

# 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

- 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

```
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

```
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

```
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 este 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 = resultado;  
                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 ...  
}
```

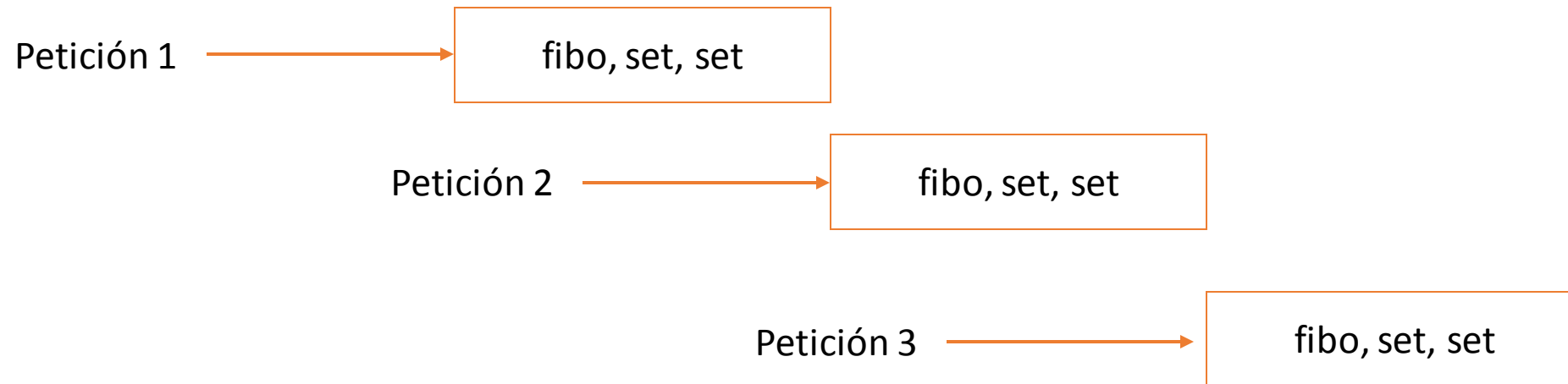
```
synchronized (this) {  
    ... región crítica ...  
}
```

- 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 = resultado;  
        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 escritura 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();
```

# 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

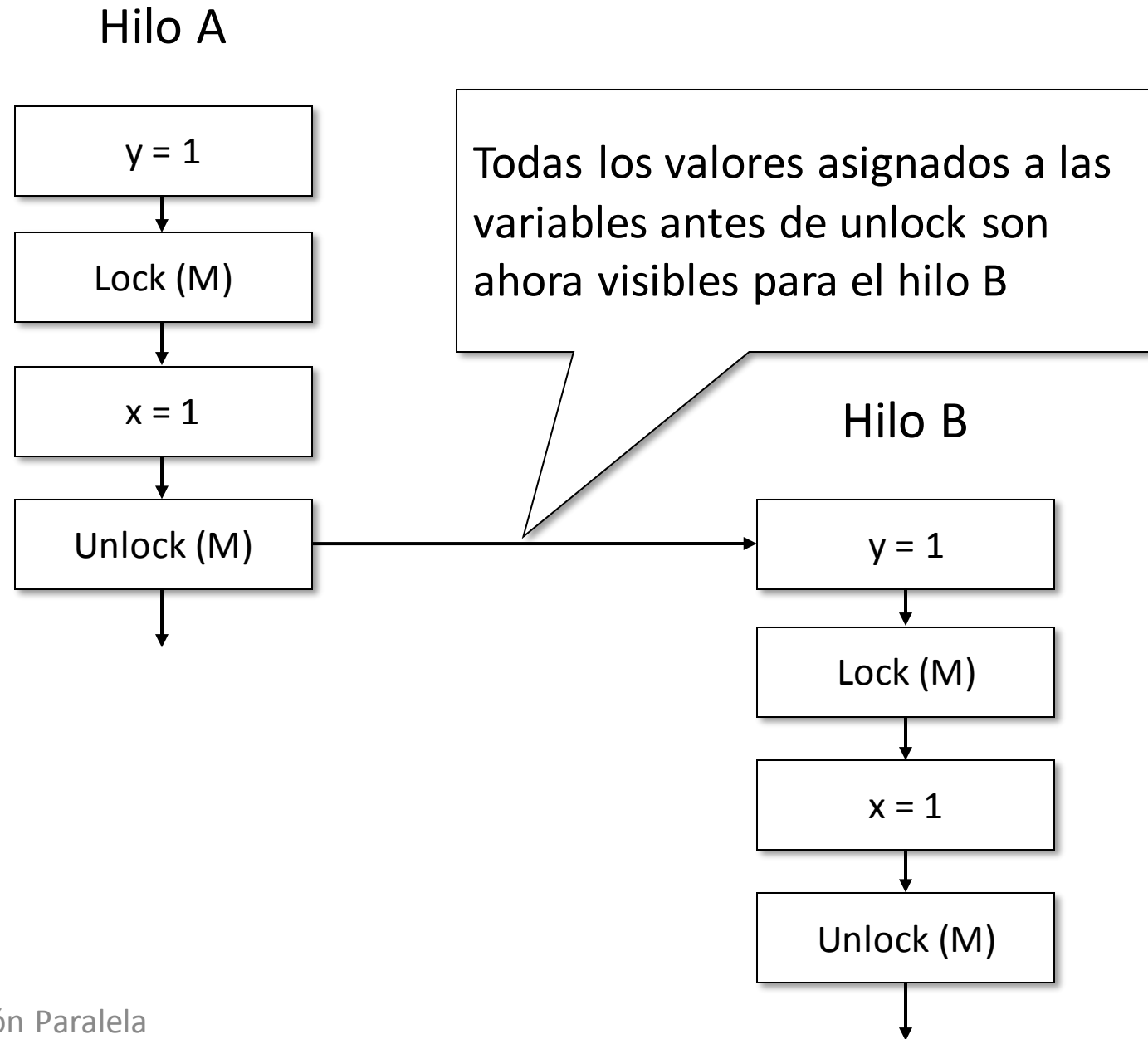
```
public class MutableInteger {  
    private int value;  
    public int get() { return value; }  
    public void set(int value) { this.value = value; }  
}
```

**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")`

# GUIs interactivas

# Aplicaciones Swing

- Hilo principal (el que ejecuta main).
- Hilo de eventos (Event Dispatch Thread).
  - Ejecuta los eventos de la GUI (eventos Swing)
- Hilos en segundo plano
  - Se encargan de ejecutar tareas costosas

# Hilo principal

```
SwingUtilities.invokeLater(new Runnable() {  
    public void run() {  
        createAndShowGUI();  
    }  
});
```



Crea la GUI en el  
hilo de eventos

# Hilo de eventos

- La mayoría de métodos de las clases de Swing no son *thread safe*
    - No se deben invocar desde otros hilos
    - Puede provocar inconsistencias (¡difícil de depurar!)
  - Los callbacks de los listeners (ej., **ActionListener.actionPerformed**) se ejecutan en el hilo de eventos.
- ! Si un callback puede tardar mucho tiempo, bloqueará el hilo de eventos.  
¿Cómo se podrían ejecutar tareas en segundo plano?
- ! Si se ejecuta un hilo en background, ¿cómo actualizar el hilo de eventos?

# Hilo de eventos

- `SwingUtilities.isEventDispatchThread`
- `SwingUtilities.invokeLater`
  - Encola un `Runnable` para ser ejecutado en el hilo de eventos
- `SwingUtilities.invokeAndWait`
  - Encola un `Runnable` y espera a que sea procesado por el hilo de eventos

# Tareas en background

- **SwingWorker**
  - Clase abstracta que proporciona funcionalidad de ejecución en segundo plano y comunicación de valores al hilo de eventos.

Resultado

Barra de  
progreso

```
public class Tarea extends SwingWorker<Integer, Void> {  
    @Override  
    protected Integer doInBackground() {  
        int i = 0;  
        Random r = new Random();  
        while (r.nextInt(100) != 38) i++;  
        return i;  
    }  
}
```

# SwingWorker

- `doInBackground()`
  - Realiza la tarea y devuelve un resultado
  - Se ejecuta en un hilo en segundo plano
- `done()`
  - Se ejecuta cuando el hilo en segundo plano ha terminado su ejecución
  - Se puede obtener el resultado con `get()`
  - Se ejecuta en el hilo de eventos
- `execute()`
  - Programa la ejecución de la tarea
- `publish(T valores...)`
  - Se invoca desde `doInBackground` para notificar que se desean comunicar valores intermedios al hilo de eventos
- `process(List<V> valores)`
  - Obtiene los valores de `publish()`
  - Se ejecuta en el hilo de eventos



# SwingWorker

```
protected Item doInBackground() {  
    int i = 0;  
    Random r = new Random();  
    while (r.nextInt(100) != 38) {  
        if (i % 10 == 0) publish(i);  
    }  
    return i;  
}
```

Mostrar el número de intentos



```
protected void process(List<Integer> valores) {  
    jLabel1.setText(valores.get(valores.size() - 1) + "");  
}
```

# SwingWorker

```
protected Item doInBackground() {  
    int i = 0;  
    Random r = new Random();  
    while (r.nextInt(100) != 38) {  
        if (i % 10 == 0) publish(i);  
    }  
    return i;  
}  
  
protected void done() {  
    int intentos = get();  
    jLabel.setText(intentos + "");  
}
```

Mostrar el número de  
intentos final



# SwingWorker

- `cancel()`
  - Para pedir que se cancele el trabajo
- `isCancelled()`
  - Para ser consultado en `doInBackground`
  - Devuelve `true` si se invocó a `cancel`
- Es el programador el que debe gestionar cómo cancelar el proceso si `isCancelled() == true`.

# SwingWorker

```
protected Item doInBackground() {  
    int i = 0;  
    Random r = new Random();  
    while (r.nextInt(100) != 38) {  
        if (isCancelled())  
            return -1;  
        if (i % 10 == 0) publish(i);  
    }  
    return i;  
}  
}  
  
miWorker.cancel(true);
```

Pedir la cancelación  
(puede lanzar  
InterruptedException)

