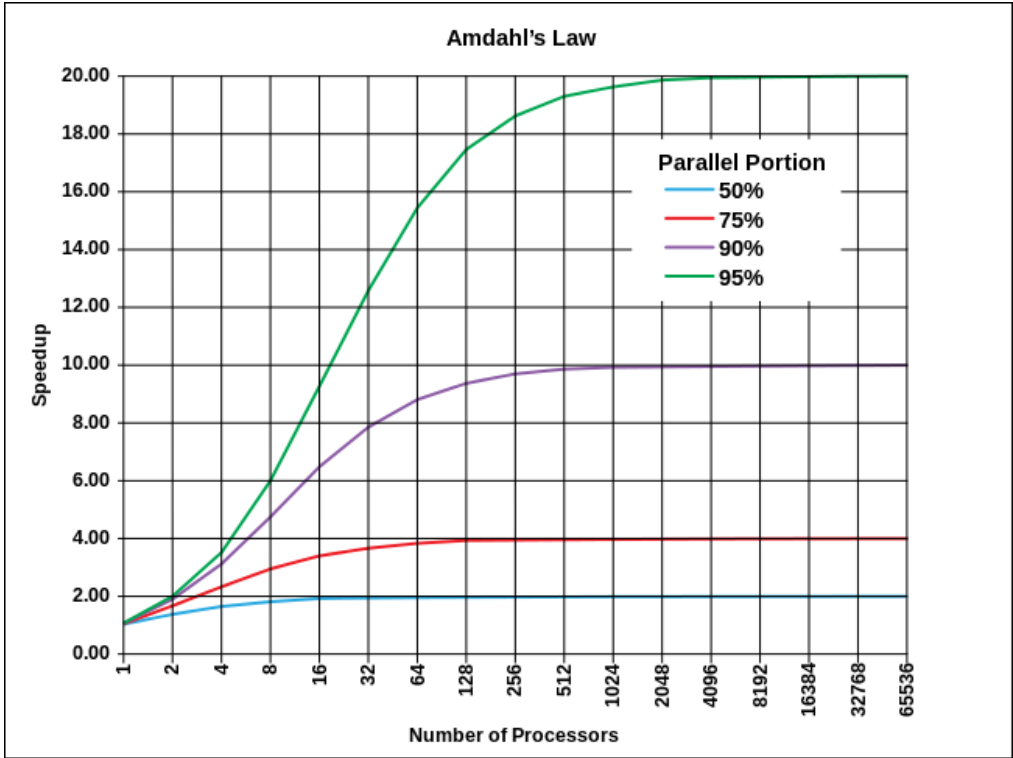
В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента.

$$S_p = \frac{1}{\alpha + \frac{1 - \alpha}{p}}$$

α – доля последовательных вычислений

P – кол-во процессоров

S - ускорение



Некоторые языки спроектированы таким образом, чтобы изолировать конкурентные задачи друг от друга.

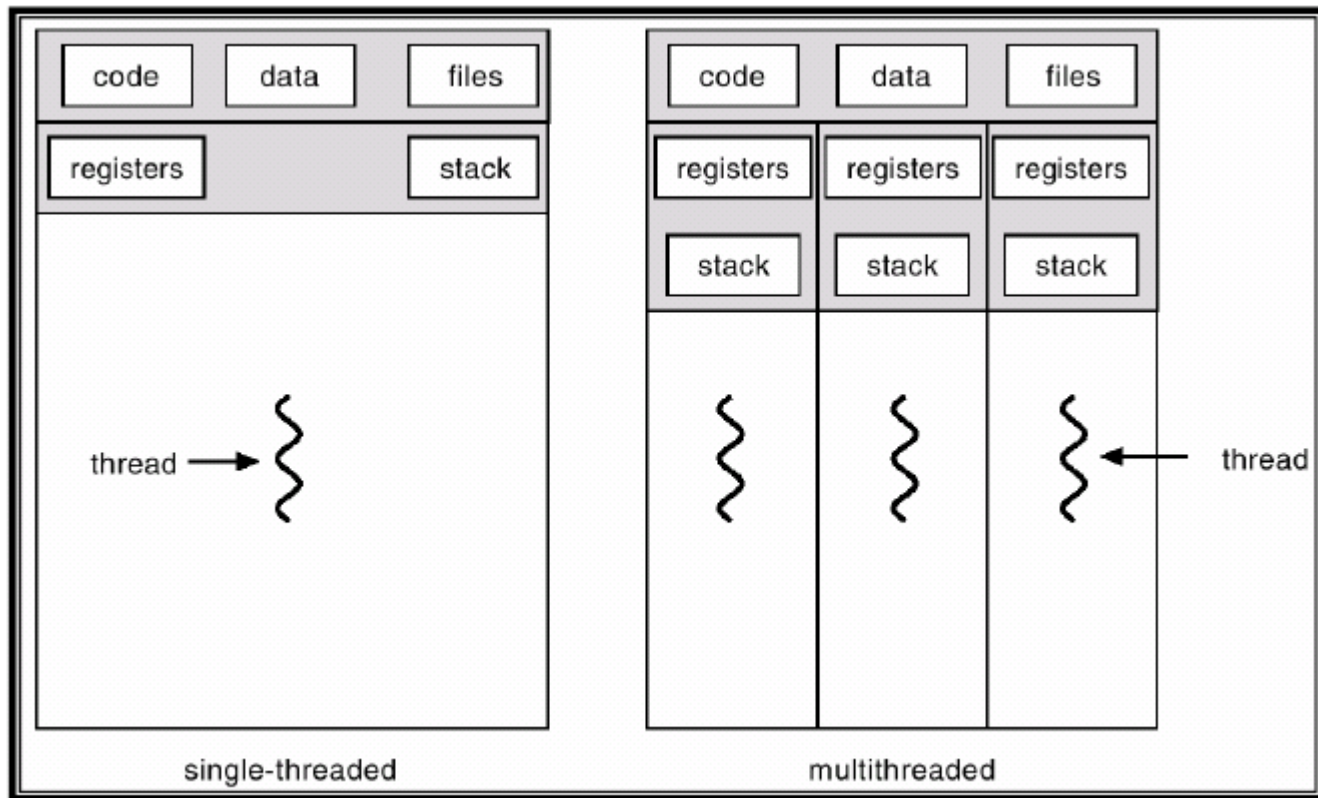
Например **ERLANG**.

Он также поддерживает безопасные механизмы взаимодействия между параллельными задачами.

Java со своего рождения так же имеет встроенную поддержку построение многопоточного конкурентного кода.

Придерживается более традиционного подхода решения вопроса многозадачности – создание **ПОТОКОВ** в рамках одного процесса.

Поток в java мапится на поток операционной системы, где запущена программа



Задача (task) – некоторая работа которая может быть выполнена.

Поток (thread) – механизм который может выполнить задачу.

Определить неделимую задачу для выполнения можно с помощью реализации интерфейса Runnable:

```
public class SomeTask implements Runnable {  
    public void run() {  
        for (int i = 0; i < 3; i++) {  
            System.out.print(format("#%d(%s)", i,  
                Thread.currentThread().getName()));  
        }  
    }  
}
```

Запустить можно задачу в рамках main потока можно так:

```
public static void main(String[] args) {  
    SomeTask launch = new SomeTask();  
    launch.run();  
}
```

Для запуска задачи в отдельном потоке можно использовать класс Thread:

```
public static void main(String[] args) {  
    Thread t = new Thread(new SomeTask());  
    t.start();  
}
```


Следовательно можно попробовать создать несколько параллельных потоков

```
public static void main(String[] args) {  
    for (int i = 0; i < 10; ++i) {  
        new Thread(new SomeTask()).start();  
    }  
    System.out.println("Waiting end of some task.");  
}
```

Вывод программы?

Вывод программы:

Waiting end of some task.

```
#0(Thread-4)#1(Thread-4)#2(Thread-4)#0(Thread-5)#0(Thread-6)#0(Thread-0)#1(Thread-6)#0(Thread-8)#1(Thread-8)#2(Thread-8)#0(Thread-7)#1(Thread-7)#2(Thread-7)#2(Thread-6)#0(Thread-9)#1(Thread-9)#2(Thread-9)#0(Thread-1)#1(Thread-1)#2(Thread-1)#1(Thread-0)#2(Thread-0)#1(Thread-5)#0(Thread-2)#2(Thread-5)#1(Thread-2)#2(Thread-2)#0(Thread-3)#1(Thread-3)#2(Thread-3)
```

Process finished with exit code 0

- переключение между потоками контролируется планировщиком ОС
- на многопроцессорной машине планировщик распределит потоки по процессорам

- переключение между потоками контролируется планировщиком ОС
- на многопроцессорной машине планировщик распределит потоки по процессорам
- алгоритм планировщика не детерминирован – следовательно вывод предыдущей программы так же будет отличаться от запуска к запуску
- каждый объект типа Thread регистрирует себя в определённом месте и garbage collector не может удалить этот объект до тех пор пока не завершится выполнение функции run()

Реализовать Runnable:

```
public class RunnableImpl implements Runnable {  
    @Override  
    public void run() {  
        // some task code here  
    }  
  
    public static void main(String[] args) {  
        new Thread(new RunnableImpl()).start();  
    }  
}
```

Унаследовать класс Thread:

```
public class SimpleThread extends Thread {  
  
    @Override  
    public void run() {  
        //some task code here  
    }  
  
    public static void main(String[] args) {  
        new SimpleThread().start();  
    }  
}
```

Через анонимный класс:

```
public class RunnableImpl {  
    public static void main(String[] args) {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                //some task code here  
            }  
        }).start();  
    }  
}
```

Замечание:

Хорошей практикой является отделение потока от бизнес логики.

Т.е. класс Runnable не должен содержать бизнес логикой, а должен быть просто инструментом для её запуска в отдельном потоке.

Важно:

- Создавать потоки руками можно только в простых примерах и тестах
- Более правильным решением является использование thread пулов
- Создавать и запускать поток в конструкторе нельзя – опасно

Любой поток может находиться в одном из 4 логических состояний:

1. **New** – состояние в момент создания, когда система выделяет ему ресурсы, а планировщик готовит его к планированию

Любой поток может находиться в одном из 4 логических состояний:

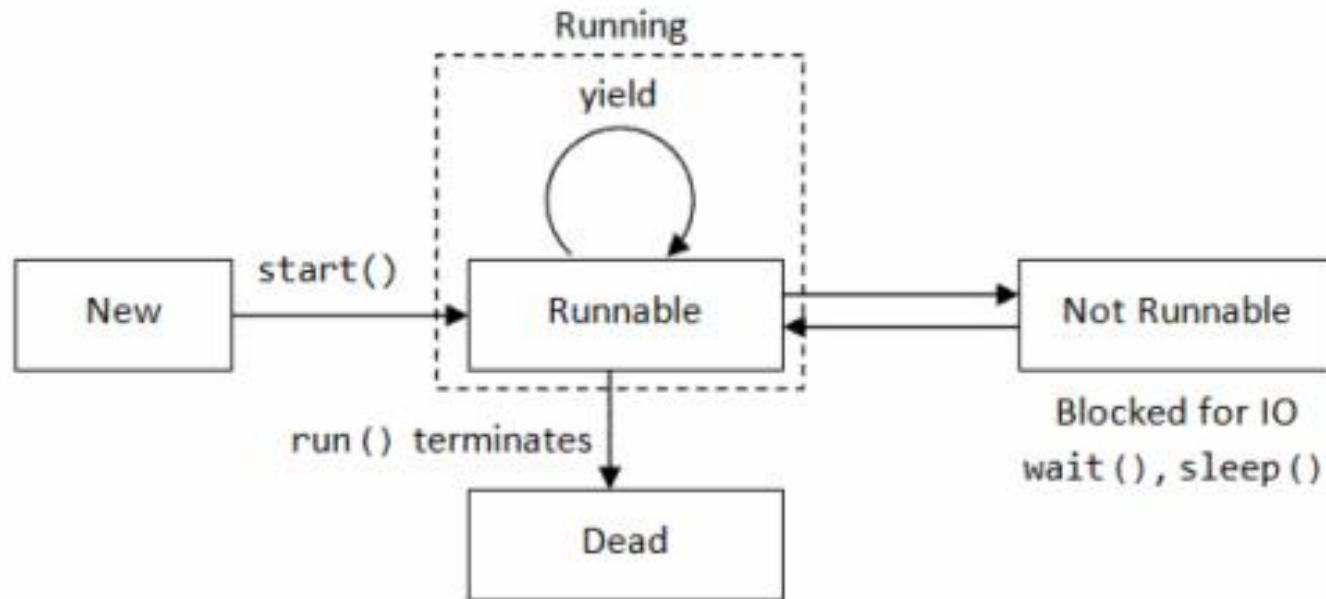
1. **New** – состояние в момент создания, когда система выделяет ему ресурсы, а планировщик готовит его к планированию
2. **Runnable** – состояние, когда поток может выполнять полезную работу, планировщик его планирует и выделяет кванты времени CPU

Любой поток может находиться в одном из 4 логических состояний:

1. **New** – состояние в момент создания, когда система выделяет ему ресурсы, а планировщик готовит его к планированию
2. **Runnable** – состояние, когда поток может выполнять полезную работу, планировщик его планирует и выделяет кванты времени CPU
3. **Blocked** – планировщик НЕ планирует его и не выделяет CPU, до возвращения в run

Любой поток может находиться в одном из 4 логических состояний:

1. **New** – состояние в момент создания, когда система выделяет ему ресурсы, а планировщик готовит его к планированию
2. **Runnable** – состояние, когда поток может выполнять полезную работу, планировщик его планирует и выделяет кванты времени CPU
3. **Blocked** – планировщик НЕ планирует его и не выделяет CPU, до возвращения в run
4. **Dead или terminate** – планировщик не планирует, задача завершена и НЕ может больше выполняться



Заставить поток «заснуть» на определённое время можно с помощью `Thread.sleep()`:

```
Thread.sleep(1000);
```

В *sleep* состоянии поток перестаёт планироваться ОС.

Более удобная форма:

```
TimeUnit.SECONDS.sleep(5);  
TimeUnit.MILLISECONDS.sleep(5);  
TimeUnit.MICROSECONDS.sleep(5);  
TimeUnit.MINUTES.sleep(5);
```


Есть возможность из потока дать «подсказку» планировщику о том что наш поток сделал достаточно и готов уступить квант времени другим потокам (используется редко).

```
public class Yield implements Runnable {  
    public void run() {  
        while (true) {  
            Task t = getNewTask();  
            t.execute();  
            Thread.yield();  
        }  
    }  
}
```

Есть возможность задать приоритет потоку через метод `setPriority` у класса.

Есть возможность задать приоритет потоку.

```
public class PriorityTest implements Runnable{
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(format("#%s (%d) ",
                Thread.currentThread().getName(),
                Thread.currentThread().getPriority()));
            Thread.yield();
        }
    }
}
```

Запустим:

```
public static void main(String[] args) {  
    for (int i = 0; i < 4; i++) {  
        Thread t = new Thread(new PriorityTest());  
        t.setPriority(i % 2 == 0 ? Thread.MAX_PRIORITY  
: Thread.MIN_PRIORITY);  
        t.start();  
    }  
}
```

Какой будет вывод у программы?

#Thread-2(10)

#Thread-2(10)

#Thread-0(10)

#Thread-0(10)

#Thread-0(10)

#Thread-2(10)

#Thread-1(1)

#Thread-3(1)

#Thread-1(1)

#Thread-3(1)

#Thread-1(1)

#Thread-3(1)

Process finished with exit code 0

Важно:

- Приоритет является мерой важности потока для планировщика.
- Планировщик будет стараться давать больший квант времени на исполнения потока с более высоким приоритетом.

Важно:

- Приоритет является мерой важности потока для планировщика.
- Планировщик будет стараться давать больший квант времени на исполнения потока с более высоким приоритетом.
- Так же это не значит что потоки с более низким приоритетом вообще не будут планироваться.
- Обычно манипулирование приоритетом потока является ошибочной практикой.

Важно:

- Приоритет является мерой важности потока для планировщика.
- Планировщик будет стараться давать больший квант времени на исполнения потока с более высоким приоритетом.
- Так же это не значит что потоки с более низким приоритетом вообще не будут планироваться.
- Обычно манипулирование приоритетом потока является ошибочной практикой.
- Задать приоритет можно с помощью метода `setPriority()`.
- Обычно стараются использовать 3 из 10 уровней приоритета: `MAX_PRIORITY`, `NORM_PRIORITY` и `MIN_PRIORITY`

Daemon threads предназначены для выполнения минорных задач до тех пор пока программа не закончит выполнение.

Программа закончит выполнение только тогда когда все НЕ демон потоки завершат свою работу.

Таким образом, завершая выполнение, программа убивает все демон потоки и завершается.

```
public class DaemonExample implements Runnable {  
    public void run() {  
        try {  
            // invoke long task logic (> one second)  
        } finally {  
            System.out.println("Thread has done.");  
        }  
    }  
}
```

```
public static void main(String[] args)
    throws InterruptedException {
    Thread t = new Thread(new DaemonExample());
    t.setDaemon(true);
    t.start();
    System.out.println("Daemon thread has been
started.");
}
```

Что выведет программа?

Daemon thread has been started.

Process finished with exit code 0

Код из finally блока не вызвался.

Важно:

- чтобы поток сделать демоном, нужно на нём перед стартом вызвать метод `setDaemon(true)`
- `finally` блок не будет вызван, т.к. поток прерывается «грубо», без освобождения занимаемых ресурсов

Так же:

- узнать является ли поток демоном можно вызвав метод `isDaemon()` на потоке

Любой поток может дожидаться завершения работы другого потока с помощью метода класса Thread - join().

```
Thread t = new Thread(new Join());  
t.start();  
t.join();
```

Важно:

- ожидающий поток блокируется и не планируется пока ожидаемый не завершит свою работу
- `join` может быть вызван с аргументом задающим кол-во мс которое необходимо ожидать
- состояние потока можно проверить с помощью `isAlive` метода

Какой вывод будет у программы?

```
public class ThrowExceptionSimpleCase implements
Runnable {
    public void run() {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        new Thread(new ThrowExceptionSimpleCase())
            .start();
    }
}
```


Exception in thread "Thread-0"

`java.lang.RuntimeException`

at

ThrowExceptionSimpleCase.run(ThrowExceptionSimpleCase.java:3)

at java.lang.Thread.run(Thread.java:745)

Process finished with exit code 0

Если исключение будет выброшено из функции `run()`, оно запишется на консоль, поток завершит работу.

Попробуем поймать исключение:

```
public static void main(String[] args) {  
    try {  
        new Thread(new ThrowExceptionSimpleCase())  
        .start();  
    } catch (RuntimeException ex) {  
        System.out.println("Exception has been catch");  
    }  
}
```

```
Exception in thread "Thread-0"  
java.lang.RuntimeException  
    at  
    ThrowExceptionSimpleCase.run(ThrowExceptionS  
impleCase.java:3)  
        at java.lang.Thread.run(Thread.java:745)  
  
Process finished with exit code 0
```

Ничего не поменялось!

Нельзя перехватить исключение из контекста другого потока.

Метод `Thread.UncaughtExceptionHandler()` позволяет задавать обработчик `uncaught` исключений потоку.

```
public static void main(String[] args) {  
    Thread t = new Thread(new  
    ThrowExceptionSimpleCase());  
    t.setUncaughtExceptionHandler((t1, e) ->  
        System.out.printf("Exception %s has been  
catch from thread %s",  
        e, t1.getName()));  
}
```

Вывод программы:

Exception java.lang.RuntimeException has been catch from thread
Thread-0

Process finished with exit code 0

Метод `Thread.setDefaultUncaughtExceptionHandler()` позволяет задавать обработчик `uncaught` исключений всем потокам по-умолчанию.

Следующие методы класса Thread устарели и являются опасными, т.к. не освобождают занятые ими ресурсы.

```
public final void stop()  
public final synchronized void stop(Throwable obj)  
public void destroy()  
public final void suspend()  
public final void resume()
```