

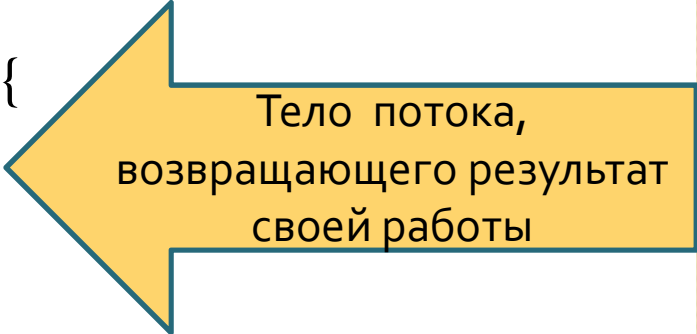
ВЫСОКОУРОВНЕВЫЙ ИНТЕРФЕЙС УПРАВЛЕНИЯ ПОТОКАМИ (Продолжение)

Потоки исполнения

III) Механизм асинхронного исполнения

- ❑ Интерфейс **Callable** (асинхронная работа с возвратом результата)

```
public interface Callable <T> {  
    T call() throws Exception;  
}
```



Тело потока,
возвращающего результат
своей работы

- ❑ Интерфейс **Future** (доступ к результату потока **Callable**)

```
public interface Future <T> {  
    T get();  
    T cancel(boolean mayInterrupt);  
    boolean isDone();  
    .....  
}
```

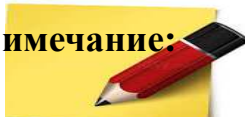
Потоки исполнения



Интерфейс **Callable** похож на **Runnable**;

- отличие состоит в том, что **Callable** предоставляет возможность вернуть результат в поток, который этот **Callable** запустил;
- был добавлен с версии Java 5, так как использует обобщения (generics) для определения типа возвращаемого значения;
- имеется возможность «выбрасывать» любые исключения (контролируемые и неконтролируемые), не оказывая влияния на другие выполняющиеся задачи.

Примечание:



Интерфейс Runnable вообще не допускал выбрасывания контролируемых исключений, а выброс неконтролируемого (RuntimeException) исключения приводил к остановке потока и всего приложения.

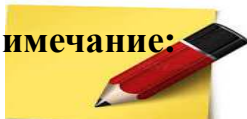
Потоки исполнения



Потоки (задачи) **Callable** возвращают результат, доступ к которому можно получить через объект типа **java.util.concurrent.Future**:

- метод *get()* - устанавливает блокировку пока ожидает завершения задачи **Callable**, чтобы получить результат, или исключения, если в процессе выполнения задачи произошла ошибка;
- метод *cancel()* – отменяет вычисление, если задача еще не стартовала и уже не будет запущена, и если вычисление уже идет, то оно прерывается;
- метод *isDone()* – проверяет закончено ли уже выполнение задачи.

Примечание:



Для удобства работы с потоками разного типа (**Callable** и **Runnable**) создан класс **FutureTask**, который представляет собой класс-обертку.

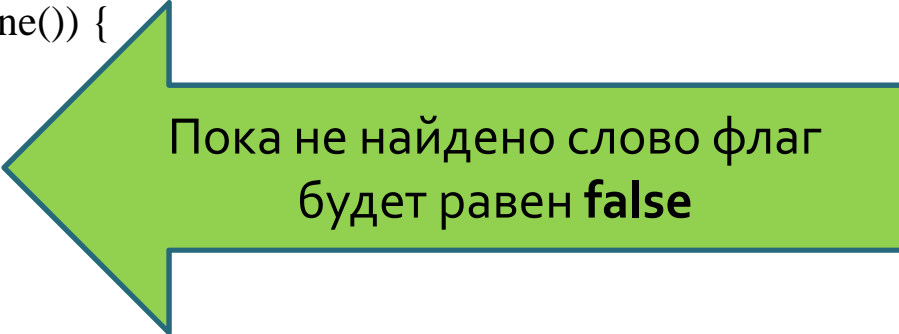
Потоки исполнения

- ❑ *Рассмотрим пример обработки файловой системы через потоки типа **Callable**:*
 - Необходимо подсчитать количество файлов в указанной директории и ее поддиректориях, которые содержат некоторое ключевое слово;
 - Есть класс-поток **CounterMath**, который определяет все элементы указанной директории и обрабатывает файлы, а для обработки поддиректории запускает новый поток (т.е. каждая директория обрабатывается в отдельном потоке);
 - Есть класс **Main**, который будет запускать задачу и собирать результат.

Потоки исполнения

Пример 17:

```
public class CounterMath implements Callable <Integer> {  
    private File dir;        private String word;  
    public CounterMath(File dir, String word) {  
        this.dir = dir;  
        this.word = word;  
    }  
    public boolean search(File ff) {  
        try (Scanner sc = new Scanner(new FileInputStream(ff))) {  
            boolean flag = false;  
            while ( !flag && sc.hasNextLine()) {  
                String str = sc.nextLine();  
                if (str.contains(word))  
                    flag = true;  
            }  
            return flag;  
        } catch (IOException e) {        return false;        }  
    }  
    // ...  
}
```



Пока не найдено слово флаг
будет равен **false**

Потоки исполнения

Продолжение примера 17:

```
public Integer call() {  
    int count = 0;  
    try {  
        File[] files = dir.listFiles();  
        ArrayList< Future<Integer> > result = new ArrayList<>();  
        for (File ff : files)  
            if (ff.isDirectory()) {  
                CounterMath counter = new CounterMath(ff, word);  
                FutureTask<Integer> task = new FutureTask <Integer> (counter);  
                result.add(task);  
                new Thread(task).start();  
            }  
            else if ( search(ff) )  
                count++;  
        for (Future<Integer> rez : result)  
            count += rez.get();  
    } catch (ExecutionException | InterruptedException e) {  
        e.printStackTrace();  
    }  
    return count;  
}  
}
```

Получить список элементов директории

Если элемент есть
директория, то создать
новый поток и связать его с
FutureTask для извлечения
результата

Если элемент есть файл, то осуществить поиск: если слово
присутствует, то увеличить счетчик

Собрать результаты из
всех поддиректорий

Потоки исполнения

Продолжение примера 17:

```
public class Main {  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter directiry -> ");  
        String dir = sc.next();  
        System.out.print("Enter keyword -> ");  
        String word = sc.next();  
        CounterMath counter = new CounterMath(new File(dir), word);  
        FutureTask<Integer> task = new FutureTask<Integer>(counter);  
        new Thread(task).start();  
        try {  
            System.out.println(task.get() + " files.");  
        } catch (ExecutionException | InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

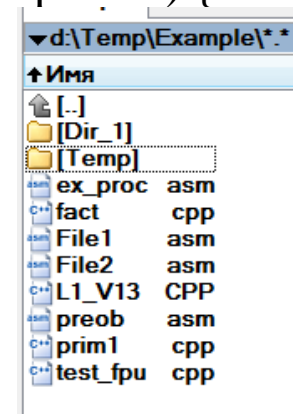
Создать поток

Упаковать поток в тип FutureTask

Ждать и
получить
результат

Вывод в консоли:

```
Enter directiry -> d:\Temp\Example\  
Enter keyword -> mov  
14 files.
```



Потоки исполнения

IV) Механизм управления задачами, основанный на пуле потоков



Причины для использования пула потоков:

- *выиграть некоторую производительность, когда потоки повторно используются*
 - ✓ системные издержки создания нового потока для каждой задачи значительны (т.е. будет тратить больше времени и потребляться больше системных ресурсов на создание и уничтожение потоков, чем на их исполнение);
- *более эффективное использование памяти*
 - ✓ создание слишком большого количества потоков в одной JVM может привести к нехватке системной памяти или пробуксовке из-за чрезмерного потребления памяти (необходимы меры по ограничению количества задач, обрабатываемых в заданное время);
- *лучшее построение программы*
 - ✓ отделение управления потоками и их создание от остальной части приложения.

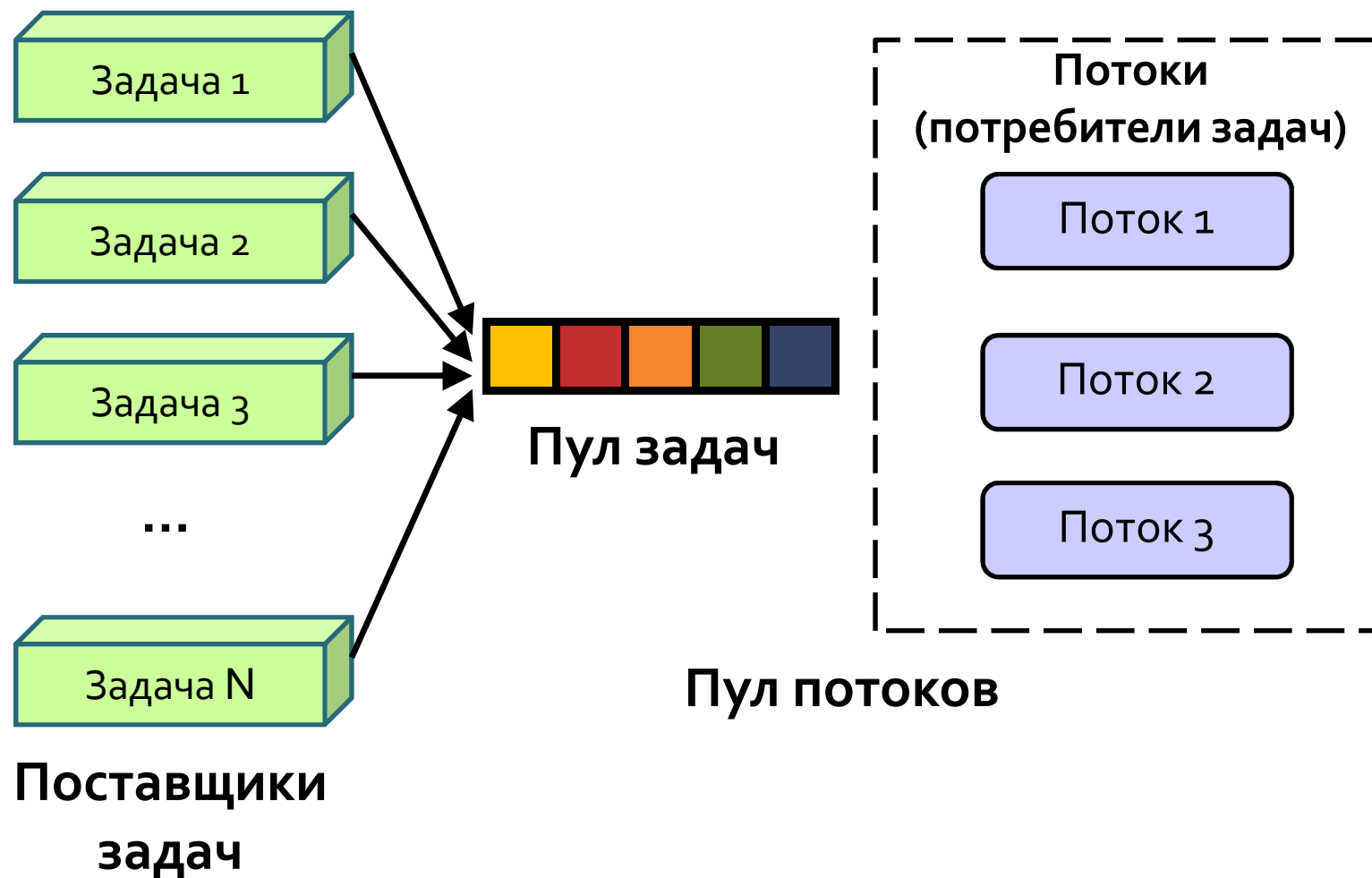
Потоки исполнения

- ❑ **Пул потоков** – это набор существующих рабочих потоков, который отделен от выполняемых им задач типа **Runnable** и **Callable** и используется для выполнения нескольких задач одновременно.



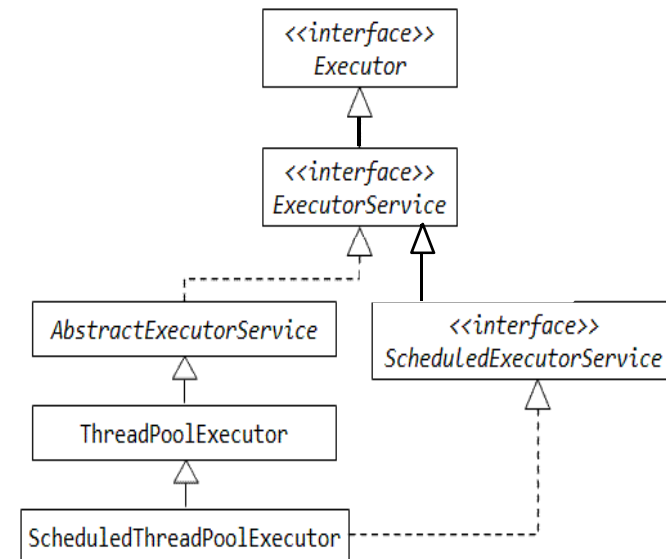
- поскольку поток уже существует, когда приходит задача, то задержка, происходящая из-за создания потока, устраняется (т.е. приложение становится более быстрореагирующим);
- правильно настроив количество потоков в пуле, предотвращается пробуксовка ресурсов (если количество задач выходит за определенные пределы, то задача ожидает до тех пор, пока поток не станет доступным, чтобы ее обработать).

ТИПОВАЯ АРХИТЕКТУРА ПУЛА ПОТОКОВ



Потоки исполнения

- ❑ Компоненты приложений, которые инкапсулируют функции создания и управления потоками, называются исполнителями (*Executors*);
- ❑ Пакет **java.util.concurrent** определяет три интерфейса исполнителей:
 - **Executor** – это простой интерфейс, который поддерживает запуск новых задач;
 - **ExecutorService** – это подинтерфейс **Executor**, который добавляет функции, помогающие управлять жизненным циклом, как задач, так и самого исполнителя;
 - **ScheduledExecutorService** – это подинтерфейс **ExecutorService**, который поддерживает отложенное и/или периодическое выполнение задач.



Потоки исполнения

I) Интерфейс Executor

- ❑ Предоставляет единственный метод *execute()*, предназначенный, чтобы быть точечной заменой для идиомы низкого уровня создания потока;
- ❑ Используется для выполнения задачи типа **Runnable**;



➤ Допустим, что ***rr*** – это объект типа **Runnable**, а ***ex*** – это объект типа **Executor**, тогда:

- ✓ **(new Thread(rr)).start();** - создание и немедленный запуск потока на низком уровне;
- ✓ **ex.execute(rr);** - это может быть тоже самое (используется существующий рабочий поток для запуска ***rr***) или, скорее всего, размещение ***rr*** в очереди для ожидания доступности рабочего потока.

II) Интерфейс ExecutorService

- ❑ Предоставляет универсальный метод *submit()*, который принимает и запускает на исполнение задачи и типа **Runnable** и типа **Callable**:
 - возвращает объект типа **Future**, который используется для получения возвращаемого значения задачи типа **Callable** и управления состоянием обеих задач **Callable** и **Runnable**;
- ❑ Предоставляет ряд методов для управления завершением работы исполнителя:
 - метод *shutdown()* – ожидает завершения запущенных задач;
 - метод *shutdownNow()* – останавливает исполнителя немедленно, прерывая запущенные задачи.
- ❑ Обеспечивает методы для подачи больших коллекций объектов **Callable**:
 - метод *invokeAll(...)* – вызывающий поток блокируется до завершения всех переданных задач;
 - метод *invokeAny(...)* – вызывающий поток блокируется до завершения любой задачи.

III) Интерфейс ScheduledExecutorService

- ❑ Предоставляет возможность откладывать начало исполнения задач на определенный промежуток времени, а также планировать выполнение задач через заданный временной интервал:



- метод *schedule(...)* - выполняет задачи типа **Callable** и **Runnable**, которые становятся доступными после указанной временной задержки;
- метод *scheduleAtFixedRate(...)* - выполняет задачи типа **Runnable**, с указанной начальной задержкой и повторяет их через указанный интервал (завершение произойдет по завершению исполнителя, интервал повторения не учитывает время выполнения задачи);
- метод *schedule WithFixedRate(...)* - выполняет задачи типа **Runnable**, с указанной начальной задержкой и повторяет их через указанный интервал между окончанием одного исполнения и началом следующего.

ИСПОЛЬЗОВАНИЕ ПУЛА ПОТОКОВ

- ❑ Создайте поток-задачу (объекты типа **Runnable** и **Callable**);
- ❑ Создайте пул потока требуемого типа;
- ❑ Передайте задачу пулу потоков вызовом соответствующего метода;
- ❑ Сохраните ссылку на объект типа **Future** для возможности отслеживать прогресс исполнения задачи (при необходимости);
- ❑ Завершите работу с пулом потоков (например, вызвав метода *shutdown()*).



Пул потоков представляется через объекты типа:

- ✓ ***ThreadPoolExecutor*** - реализующий интерфейс **ExecutorService**
- ✓ ***ScheduledExecutorService*** - реализующий интерфейс **ScheduledExecutorService**

Потоки исполнения

- Самый простой способ создать исполнителя – использовать фабричные методы класса **java.util.concurrent.Executors**:

<i>Метод</i>	<i>Назначение</i>
<code>newCachedThreadPool</code>	Создает рабочие потоки по мере необходимости (бездействующие существуют 1 минуту)
<code>newFixedThreadPool</code>	Создает пул с фиксированным количеством потоков (бездействующие не уничтожаются)
<code>newSingleThreadExecutor</code>	Создает пул с одним потоком, который последовательно выполняет задачи
<code>newScheduledThreadPool</code>	Создает пул потоков, допускающий запуск задач по графику
<code>newSingleThreadScheduledExecutor</code>	Создает пул с одним потоком, допускающий запуск задач по графику

Потоки исполнения

Пример 18:

```
public class MyTestCallable implements Callable<String> {
    private int workNumber;
    MyTestCallable(int workNumber) {
        this.workNumber = workNumber;
    }
    public String call() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Work" + workNumber + ": " + i);
            try {
                Thread.sleep((int) (Math.random() * 1000));
            } catch (InterruptedException e) {
            }
        }
        return "work " + workNumber;
    }
}
// ...
```

Потоки исполнения

Продолжение примера 18:

```
public class Main {  
    public static void main(String[] args) {  
        int numOfWorks = 20;  
        ExecutorService pool = Executors.newFixedThreadPool(4);  
        MyTestCallable works[] = new MyTestCallable[numOfWorks];  
        Future[] futures = new Future[numOfWorks];  
        for (int i = 0; i < numOfWorks; ++i) {  
            works[i] = new MyTestCallable(i + 1);  
            futures[i] = pool.submit(works[i]);  
        }  
        for (int i = 0; i < numOfWorks; ++i) {  
            try {  
                System.out.println(futures[i].get() + " ended");  
            } catch (Exception ex) {  
                ex.printStackTrace();  
            }  
        }  
    }  
}
```

Создание пула потока
как объекта класса
ThreadPoolExecutor

Получение
результата по
завершению
задачи

Потоки исполнения

Вывод в консоли:

Work 1: 1
Work 4: 1
Work 2: 1
Work 3: 1
Work 2: 2
Work 3: 2
Work 2: 3
Work 1: 2
Work 3: 3
...

...
Work 7: 1
work 1 ended
work 2 ended
work 3 ended
Work 5: 4
Work 4: 5
Work 6: 2
work 4 ended
Work 8: 1
...

...
Work 19: 3
Work 19: 4
work 17 ended
Work 20: 4
Work 18: 5
work 18 ended
Work 19: 5
Work 20: 5
work 19 ended
work 20 ended

Потоки исполнения

- ❑ Если ни один из исполнителей, предоставляемых фабричными методами, не удовлетворяет ваши потребности, то стройте **ThreadPoolExecutor** или **ScheduledThreadPoolExecutor** самостоятельно:
 - конструктор **ThreadPoolExecutor** имеет следующие параметры:
 - ❖ *corePoolSize* - число рабочих потоков пула, которые сохраняются даже если они простаивают;
 - ❖ *maximumPoolSize* - максимальное количество потоков в пуле;
 - ❖ *keepAliveTime* - когда число потоков больше чем используется, то это максимальное время, которое простаивающие потоки будут ждать новых задач перед завершением;
 - ❖ *unit* - единица измерения времени для параметра *keepAliveTime*;
 - ❖ *workQueue* – очередь для удержания задач, прежде чем они выполнятся (эта очередь будет содержать только задачи **Runnable** на представление методу *execute()*).

Потоки исполнения

Пример 19:

```
class MyTask implements Runnable {  
    private String taskInfo;  
    public MyTask(String taskInfo) {  
        this.taskInfo = taskInfo;  
    }  
    @Override  
    public void run() {  
        System.out.println(taskInfo);  
    }  
}  
// ...
```


Потоки исполнения

Продолжение примера 19:

```
public static void main(String[] args) {  
    ThreadPoolExecutor tpe =  
        new ThreadPoolExecutor(  
            5, 10, 30L, TimeUnit.SECONDS,  
            new LinkedBlockingQueue<Runnable>());  
    MyTask[] tasks = new MyTask[10];  
    for (int i = 0; i < tasks.length; i++) {  
        tasks[i] = new MyTask("Task " + i);  
        tpe.execute(tasks[i]);  
    }  
    tpe.shutdown();  
}
```

Очередь должна быть
одним из типов
блокирующих
очередей

Вывод в консоли:

Task 0
Task 2
Task 1
Task 3
Task 4
Task 5
Task 6
Task 9
Task 8
Task 7

Потоки исполнения

Изменения в примере 17:

```
public static void main(String[] args) {
```

```
.....
```

```
    CounterMath counter = new CounterMath(new File(dir), word);  
    FutureTask <Integer> task = new FutureTask <Integer> (counter);  
    new Thread(task).start();
```

```
.....
```

```
}
```

Заменено

```
public static void main(String[] args) {
```

```
.....
```

```
    ExecutorService pool = Executors.newCachedThreadPool();  
    CounterMath counter = new CounterMath(new File(dir), word, pool);  
    Future <Integer> res = pool.submit(counter);
```

```
.....
```

```
    pool.shutdown();
```

```
}
```

Потоки исполнения

```
public Integer call() {
```

```
.....
```

```
    if (ff.isDirectory()) {
```

```
        CounterMath counter = new CounterMath(ff, word);
```

```
        FutureTask <Integer> task = new FutureTask <Integer> (counter);
```

```
        res.add(task);
```

```
        new Thread(task).start();
```

```
    }
```

```
.....
```

```
}
```

Заменено

```
if (ff.isDirectory()) {
```

```
    CounterMath counter = new CounterMath(ff, word, pool);
```

```
    Future <Integer> rez = pool.submit(counter);
```

```
    res.add(rez);
```

```
}
```

Вывод в консоли:

Enter directiry -> d:/Projects\Tasm\text

Enter keyWord -> mov

22 files.

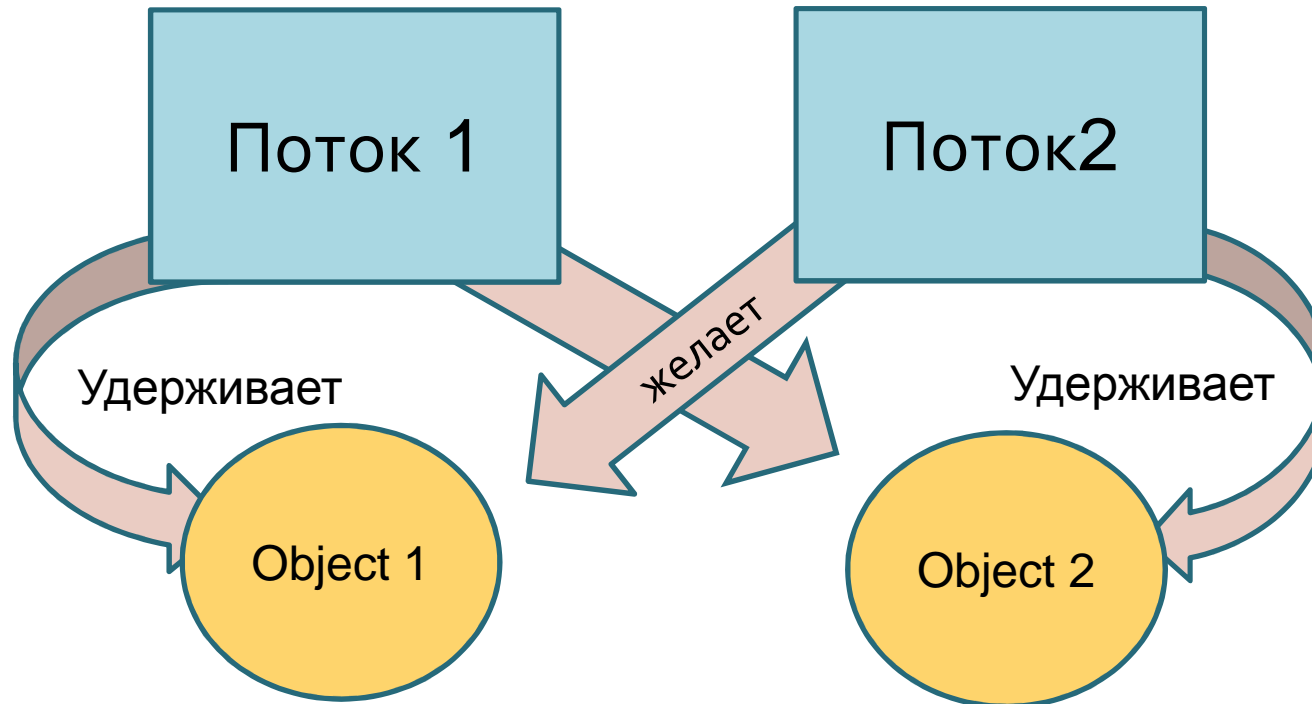
largest pool size -> 1



Тупики (Deadlocks)

Потоки исполнения

- ❑ *Тупик* описывает ситуацию, когда два или более потоков заблокированы навсегда, ожидая друг друга (т.е. когда два или более потоков ожидают друг друга, чтобы завершиться).



Потоки исполнения

Пример 1:

```
public class Main {  
    public static void main(String[] args) {
```

```
        MyObject object1 = new MyObject("Data 1");  
        MyObject object2 = new MyObject("Data 2");  
        MyObject object3 = new MyObject("Data 3");
```

```
        Thread thread1 = new MyThread("Thread_1", object1, object2, object3);  
        Thread thread2 = new MyThread("Thread_2", object2, object3, object1);  
        Thread thread3 = new MyThread("Thread_3", object3, object1, object2);
```

```
        thread1.start();  
        thread2.start();  
        thread3.start();
```

```
    }  
}
```

Создадим три
объекта класса
MyObject


Создадим три
потока для работы с
объектами класса
MyObject

Запустим потоки

Потоки исполнения

Продолжение примера 1:

```
public class MyObject {  
    public final String name;  
  
    public MyObject(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void order(MyObject object) {  
        System.out.println(((MyThread)Thread.currentThread()).threadName +  
            " Holding lock " + this.name + "...");  
        object.reply();  
    }  
  
    public synchronized void reply() {  
        System.out.println(((MyThread)Thread.currentThread()).threadName +  
            " Got lock " + this.name + "...");  
    }  
}
```



Получает объект класса MyObject для выполнения некоторых действий

Потоки исполнения

Продолжение примера 1:

```
public class MyThread extends Thread {  
    String threadName;  
    MyObject obj1 ;  
    MyObject obj2 ;  
    MyObject obj3 ;  
  
    public MyThread(String str, MyObject obj1, MyObject obj2, MyObject obj3) {  
        threadName = str;  
        this.obj1 = obj1;  
        this.obj2 = obj2;  
        this.obj3 = obj3;  
    }  
  
    public void run() {  
        synchronized (obj1) {  
            obj1.order(obj3);  
        }  
    }  
}
```

Пока поток работает с **obj1** другие потоки не могут использовать этот же объект

Потоки исполнения

Продолжение примера 1:

```
try { Thread.sleep(10);  
} catch (InterruptedException e) { }
```

Когда работа с **obj1** завершена, то уступить процессорное время другим потокам

```
synchronized (obj2) {  
    obj2.order(obj1);  
}
```

Пока поток работает с **obj2** другие потоки не могут использовать этот же объект

```
try { Thread.sleep(10);  
} catch (InterruptedException e) { }
```

Когда работа с **obj2** завершена, то уступить процессорное время другим потокам

```
synchronized (obj3) {  
    obj3.order(obj2);  
}
```

Пока поток работает с **obj3** другие потоки не могут использовать этот же объект

```
try { Thread.sleep(10);  
} catch (InterruptedException e) { }
```

Когда работа с **obj3** завершена, то уступить процессорное время другим потокам

```
}  
}
```

Потоки исполнения

Варианты выполнения программы

Вывод в консоли:


Thread_1 Holding lock Data 1...
Thread_1 Got lock Data 3...
Thread_2 Holding lock Data 2...
Thread_2 Got lock Data 1...
Thread_3 Holding lock Data 3...
Thread_3 Got lock Data 2...
Thread_1 Holding lock Data 2...
Thread_1 Got lock Data 1...
Thread_2 Holding lock Data 3...
Thread_2 Got lock Data 2...
Thread_3 Holding lock Data 1...
Thread_3 Got lock Data 3...
Thread_1 Holding lock Data 3...
Thread_1 Got lock Data 2...
Thread_3 Holding lock Data 2...
Thread_3 Got lock Data 1...
Thread_2 Holding lock Data 1...
Thread_2 Got lock Data 3...

Вывод в консоли:

Thread_1 Holding lock Data 1...
Thread_2 Holding lock Data 2...
Thread_3 Holding lock Data 3...

Вывод в консоли:

Thread_1 Holding lock Data 1...
Thread_1 Got lock Data 3...
Thread_3 Holding lock Data 3...
Thread_2 Holding lock Data 2...
Thread_2 Got lock Data 1...
Thread_3 Got lock Data 2...
Thread_1 Holding lock Data 2...
Thread_2 Holding lock Data 3...
Thread_3 Holding lock Data 1...



Механизм синхронизаторов общего назначения

Потоки исполнения

- ❑ Предназначены для управления набором взаимодействующих потоков.

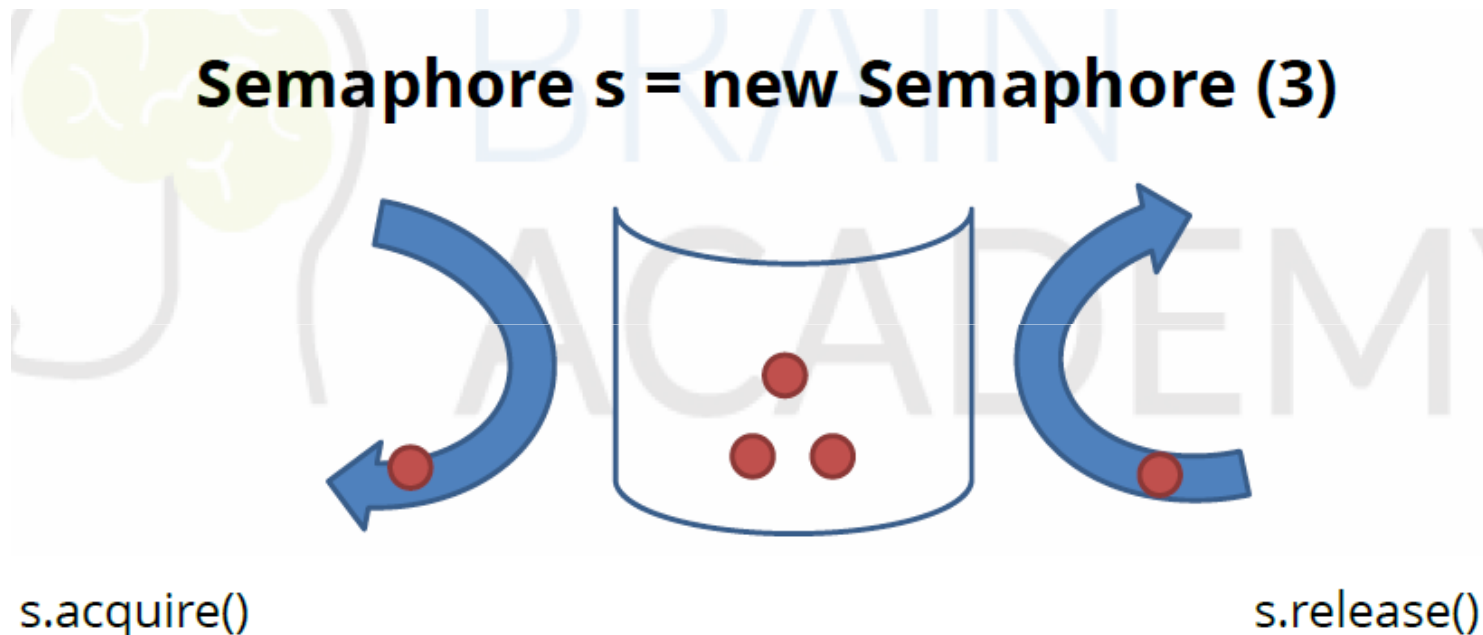
<i>Класс</i>	<i>Назначение</i>	<i>Использование</i>
Semaphore	Несколько потоков запрашивает, ожидает и получает разрешения на дальнейшую работу	Когда необходимо ограничить общее количество потоков, имеющих доступ к ресурсам
CyclicBarrier	Набор потоков ожидает, пока определенное их количество не достигнет барьера, после чего возможно выполнение действия, связанного с этой точкой	Когда результаты могут быть использованы только после того, как некоторое количество потоков завершит свою задачу

Потоки исполнения

<i>Класс</i>	<i>Назначение</i>	<i>Использование</i>
CountDownLatch	Набор потоков ожидает, пока некоторый счетчик не станет равным нулю	Когда потоки должны ожидать исходных данных для своей работы
Exchanger	Два потока обрабатывают два экземпляра одной структуры данных	Когда потоки должны обмениваться своими данными
SynchronousQueue	Один поток должен передать объект другому потоку	Когда поток должен передать объект другому потоку при готовности обоих выполнить такую передачу

Потоки исполнения

- ❑ **Семафор** (класс *Semaphore*) – это управление доступом к объекту на основе запрещений/разрешений.



- Чаще всего используются для ограничения количества потоков при работе с аппаратными ресурсами или файловой системой.

Особенности семафора

- ✓ всегда изначально устанавливается на максимальное число потоков, одновременное функционирование которых может быть разрешено;
- ✓ при превышении этого значения все желающие выполняться потоки будут приостановлены до освобождения семафора одним из потоков, работающих по его разрешению;
- ✓ при каждом новом разрешении счетчик допущенных задач на выполнение уменьшается на единицу;
- ✓ при освобождении семафора счетчик увеличивается на единицу.

Потоки исполнения

- ❑ Для получения разрешения запрашивается метод **acquire()**.
- ❑ Для освобождения вызывается метод **release()**.
- ❑ Стандартное взаимодействие методов **acquire()** и **release()**:

```
public void run() {  
    try {  
        semaphore.acquire();  
        // код использования защищаемого ресурса  
    } catch (InterruptedException e) { }  
    finally {  
        semaphore.release();    // освобождение семафора  
    }  
}
```

Потоки исполнения

- ❑ Семафор с максимальным значением **1**, используется в качестве затвора. *Например,*
 - некоторое приложение выполняет работу частями, чтобы дать возможность пользователю ознакомиться с промежуточными результатами и инициировать продолжение работы);
 - очередь на флюорографию в поликлинике (когда пациент в кабинете, то горит красная лампа, выходит – зеленая).

- ❑ Конструкторы:
 - *Semaphore* (int *permits*) – создает семафор с указанным числом разрешений и несправедливой установкой разрешений.
 - *Semaphore* (int *permits*, boolean *fair*) – создает семафор с указанным числом разрешений и с установкой указанной справедливости (**true** – предоставление разрешений в порядке запросов).

Потоки исполнения

Пример 2:

```
public class Main {  
    public static final int ITEMS_COUNT = 15;  
    public static double items[];  
    // семафор, контролирующий разрешение на доступ к элементам массива  
    public static Semaphore sortSemaphore = new Semaphore(0, true);  
  
    public static void main(String[] args) {  
        items = new double[ITEMS_COUNT];  
        for (int i = 0 ; i < items.length ; ++i)  
            items[i] = Math.random();  
  
        new Thread(new ArraySort(items)).start();  
  
        for (int i = 0 ; i < items.length ; ++i) {  
            sortSemaphore.acquireUninterruptibly();  
            System.out.println(items[i]);  
        }  
    }  
}
```

Потоки исполнения

Продолжение примера 2:

```
class ArraySort implements Runnable {
```

```
    private double items[];
```

```
    public ArraySort(double items[]) {        this.items = items;    }
```

```
    public void run() {
```

```
        for (int i = 0 ; i < items.length - 1 ; ++i) {
```

```
            for (int j = i + 1 ; j < items.length ; ++j) {
```

```
                if ( items[i] < items[j] ) {    double tmp = items[i];
```

```
                    items[i] = items[j];
```

```
                    items[j] = tmp;
```

```
                } }
```

```
                // освобождение семафора
```

```
                Main.sortSemaphore.release();
```

```
                try {                Thread.sleep(71);
```

```
                } catch (InterruptedException e) {    System.err.print(e);    }
```

```
            }
```

```
            Main.sortSemaphore.release();
```

```
        }
```

```
    }
```

Модифицированная
сортировка
выборкой

Первый
проход

Освободить семафор для
доступа к упорядочному
элементу

Освободить семафор после
упорядочивания всего массива

Потоки исполнения

Класс **CyclicBarrier**

- ❑ Поддерживает «рандеву» под названием барьер (точка синхронизации).
- ❑ Потоки, достигая барьера, переходят в состояние «ожидания» (вызывается метод *wait()*), пока все определенные потоки не достигнут барьера для того, чтобы объединить результаты их работы.

Конструкторы:

`CyclicBarrier(int parties);`

`CyclicBarrier(int parties, Runnable barrierAction);`

Когда все потоки собрались у барьера, то выполнить указанную работу, затем отпустить потоки

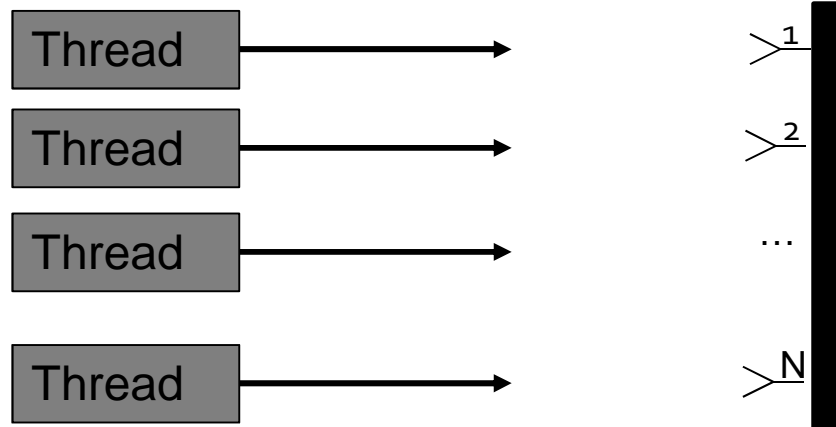


Барьер можно использовать повторно после освобождения всех ожидающих потоков.

Например, переправа на пароме через реку или пролив.

Потоки исполнения

CyclicBarrier



Потоки исполнения

Пример 3, вычисление суммы элементов матрицы, когда вычисление суммы элементов каждой строки происходит в отдельном потоке:

```
public class Main {  
    private static int matrix[][] = new int [5][6];  
    private static int rez[];  
  
    static {  
        for (int i=0; i<matrix.length; i++)  
            for (int j=0; j<matrix[i].length; j++)  
                matrix[i][j] = (int)(Math.random()*20);  
    }  
}
```

Потоки исполнения

Продолжение примера 3:

```
private static class Summator extends Thread {  
    int row;      CyclicBarrier barrier;  
    public Summator(int row, CyclicBarrier barrier) {  
        this.row = row;      this.barrier = barrier;  
    }  
    public void run() {  
        int[] line = matrix[row];  
        for (int value : line)  
            rez[row] += value;  
        System.out.println("Summa (" + (row+1) + ") = " + rez[row]);  
        try {  
            barrier.await();  
        } catch (BrokenBarrierException | InterruptedException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

Потоки исполнения

Продолжение примера 3:

```
public static void main(String[] args) {  
    final int ROWW = matrix.length;  
    rez = new int [ROWW ];  
  
    printMatrix();           // статический метод вывода матрицы  
  
    Runnable merger = new Runnable() {  
        public void run() {  
            int sum = 0;  
            for (int value : rez)  
                sum += value;  
            System.out.println("Summa elements matrix = " + summa);  
        }  
    };  
};
```

Потоки исполнения

Продолжение примера 3:

```
CyclicBarrier barrier = new CyclicBarrier(ROWW, merger);  
    for (int i=0; i<ROWW; i++)  
        new Summator(i,barrier).start();  
}  
  
public static void printMatrix() {  
    System.out.println("Matrix:");  
    for (int[] row : matrix) {  
        for (int value : row)  
            System.out.printf("%5d", value);  
        System.out.println();  
    }  
}
```

Вывод в консоли:

Matrix:

8	19	4	12	8	12
11	5	17	3	15	15
0	17	14	11	17	16
12	15	6	1	8	0
9	9	19	4	0	8

Summa (1) = 63

Summa (2) = 66

Summa (3) = 75

Summa (4) = 42

Summa (5) = 49

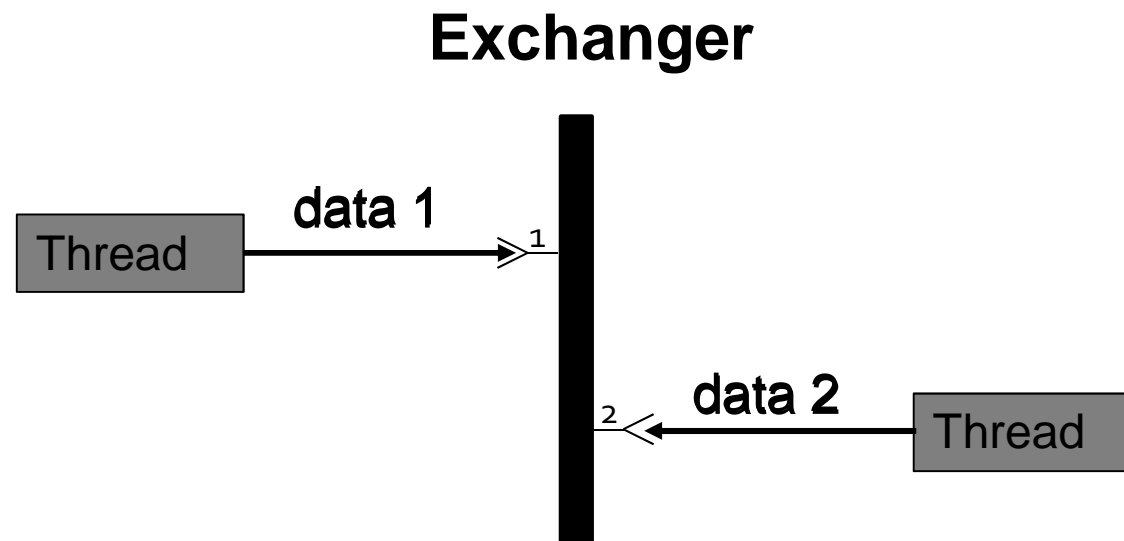
Summa elements matrix = 295

Потоки исполнения

Класс **Exchanger**

- ❑ Поддерживает передачу данных между потоками в две стороны не заботясь об синхронизации.
- ❑ Для выполнения обмена вызывается метод
<Type> exchange(<Type> sendlingValue)
- ❑ После вызова данного метода поток переходит в состояние «ожидания», пока другой поток не вызовет этот же метод на этом же объекте.

Потоки исполнения



Потоки исполнения

Пример 4, измененный пример 3:

2) Изменим только метод *main*:

```
public static void main(String[] args) {  
    final int ROWW = matrix.length;  
    rez = new int [ROWW ];  
    final Exchanger <Integer> myExchanger = new Exchanger();  
    int result = 0;  
    Runnable merger = new Runnable() {  
        public void run() {  
            int sum = 0;  
            for (int value : rez)  
                sum += value;  
            try { myExchanger.exchange(sum);  
            } catch (InterruptedException e) { e.printStackTrace(); }  
        }  
    };  
};
```

//

Добавим описание
класса обменщика и
переменную для
приема суммы

Добавим вызов передачи суммы потоку,
который ожидает обмена

Потоки исполнения

Продолжение примера 4:

```
CyclicBarrier barrier = new CyclicBarrier(ROWW, merger);
```

```
for (int i=0; i<ROWW; i++)
```

```
    new Summator(i,barrier).start();
```

```
    try {
```

```
        result = myExchanger.exchange(0);
```

```
    } catch (InterruptedException e) {
```

```
        e.printStackTrace();
```

```
    }
```

```
    System.out.println("Sum of matrix elements = "
```

```
        + result);
```

```
}
```

Вывод в консоли:

Matrix:

16 0 11 14 12 17

18 12 3 18 4 13

0 1 15 8 5 19

1 7 9 7 8 17

13 9 10 13 2 1

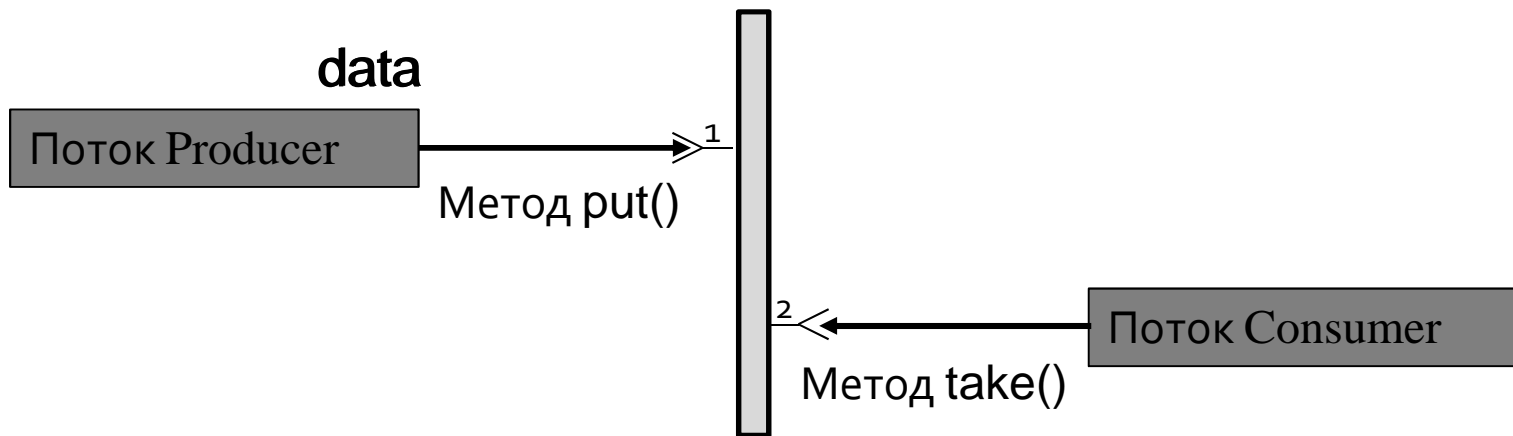
Sum of matrix elements = 283

Добавим, после запуска потоков вычислений сумм строк, вызов обменщика для получения результата, а затем выведем полученный результат

Класс **SynchronousQueue**

- ❑ Поддерживает передачу данных между потоками в одну сторону, т.е один поток отдает данные, другой – получает.
- ❑ Для поставки значения вызывается метод
void put(<Type> item)
- ❑ Для получения значения вызывается метод
<Type> take()
- ❑ Передача данных может осуществляться только в случае, когда есть поток ожидающий поступления.

SynchronousQueue



Потоки исполнения

Пример 5:

```
class Producer implements Runnable {  
    private SynchronousQueue <String> drop;  
    List <String> messages = Arrays.asList( "Mares eat oats", "Does eat oats",  
                                           "Little lambs eat ivy", "Wouldn't you eat ivy too?");  
    public Producer(SynchronousQueue <String> drop) {    this.drop = drop;    }  
    public void run() {  
        try {  
            for (String s : messages)  
                drop.put(s);  
            drop.put("DONE");  
        } catch (InterruptedException i ntEx) {  
            System.out.println("Interrupted! Last one out, turn out the lights!");  
        }  
    }  
}
```

Потоки исполнения

Продолжение примера 5:

```
class Consumer implements Runnable {  
    private SynchronousQueue <String> drop;  
    public Consumer(SynchronousQueue <String> d) { this.drop = d; }  
    public void run() {  
        try {  
            String msg = null;  
            while (!((msg = drop.take()).equals("DONE")))  
                System.out.println(msg);  
        } catch (InterruptedException intEx) {  
            System.out.println("Interrupted! Last one out, turn out the lights!");  
        }  
    }  
}
```

Потоки исполнения

Продолжение примера 5:

```
public class Main {  
    public static void main(String[] args) {  
        SynchronousQueue <String> drop =  
            new SynchronousQueue<String>();  
        new Thread(new Producer(drop)).start();  
        new Thread(new Consumer(drop)).start();  
    }  
}
```

Вывод в консоли:

```
Mares eat oats  
Does eat oats  
Little lambs eat ivy  
Wouldn't you eat ivy too?
```

Потоки исполнения

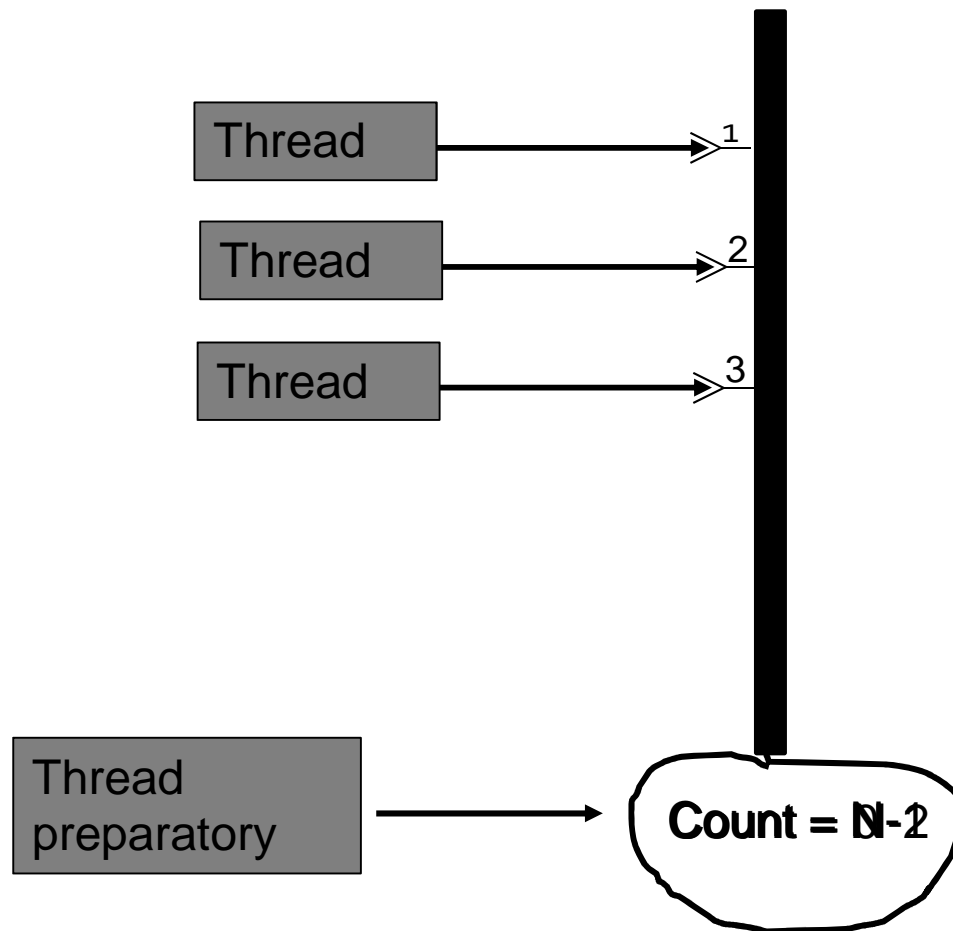
Класс **CountDownLatch**

- ❑ Поддерживает ожидание для набора потоков до тех пор, пока значение некоторого счетчика не обнулится.
- ❑ Отличия от класса **CyclicBarrier**:
 - для снятия затвора необязательно, чтобы все потоки набора находились в состоянии «ожидания»;
 - значение счетчика может быть уменьшено обработчиками внешних событий;
 - затвор устанавливается единожды (повторное использование после обнуления счетчика невозможно).
- ❑ Этот класс напоминает стартовый барьер на скачках: задерживает все потоки до тех пор, пока не будет выполнено определенное условие, которое освобождает все потоки одновременно

Потоки исполнения

- ❑ Для создания затвора вызывается конструктор
CountDownLatch(int count)
- ❑ Поток, который должен ожидать наступления некоторых условий (данных) для своей работы, вызывает метод
void await()
- ❑ Потоки, которые подготавливают данные (условия) вызывают метод
public void countDown()

CountDownLatch



Потоки исполнения

Пример 6:

```
public class Main {  
    public static void main(String args[]) {  
        CountdownLatch cdl = new CountdownLatch(5);  
        long timeS = 0,    timeF = 0;  
        new MyThread(cdl);  
        try {  
            System.out.print("Главный поток ожидает -> ");  
            timeS = System.nanoTime();  
            cdl.await();  
            timeF = System.nanoTime();  
        } catch (InterruptedException exc) {  
            System.out.println(exc);  
        }  
        double time = (timeF-timeS)/1_000_000_000.0;  
        System.out.printf("%7.5f секунд\n", time);  
    }  
}
```

Потоки исполнения

Продолжение примера 6:

```
class MyThread implements Runnable {
```

```
    CountdownLatch latch;
```

```
    MyThread(CountDownLatch c) {
```

```
        latch = c;
```

```
        new Thread(this).start();
```

```
    }
```

```
    public void run() {
```

```
        for (int i = 0; i < 5; i++) {
```

```
            try {
```

```
                Thread.sleep((int)(Math.random()*20));
```

```
            } catch (InterruptedException e) { e.printStackTrace(); }
```

```
            latch.countDown();           // decrement count
```

```
        }
```

```
    }
```

Вывод в консоли:

Главный поток ожидает -> 0,03949 секунд

Потоки исполнения

Примера 7:

```
class Runner extends Thread {  
    private CountdownLatch timer;  
    public Runner(CountDownLatch cdl, String name) {  
        timer = cdl;  
        this.setName(name);  
        System.out.println(this.getName() +  
            " ready and waiting to start");  
        start();  
    }  
    ...  
}
```

Потоки исполнения

Продолжение примера 7:

```
public void run() {  
    try {  
        timer.await();  
    } catch (InterruptedException ie) {  
        System.err.println("interrupted -"+  
            "can't start running the race");  
    }  
    System.out.println(this.getName() +  
        " started running");  
}  
}
```

Потоки исполнения

Продолжение примера 7:

```
public class Demo {  
    public static void main(String[] args) {  
        CountdownLatch counter = new CountdownLatch(5);  
        new Runner(counter, "Carl");  
        new Runner(counter, "Joe");  
        new Runner(counter, "Jack");  
        System.out.println("Starting the countdown ");  
        long countVal = counter.getCount();  
        while (countVal > 0) {  
            Thread.sleep(1000);  
            System.out.println(countVal);  
            if (countVal == 1) {  
                System.out.println("Start");  
            }  
            counter.countDown();  
            countVal = counter.getCount();  
        }  
    }  
}
```


Потоки исполнения

Вывод в консоли:

```
Carl ready and waiting to start
Joe ready and waiting to start
Jack ready and waiting to start
Starting the countdown
5
4
3
2
1
Start
Joe started running
Carl started running
Jack started running
```



Параллельное программирование



Потоки исполнения

- ❑ *Параллельное программирование* – это общее название технологий, которые используют в своих целях многоядерные процессоры, содержащие два и более ядер.
- ❑ Преимуществом систем, основанных на параллельном программировании, является значительное увеличение производительности.
- ❑ В комплект JDK 7 (пакет **java.util.concurrent**) добавлен новый фреймворк для поддержки параллельного программирования, называемый **Fork/Join Framework**:
 - упрощает создание и использование нескольких потоков;
 - позволяет автоматически масштабировать количество доступных для использования процессоров.

Потоки исполнения

Основные классы:

<code>ForkJoinTask<V></code>	Абстрактный класс, определяющий задачу
<code>ForkJoinPool</code>	Управляет выполнением объекта класса <code>ForkJoinTask</code>
<code>RecursiveAction</code>	Производный от класса <code>ForkJoinTask<V></code> класс для задач, которые не возвращают значений
<code>RecursiveTask<V></code>	Производный от класса <code>ForkJoinTask<V></code> класс для задач, возвращающих значения

Основные методы:

```
final ForkJoinTask<V> fork()  
final V join()
```

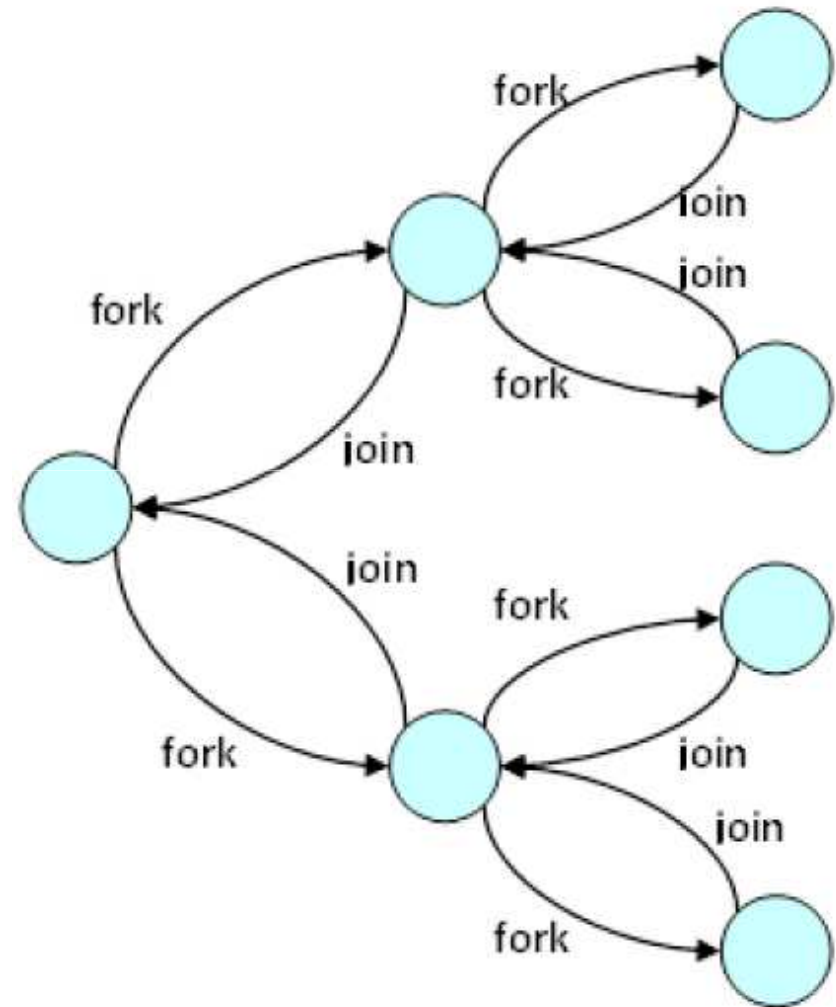
- Метод `fork()` передает задачу для асинхронного исполнения (т.е. поток, который вызывает этот метод, продолжает исполняться).
- Метод `join()` ожидает завершения задачи, для которой он вызывается.

Потоки исполнения

Операция *fork()* (разветвить) начинает новую параллельную подзадачу **fork/join**.

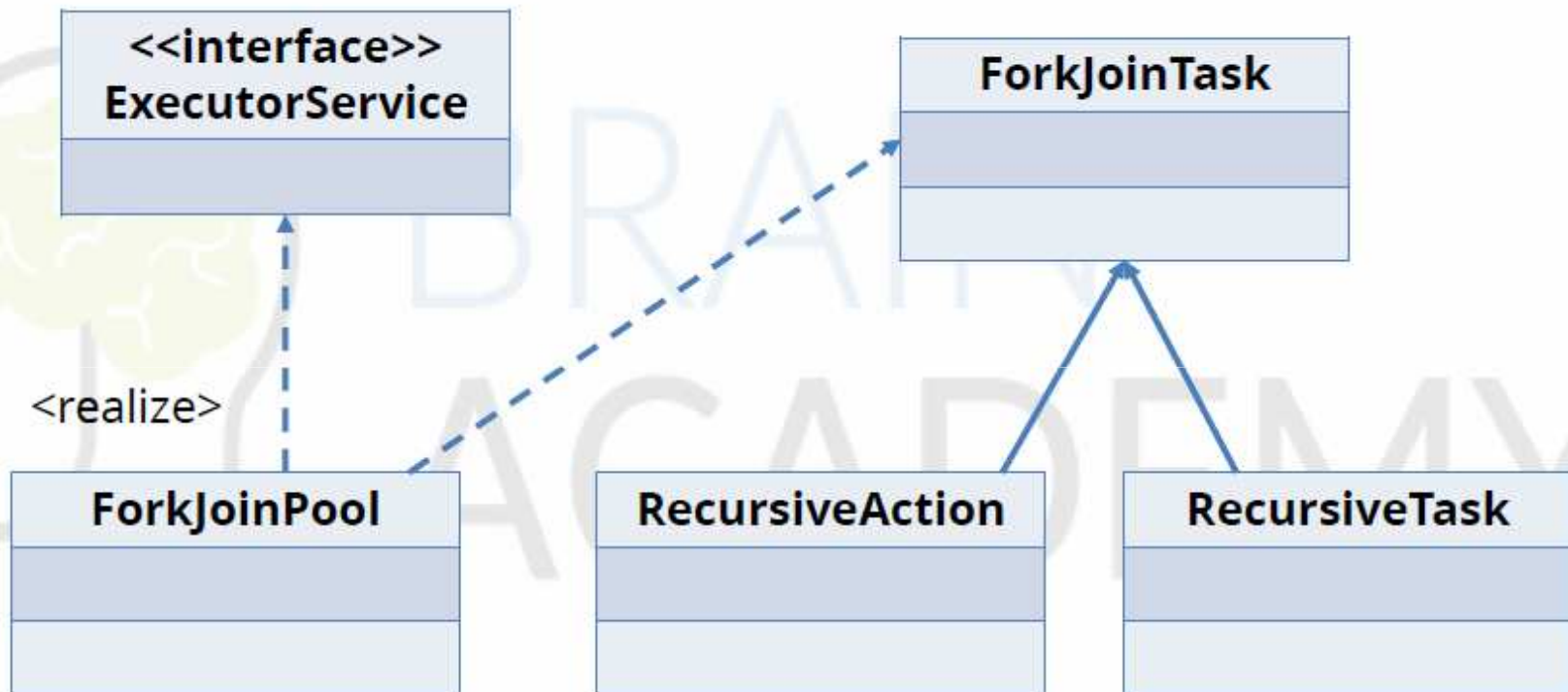
Операция *join()* выполняет присоединение к текущей задаче когда раздвоенные подзадачи завершены.

Это рекурсивный алгоритм типа «разделяй и властвуй», неоднократно разделяет подзадачи, пока они не достаточно малы, чтобы решаться.



Потоки исполнения

Структура основных компонентов



Потоки исполнения

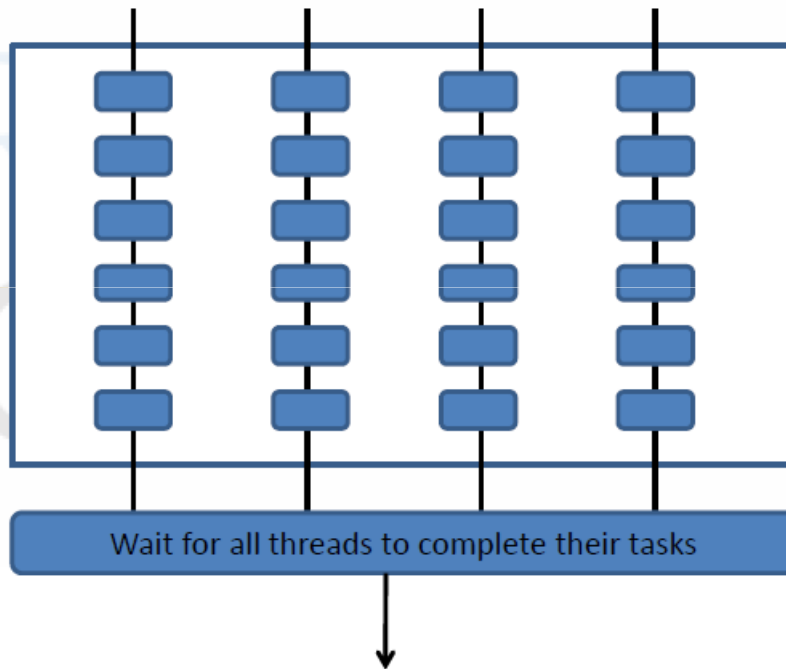
- ❑ Чтобы создать задачу необходимо расширить следующие классы:
 - **RecursiveAction**, который не возвращает результат;
 - **RecursiveTask**, который возвращает результат.
- ❑ В этих классах определено 4 метода, но основной метод, тело которого и представляет задачу, является метод:

*protected abstract void **compute()***
*protected abstract V **compute()***
- ❑ Для выполнения задач используется класс **ForkJoinPool**, который обычно содержит количество рабочих потоков, соответствующее количество процессоров.
- ❑ Этот класс поддерживает выполнение своих потоков, используя подход «захват задачи» (work-stealing).

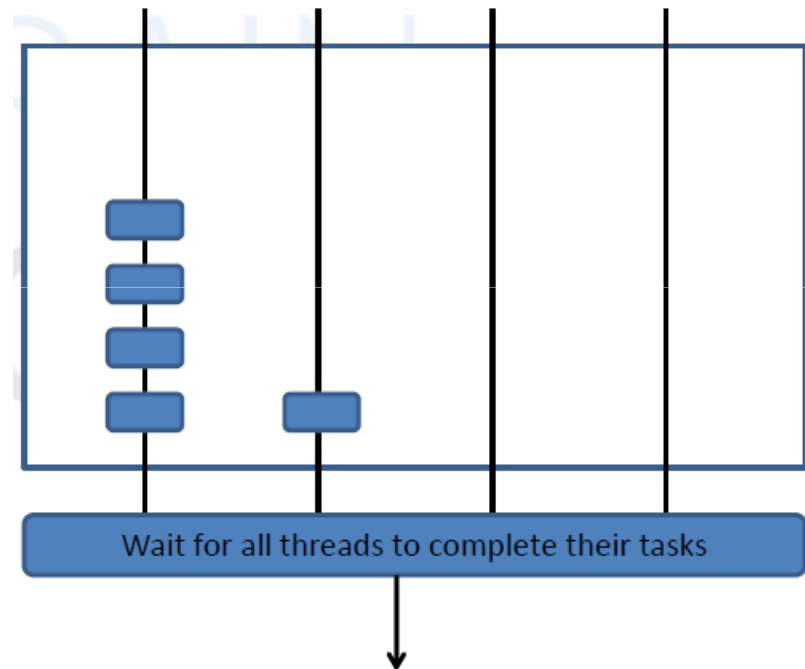
Потоки исполнения

Подход «захват задачи»

Начальное состояние

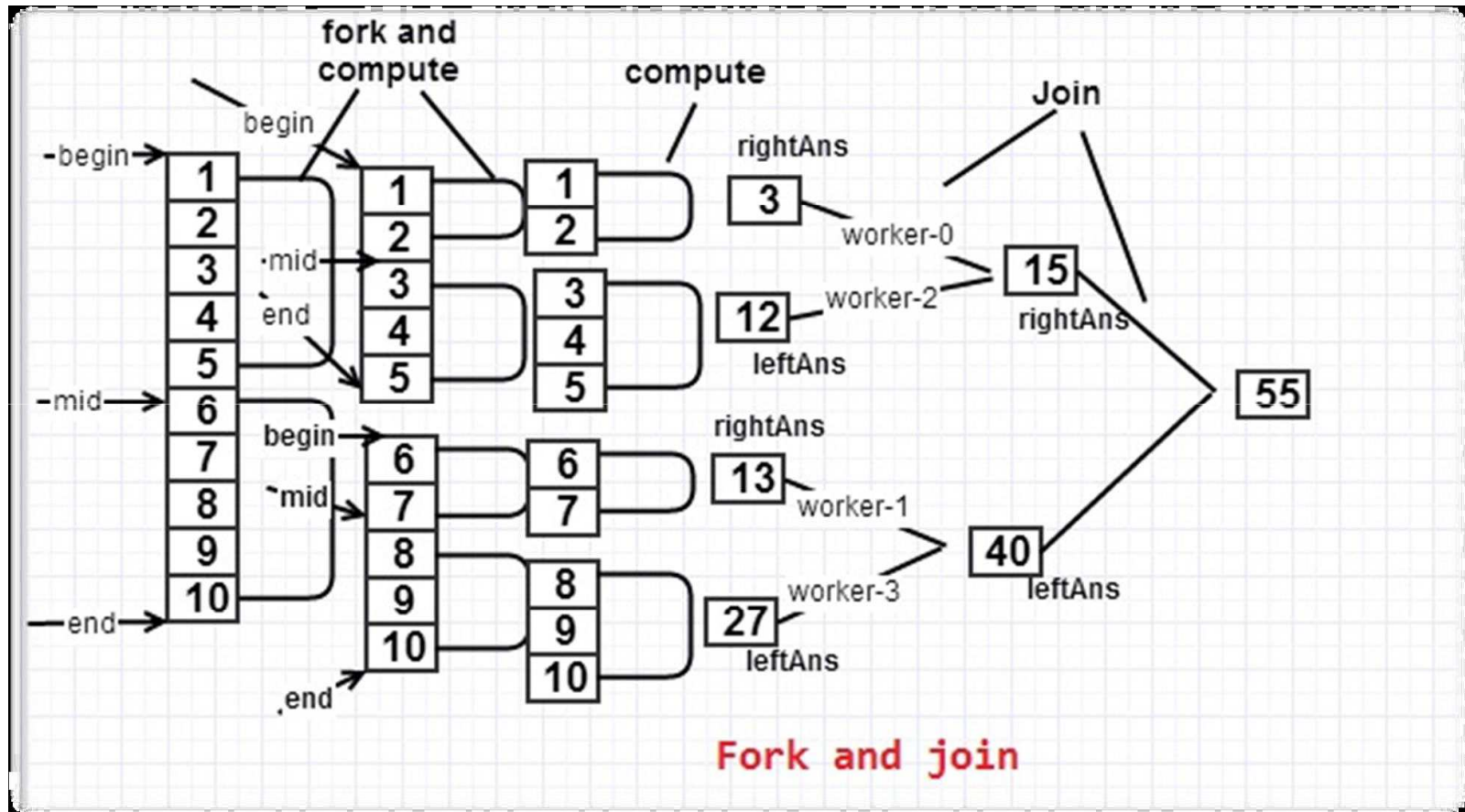


Текущее состояние



Потоки исполнения

Пример.



Потоки исполнения

Пример 8:

```
public class SumOfNUsingForkJoin {  
    private static long N = 1000_000L;  
    private static final int NUM_THREADS = 10;  
    static class RecSumOfN extends RecursiveTask<Long> {  
        long from, to;  
        public RecSumOfN(long from, long to) {  
            this.from = from;  
            this.to = to;  
        }  
    }  
}
```

Количество чисел

Количество потоков

Диапазон чисел

Потоки исполнения

Продолжение примера 8:

```
public Long compute() {  
    if ((to - from) <= N / NUM_THREADS) {  
        long localSum = 0;  
        for (long i = from; i <= to; i++) {  
            localSum += i;  
        }  
        System.out.printf("\t Summing of range %d to %d is %d %n",  
            from, to, localSum);  
        return localSum;  
    } else {  
        long mid = (from + to) / 2;  
        System.out.printf("Forking into two ranges: " +  
            "%d to %d and %d to %d %n", from, mid, mid, to);  
        RecSumOfN firstHalf = new RecSumOfN(from, mid);  
        firstHalf.fork();  
        RecSumOfN secondHalf = new RecSumOfN(mid + 1, to);  
        long resultSecond = secondHalf.compute();  
        return firstHalf.join() + resultSecond;  
    }  
}
```

Если количество чисел
меньше предполагаемого
значения для потока,
то начать вычисления

Найти сумму чисел

Поделить диапазон чисел пополам

Создать задачу
для 1-ой
половины

Отделить в отдельный поток

Запустить в этот же потоке

Потоки исполнения

Продолжение примера 8:

```
public static void main(String[] args) {  
    ForkJoinPool pool = new ForkJoinPool(NUM_THREADS);  
    long computedSum = pool.invoke(new RecSumOfN(0, N));  
    long formulaSum = (N * (N + 1)) / 2;  
    System.out.printf("Sum for range 1..%d; computed sum =  
    %d, formula sum = %d %n", N, computedSum, formulaSum);  
}  
}
```

Создать пул

Для сравнения результата

Потоки исполнения

Вывод в консоли:

Forking computation into two ranges: 0 to 500000 and 500000 to 1000000

Forking computation into two ranges: 500001 to 750000 and 750000 to 1000000

Forking computation into two ranges: 750001 to 875000 and 875000 to 1000000

Forking computation into two ranges: 875001 to 937500 and 937500 to 1000000

Forking computation into two ranges: 0 to 250000 and 250000 to 500000

Forking computation into two ranges: 250001 to 375000 and 375000 to 500000

Forking computation into two ranges: 375001 to 437500 and 437500 to 500000

Summing of range 937501 to 1000000 is 60546906250

Summing of range 437501 to 500000 is 29296906250

Summing of range 375001 to 437500 is 25390656250

Forking computation into two ranges: 250001 to 312500 and 312500 to 375000

Forking computation into two ranges: 0 to 125000 and 125000 to 250000

Summing of range 875001 to 937500 is 56640656250

Forking computation into two ranges: 750001 to 812500 and 812500 to 875000

Forking computation into two ranges: 125001 to 187500 and 187500 to 250000

Потоки исполнения

Продолжение вывода в консоли:

Summing of range 187501 to 250000 is 13671906250
Summing of range 125001 to 187500 is 9765656250
Summing of range 312501 to 375000 is 21484406250
Summing of range 812501 to 875000 is 52734406250
Forking computation into two ranges: 0 to 62500 and 62500 to 125000
Summing of range 62501 to 125000 is 5859406250
Summing of range 0 to 62500 is 1953156250
Forking computation into two ranges: 500001 to 625000 and 625000 to 750000
Forking computation into two ranges: 625001 to 687500 and 687500 to 750000
Summing of range 687501 to 750000 is 44921906250
Summing of range 625001 to 687500 is 41015656250
Forking computation into two ranges: 500001 to 562500 and 562500 to 625000
Summing of range 562501 to 625000 is 37109406250
Summing of range 500001 to 562500 is 33203156250
Summing of range 250001 to 312500 is 17578156250
Summing of range 750001 to 812500 is 48828156250
Sum for range 1..1000000; computed sum = 500000500000, formula sum = 500000500000