Programación paralela en Java

Metodología de la Programación Paralela

Jesús Sánchez Cuadrado (jesusc@um.es)

Curso 2019/20

Introducción

- Concurrencia es parte del lenguaje
- Soporte nativo para hilos en la JVM
  - Clase Thread
  - Objetos son monitores
  - Frameworks de concurrencia (> Java 7)
- Objetivo
  - Aplicaciones más rápidas
  - Aplicaciones que respondan mejor
- Reto
  - Construcción de programas OO correctos
  - Mantener la integridad de las clases

Introducción

`CalculaPi.java`

- Un hilo extiende de la clase Thread
  - Thread.start
  - Thread.join
- Se puede crear un Runnable
  - Pasarlo como argumento a un objeto Thread en la construcción

Construcciones básicas

Servidor simple

ServidorSimple.java

```
private static void aceptarConexion(Socket s, MyServlet servlet)
  throws IOException {

  BufferedReader in = new BufferedReader(
     new InputStreamReader(s.getInputStream()));
  String line = in.readLine();
  if (line != null) {
    servlet.process(line);
  }
  in.close();

}
```

Servidor simple

ServidorSimple.java

```
public static class MyServlet {
  private int contador = 0;

  public void process(String message) {
      if (message.equals("contar")) {
        contador++;
      } else if (message.equals("mostrar")) {
        System.out.println(contador);
      }
  }
}
```

¿Se pueden aprovechar mejor los recursos del servidor?

- Múltiples cores
- Solapar I/O con otras peticiones

Servidor con hilos

ServidorHilos.java

```
try (ServerSocket socket = new ServerSocket(9000)) {
  while (true) {
    final Socket s = socket.accept();
    Runnable r = new Runnable() {
        @Override
        public void run() {
           try {
              aceptarConexion(s, servlet);
           } catch (IOException e) {
              e.printStackTrace();
           }
        }
    };
    new Thread(r).start();
}
```

Procesamiento concurrente

Condiciones de carrera

- Cuando la corrección del programa depende de cómo se intercale la ejecución de los hilos.
  - Si el contador se usa para generar p.ej., IDs únicos en el sistema estaríamos violando la integridad del sistema

- Tipos
  - check-then-act
    - Uso de un dato desactualizado para tomar una decisión
    - Ejemplo, inicialización perezosa
  - read-modify-write
    - El nuevo estado se deriva de un estado previo
    - Ejemplo, contador compartido por varios hilos

```
private Fat instance = null;
public Fat getInstance() {
 if (instance == null)
   instance = new Fat();
 return instance;
}
```

Condiciones de carrera

- Evitar que un hilo use una variable cuando otro hilo está a mitad de modificarla.
  - Un hilo puede “ver” el estado de una variable antes o después, pero no a mitad de su proceso de actualización
  - La actualización debe ser **atómica**
- Pueden existir varias formas de conseguir atomicidad
  - Locks
  - Variables atómicas

Variables atómicas

- Clases de soporte para diferentes tipos
  - AtomicInteger
  - AtomicLong
  - AtomicBoolean
  - AtomicReference

- Métodos
  - set
  - compareAndSet
  - incrementAndGet
  - etc.

Variables atómicas

```
public static class MyServlet {
  private AtomicInteger contador = new AtomicInteger(0);

  public void process(String message) {
    if (message.equals("contar")) {
       contador.incrementAndGet();
    } else if (message.equals("mostrar")) {
       System.out.println(contador.get());
    }
  }
}
```

Variables atómicas

- Basadas en operaciones atómicas de tipo CAS (compare-and-swap)
- Una operación CAS tiene tres operandos:
  - La dirección de memoria sobre la que opera (M)
  - El valor que se espera que tenga la variable (A)
  - El nuevo valor que se quiere establecer (B)
- La operación CAS actualiza M con B, si y sólo si el valor actual en M es el mismo que A. Si no, la actualización falla.
- Si varios hilos intentan actualizar, uno de ellos gana y el resto pierden.
  - No hay cambios de contexto
  - Un hilo puede reintentar más adelante

Varias variables

- ¿Es esté código Thread Safe?
- ¿Podríamos utilizar variables atómicas?

```
public static class MyServlet {
  private int ultimoNumero = -1;
  private int ultimoResultado = -1;

  public void process(String mensaje) {
    if (mensaje.contains(“fibonacci")) {
       int numero = extraer(mensaje);
       int resultado;
       if (ultimoNumero == numero) {
          resultado = ultimoResultado;
       } else {
          resultado = fibonacci(numero);
          ultimoResultado = resutado;
          ultimoNumero = numero;
       }
       // mostrar resultado
    } ...
  }
```

Lock intrínseco

- Cualquier objeto en Java puede actuar como un *lock* a través de la palabra clave `synchronized`

```
synchronized (obj) {
  … región crítica …
}
```

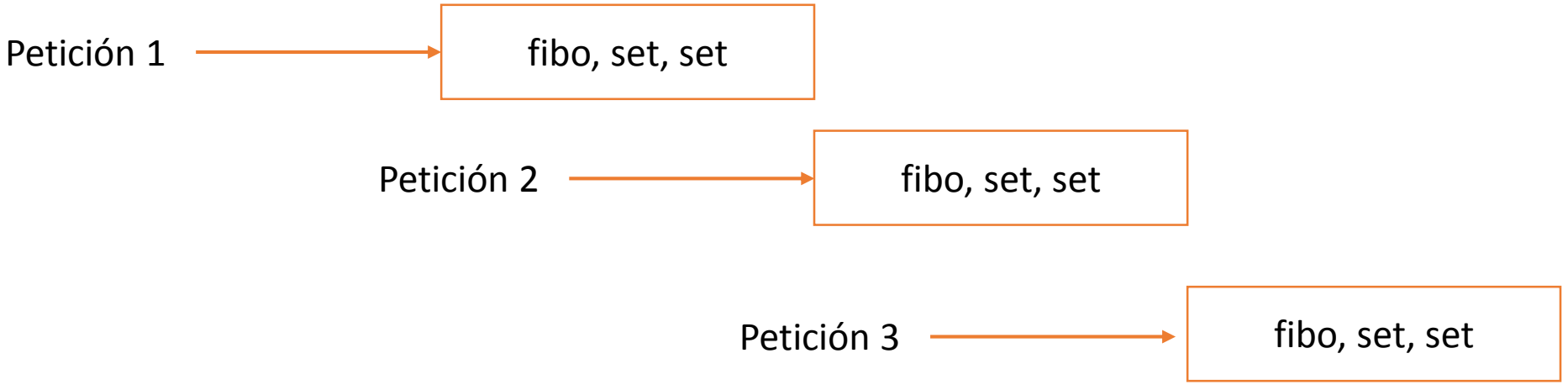
```
synchronized (this) {
  … región crítica …
}
```

- También aplicable a métodos

```
public synchronized void process(String mensaje) {
  … Todo el método se ejecuta en exclusión mutua …
}
```

Servidor con cache - Locks

- Solución sencilla
  - Método sincronizado
- ¿Cuál es el problema?



```
public static class MyServlet {
  @GuardedBy(“this”)
  private int ultimoNumero = -1;
  @GuardedBy(“this”)
  private int ultimoResultado = -1;

  public void process(String mensaje) {
   if (mensaje.contains(“fibonacci")) {
    int numero = extraer(mensaje);
    int resultado = -1;
    synchronized(this) {
      if (ultimoNumero == numero) {
       resultado = ultimoResultado;
      }
    }
    …
```

La parte más costosa se hace en paralelo

```
if (resultado == -1){
   resultado = fibonacci(numero);
   synchronized(this) {
     ultimoResultado = resutado;
     ultimoNumero = numero;
   }
}
```

Locks

- El objetivo de los mecanismos de sincronización (locking) es:
  - Preservar el invariante de las clases en presencia de concurrencia
  - Todas las variables que participan en un invariante de dado deben protegerse con el mismo *lock*
- ¿Es suficiente con sincronizar todos los métodos de la clase?
  - Supongamos un objeto vector que es thread-safe
  - Operación “put-if-absent” que combina dos métodos sincronizados

```
if (!vector.contains(elemento))
  vector.add(elemento);
```

- Necesario sincronizar cualquier camino en el código que pueda mutar el estado de un objeto

ReadWriteLock

- Permite que varios hilos lean un recurso, pero solo uno puede escribirlo
- Mantiene dos locks

```
ReadWriteLock rwLock = new ReentrantReadWriteLock();
rwLock.readLock().lock();
// Varios lectores pueden entrar si el lock
// no ha sido adquirido para escriture y no
// hay threads que quieran entrar para escribir
rwLock.readLock().unlock();

rwLock.writeLock().lock();
// un solo hilo que vaya escribir puede entrar,
// y solo si no hay hilos leyendo
rwLock.writeLock().unlock();
```

ReadWriteLock

- Pregunta
  - ¿Cómo podríamos usarlo en el servidor?
  - Pensar qué mensajes requieren lectura y cuáles escritura

Visibilidad

- Programación secuencial:
  - Lectura, Escritura, Lectura
- Programación multi-hilo:
  - Escrituras y lecturas en hilos diferentes pueden dar lugar a resultados inesperados
- No hay garantías de que el valor V_1 escrito por un hilo T_1 sea visible (puede leerse el valor) para otro hilo T_2 inmediatamente.
  - Es necesario usar sincronización para garantizar esto

Visibilidad

```
public class NoVisibility {
  private static boolean ready;
  private static int number;
  private static class ReaderThread extends Thread {
    public void run() {
      while (!ready)
        Thread.yield();
        System.out.println(number);
      }
  }
  public static void main(String[] args) {
    new ReaderThread().start();
    number = 42;
    ready = true;
  } }
```

Puede:

- Terminar (42)
- Terminar (0) – Reordenación
- No terminar – Visibilidad

Visibilidad

- ¿Por qué no es thread safe?
  - Porque dos hilos diferentes pueden observar valores diferentes de value

```
public class MutableInteger {
  private int value;
  public int get() { return value; }
  public void set(int value) { this.value = value; }
}
```

Visibilidad

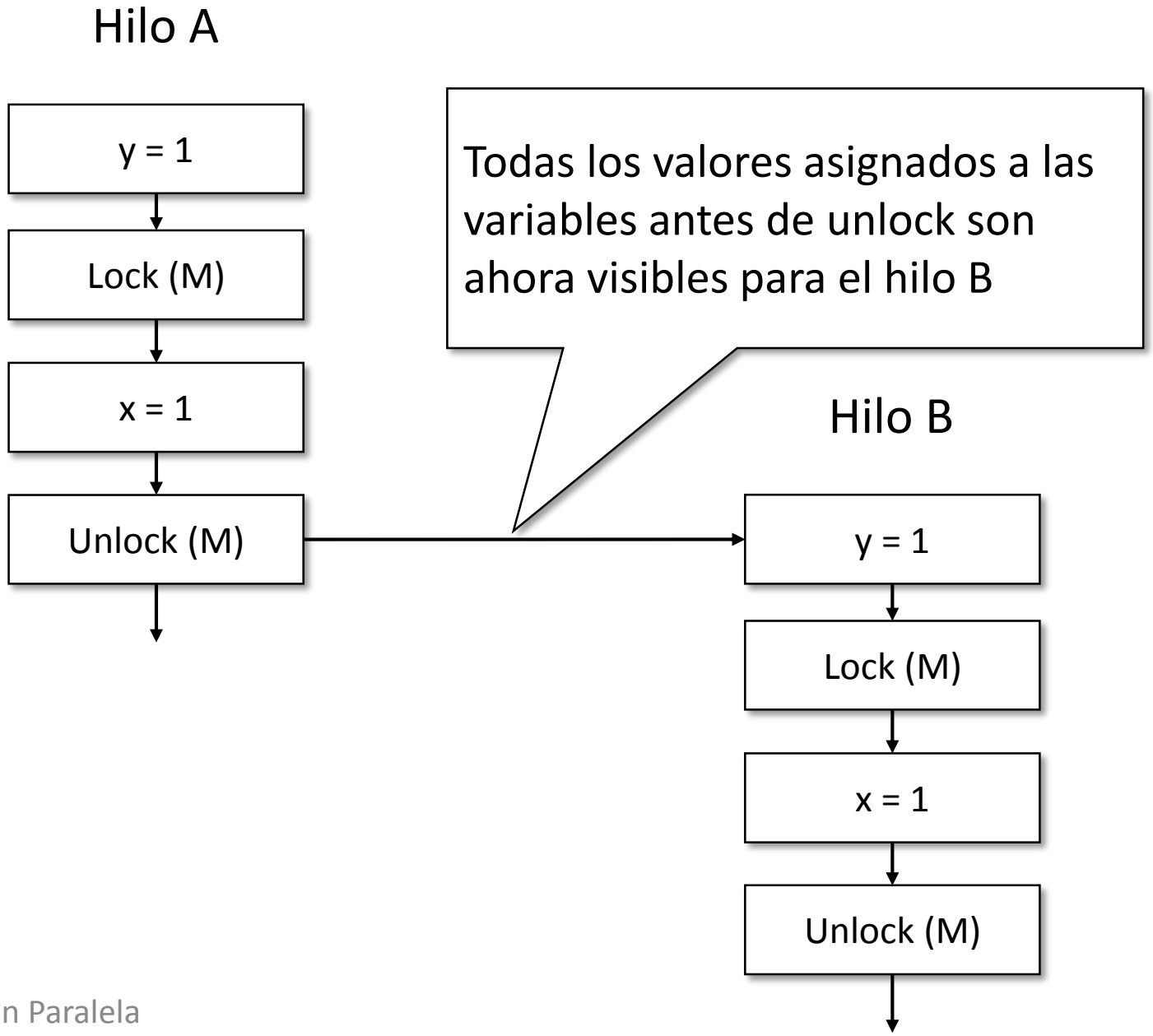
- ¿Por qué no es thread safe?
  - Porque dos hilos diferentes pueden observar valores diferentes de value

```
public class MutableInteger {
  private int value;
  public int get() { return value; }
  public void set(int value) { this.value = value; }
}
```

synchronized

Produce el efecto de hacer visible value tras terminar el lock

Visibilidad



Visibilidad – Volatile

- Modificador volatile para indicar variable compartida y que las actualizaciones se hagan de manera “predecible”
  - Las operaciones de lectura y escritura no se pueden reordenar
  - La variable no se puede almacenar en registros

- Diferencia con un lock:
  - Un lock garantizar visibilidad y atomicidad
  - Una variable volatile sólo garantiza visibilidad

Anotaciones

- @ThreadSafe
- @NotThreadSafe
- @GuardedBy(“nombre-variable-lock”)

java.util.concurrent

Introducción

- Partes
  - Executor framework
  - Sincronización
  - Colecciones concurrentes
  - Locks
  - Variables atómicas
  - Fork/Join

- Paquetes
  - java.util.concurrent
    - Tipos más communes, como thread pools, semáforos, bareras, mapas concurrentes
  - java.util.concurrent.atomic
    - Programación thread-safe sin usar locks, a través de variables atómicas
  - java.util.concurrent.locks
    - Diferentes tipos y utilidades de locks

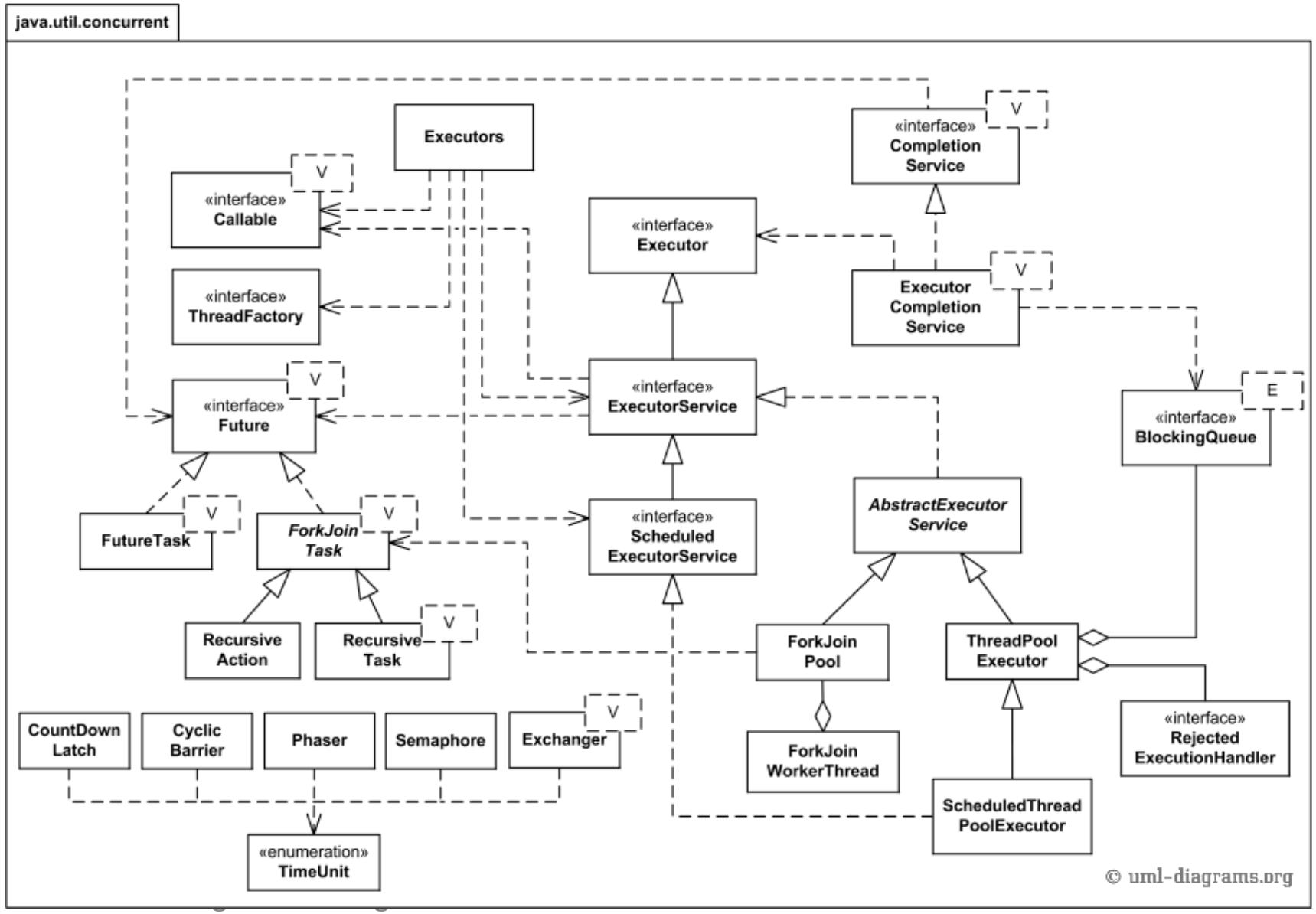
Executor framework

- Ejecución de tareas (ejemplo del servidor)
  - Secuencial: bajo rendimiento y aprovechamiento de recursos
  - Hilo por cada tarea: mala gestión de los recursos => poca escalabilidad

- Usar un “Thread pool”
  - Gestor de hilos que encola tareas

- Framework dedicado a la gestión de tareas
  - Desacopla el envío de tareas de su ejecución
  - Permite cambiar la política de ejecución
  - Productor – Consumidor:
    - Partes del programa que envían un tarea (productores)
    - Hilos que ejecutan las tareas (consumidores)

Executor framework



- Executor
  - Unidad básica ejecución tareas
- ExecutorService
  - Gestiona la ejecución de tareas
- Executors
  - Utilidades para crear ExecutorService

Servidor con Executor

ServidorExecutor.java

```
final int threads = 10;
final Executor executor =
    Executors.newFixedThreadPool(threads);

try (ServerSocket socket = new ServerSocket(9000)) {
  while (true) {
    final Socket s = socket.accept();
    Runnable r = new Runnable() {
        @Override
        public void run() {
            …
        }
    };
    executor.execute(r);
}
```

Se ejecutan en paralelo hasta 10 hilos. Se bloquea si se excede.

Thread pools

- Componentes
  - Una bolsa de hilos (worker threads)
  - Una cola de tareas (work queue)
- Funcionamiento
  - Cada hilo pide a la cola la siguiente tarea a realizar
- Ventajas
  - Amortizar el coste de crear hilos y finalizador (los hilos se reutilizan)
  - Mejorar la respuesta del sistema porque siempre habrá hilos esperando trabajo

Thread pools - Tipos

- Executors.newFixedThreadPool
  - Número máximo de hilos
  - Se crean nuevos hilos conforme se envían
  - Si un hilo muere, se vuelve a crear
- Executors. newCachedThreadPool
  - No hay límite de hilos
  - Trata de reutilizar hilos existentes para que haya una buena respuesta
  - Hilos que no se han usado en 60 segundos se eliminan
- Executors. newSingleThreadExecutor
  - Un solo hilo para procesar tareas secuencialmente
  - Se reemplazar por otro nuevo si muere
- Executors. newScheduledThreadPool
  - Ejecuta tareas periódicamente o con cierto retardo

ExecutorService – Gestión de hilos

```
public interface ExecutorService extends Executor {
  void shutdown();
  List<Runnable> shutdownNow();
  boolean isShutdown();
  boolean isTerminated();
  boolean awaitTermination(long timeout, TimeUnit unit)
      throws InterruptedException;
  ...
}
```

ExecutorService – Gestión de hilos

- shutdown
  - No se aceptan nuevas tareas
  - Las que se están ejecutando y las que quedan por ejecutar se terminan de ejecutar
- shutdownNow
  - Se acaba forzosamente
  - Intenta cancelar los trabajos en ejecución y no empieza las tareas pendientes
- awaitTermination
  - Bloquea a la espera que se haya terminado

Tareas con resultado

- Runnable no devuelve resultado, pero puede tener efector laterales
- Callable es similar a Runnable pero devuelve un resultado

```
@FunctionalInterface
public interface Callable<V> {
   V call() throws Exception;
}
```

Tareas con resultado

```
public interface ExecutorService extends Executor {

    <T> Future<T> submit(Callable<T> task);

    <T> Future<T> submit(Runnable task, T result);

    Future<?> submit(Runnable task);
}
```

Tareas con resultado

```
public interface ExecutorService extends Executor {
  <T> List<Future<T>>
  invokeAll(Collection<? extends Callable<T>> tasks)

  <T> List<Future<T>>
  invokeAll(Collection<? extends Callable<T>> tasks,
            long timeout, TimeUnit unit)

  <T> T
  invokeAny(Collection<? extends Callable<T>> tasks)

  <T> T
  invokeAny(Collection<? extends Callable<T>> tasks,
                    long timeout, TimeUnit unit)
}
```

FibonacciExample.java

Future

- Representa el resultado de una computación asíncrona
- Manejo de computaciones asíncronas
  - Se ejecuta un cálculo pero la llamada vuelve automáticamente

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get()
     throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit)
     throws InterruptedException,
            ExecutionException, TimeoutException;
}
```

Ejercicio

- Extender servidor la funcionalidad:
  - Mensaje para invocar a fibonacci
  - Mensaje para consultar el estado

Parallel streams

- Un **stream** es una secuencia de elementos, emitidos por una fuente
  - Un stream no guarda elementos, se calculan bajo demanda
  - Un stream es independiente de la fuente, e.g., colecciones, arrays, base de datos
  - Sobre un stream se pueden aplicar operaciones
    - filter, map, reduce, find, match, sorted, etc
  - Los streams se pueden combinar en forma de pipeline
    - Evaluación perezosa y corto-circuito
  - Iteración interna a través de funciones lamda

Parallel streams

- Operaciones sobre streams en paralelo
  - Misma API que Java Streams
- Ejemplo:

```
List<Long> nums = LongStream.rangeClosed(0, 1000).boxed()
      .collect(Collectors.toList());
nums.parallelStream().reduce(0L, Long:sum)).get()
```

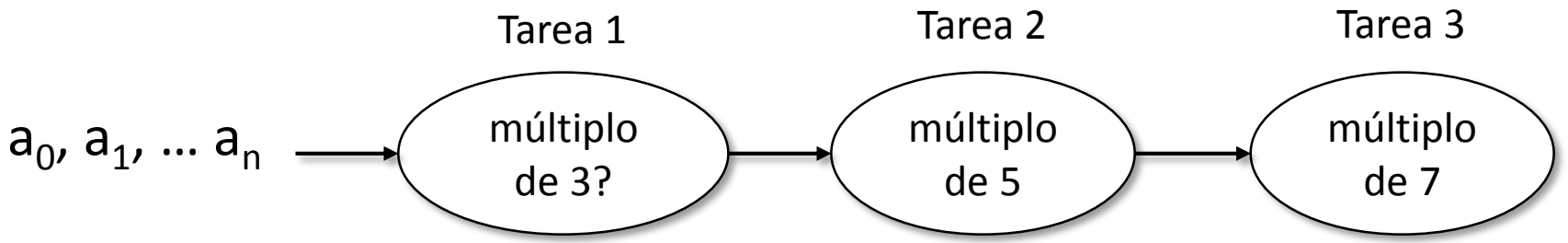
Parallel streams

- Establecer el número de hilos manualmente

```
ForkJoinPool miThreadPool = new ForkJoinPool(4);
long actualTotal = miThreadPool.submit(
  () -> nums.parallelStream().reduce(0L, Long::sum)).get();
```

Parallel Stream

- Ejercicio
  - Implementar pipeline para calcular los números de una lista que son múltiplos de ciertos números primos



Ejercicio

- Contador de palabras
- Implementar de diferentes maneras
  - Hilos
  - Executor framework
  - Parallel stream

```
for (File file : files) {
  count += Utils.countWords(file);
}
```

Bibliografía

- Diapositivas basadas en:
  - Java Concurrency in Practice. Brian Goetz. http://jcip.net/