.UBAfiuba
FACULTAD DE INGENIERÍA

Programación concurrente en Java

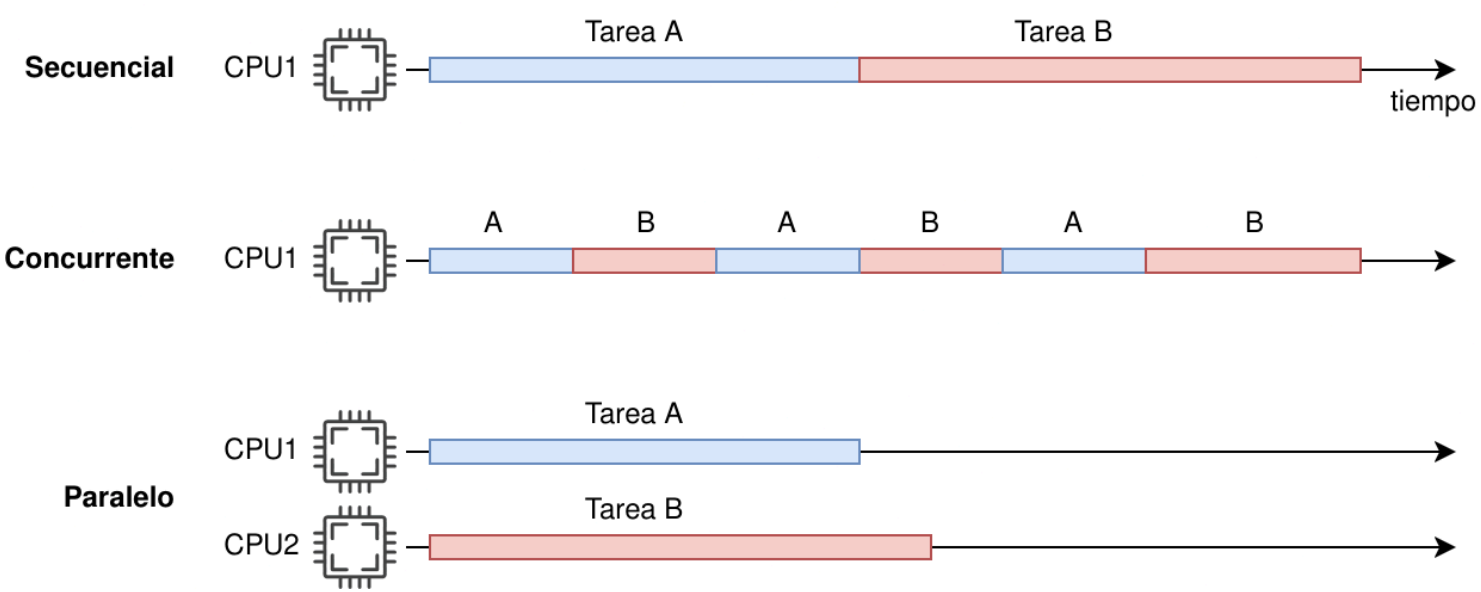
TB025 - Paradigmas de Programación

Curso Essaya

Concurrencia y Paralelismo

Concurrencia es la habilidad de lidiar con múltiples tareas que pueden estar en progreso al mismo tiempo (pero no necesariamente en ejecución).

Paralelismo es la habilidad de ejecutar múltiples tareas en forma simultánea. Requiere soporte de hardware (múltiples CPUs o cores).



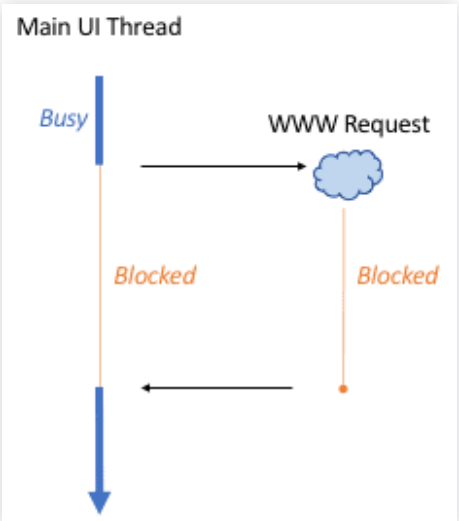
I/O bound vs CPU bound

Los programas generalmente se pueden categorizar en dos tipos:

> Un programa es **I/O bound** cuando hace uso intensivo de operaciones de entrada/salida.

Ejemplos: servidores, bases de datos, interfaces de usuario.

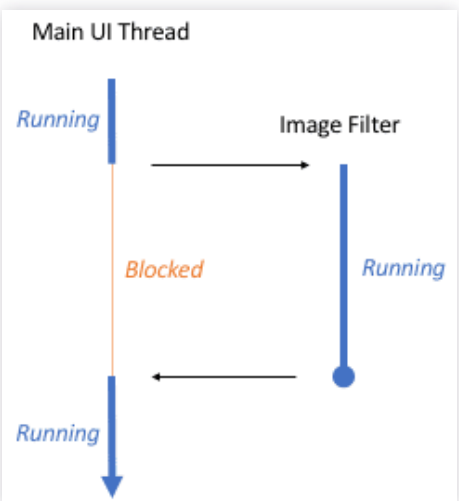
Suelen beneficiarse más de la **concurrencia**.



> Un programa es **CPU bound** cuando hace uso intensivo del CPU.

Ejemplos: simuladores, videojuegos, procesamiento de imágenes, criptografía, IA.

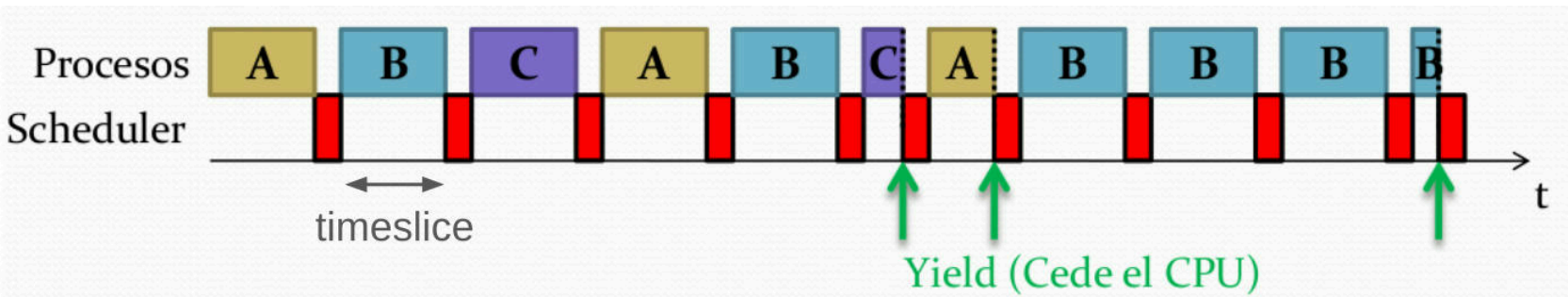
Suelen beneficiarse más del **paralelismo**.



Procesos

Son unidades de ejecución independientes y aisladas, con su propio espacio de memoria.

Son manejados por el sistema operativo. El **scheduler** (planificador) es el componente del SO que asigna tiempo de CPU a cada proceso (y en el caso de sistemas multicore, asigna los diferentes cores).



Para comunicarse entre sí, los procesos requieren mecanismos de IPC (*comunicación inter-proceso*) provistos por el SO, como **pipes** o **sockets**.

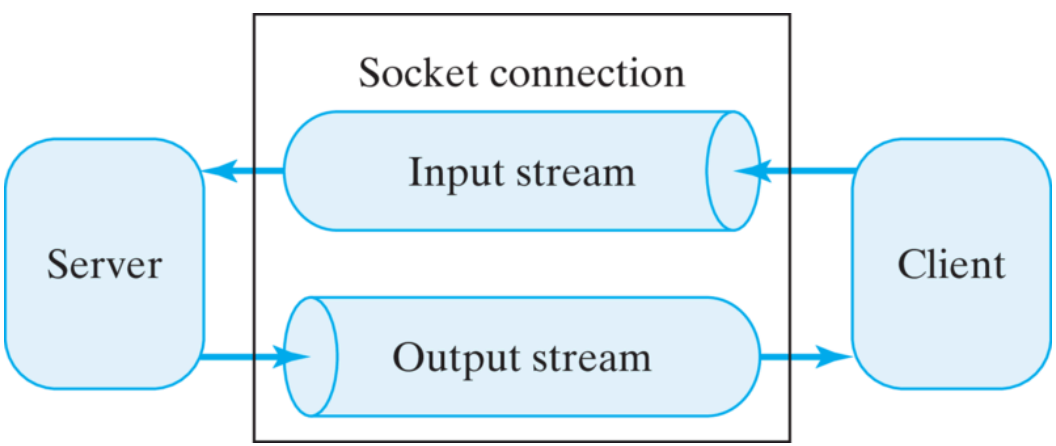
Ejemplo: sockets

Servidor:

```
try (ServerSocket serverSocket = new ServerSocket(1234)) {
    Socket clientSocket = serverSocket.accept();
    BufferedReader in = new BufferedReader(
        new InputStreamReader(clientSocket.getInputStream())
    );

    String line = in.readLine();
    // ...
}
```

Cliente:

```
try (Socket clientSocket = new Socket("localhost", 1234)) {
    PrintWriter out = new PrintWriter(
        clientSocket.getOutputStream(),
        true
    );
    out.println("hola");
}
```

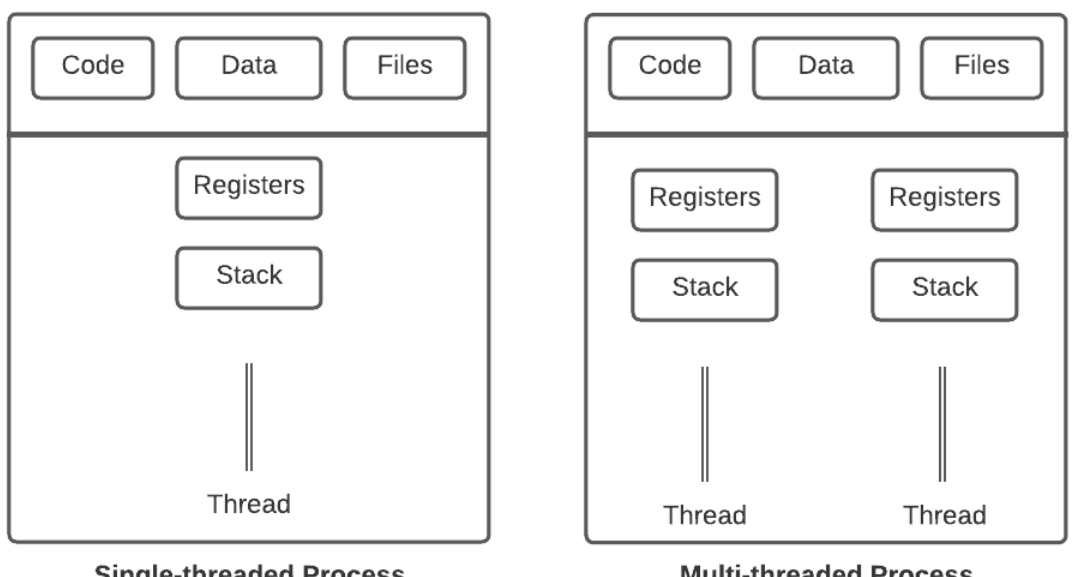


Hilos

Son unidades de ejecución dentro de un proceso.

Comparten el mismo espacio de memoria.

Pueden ser manejados por el sistema operativo o por la propia aplicación (dependiendo del lenguaje y la biblioteca de threads utilizada).



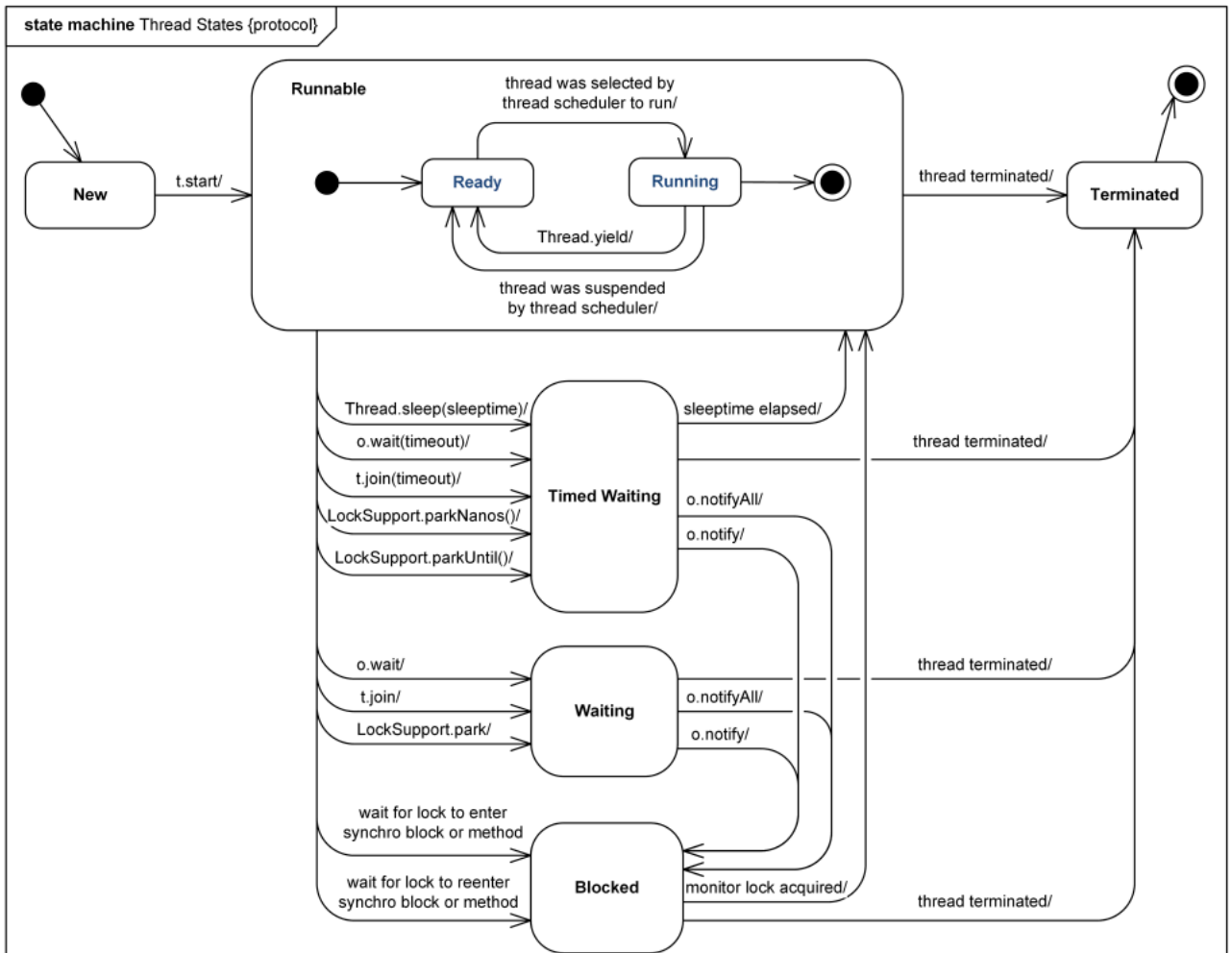
Hilos en Java

La clase `Thread` representa un hilo de ejecución en Java.

```
public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(() -> {
            System.out.println("Hola desde el hilo!");
        });
        t.start(); // Inicia el hilo

        System.out.println("Hola desde el hilo principal!");
    }
}
```

El proceso finaliza cuando no queda ningún hilo de tipo *non-daemon* en ejecución.

Estados de los hilos de Java



Sincronización usando *interrupt* y *join*

```java
public class Main {
    static void mostrar(String mensaje) {
        System.out.format("%s: %s%n",
                Thread.currentThread().getName(),
                mensaje);
    }

    static void martinFierro() {
        String[] mensajes = {
                "Aquí me pongo a cantar",
                "Al compás de la vigüela",
                "Que al hombre que lo desvela",
                "Una pena estrordinaria,",
                "Como la ave solitaria",
                "Con el cantar se consuela."
        };
        try {
            for (String mensaje : mensajes) {
                Thread.sleep(4000);
                mostrar(mensaje);
            }
        } catch (InterruptedException e) {
            mostrar("No terminé!");
        }
    }
    public static void main(String[] args) throws InterruptedException {
        long paciencia = 10_000; // milisegundos

        mostrar("Lanzando el hilo");
        long inicio = System.currentTimeMillis();
        Thread t = new Thread(Main::martinFierro);
        t.start();

        mostrar("Esperando a que el hilo termine");
        while (t.isAlive()) {
            mostrar("Sigo esperando...");
            if ((System.currentTimeMillis() - inicio) > paciencia) {
                mostrar("Me cansé de esperar!");
                t.interrupt();
            }
            t.join(1000);
        }
        mostrar("Al fin!");
    }
}
```

Ejercicio

```java
public class Main {
    static int a = 0;

    static void incThread() {
        for (int i = 0; i < 100000; i++) {
            a++;
        }
    }

    static void decThread() {
        for (int i = 0; i < 100000; i++) {
            a--;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(Main::incThread);
        Thread t2 = new Thread(Main::decThread);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(a); // ¿Qué valor imprime?
    }
}
```

Condiciones de carrera

Una **condición de carrera** ocurre cuando el orden de las operaciones realizadas por dos o más hilos afecta el resultado final, a menudo provocando bugs.

```
x = 5;

// en thread 1:
x++;

// en thread 2:
x++;
```

Time	Thread 1	Thread 2
1	load x into register (5)	
2	add 1 to register (6)	
3	store register in x (6)	
4		load x into register (6)
5		add 1 to register (7)
6		store register in x (7)

Time	Thread 1	Thread 2
1	load x into register (5)	
2	add 1 to register (6)	
3		load x into register (5)
4	store register in x (6)	
5		add 1 to register (6)
6		store register in x (6)

Las condiciones de carrera son solo una de las posibles causas de bugs en programas concurrentes.

La JVM define un **modelo de memoria** que especifica cómo y cuándo los cambios hechos por un hilo son visibles para otros hilos.

Una función o objeto es **thread-safe** si puede ser utilizado por múltiples hilos sin condiciones de carrera.

Sincronización: Candados

Un **candado** (**lock** / **mutex**) es un mecanismo de sincronización que permite a los hilos coordinar el acceso a recursos compartidos, previniendo condiciones de carrera.

```java
public class Main {
    static int a = 0;
    static Object candado;

    static void incThread() {
        for (int i = 0; i < 100000; i++) {
            synchronized(candado) {
                a++;
            }
        }
    }

    static void decThread() {
        for (int i = 0; i < 100000; i++) {
            synchronized(candado) {
                a--;
            }
        }
    }
}
```

Todos los mecanismos de sincronización en Java se construyen en base a un **candado intrínseco** (o **monitor**) asociado a cada objeto.

El entrar al bloque `synchronized(candado) { ... }` el hilo intenta **adquirir** el candado intrínseco del objeto `candado`.

- Si el candado está libre, el hilo lo adquiere y entra al bloque.
- Si el candado está ocupado por otro hilo, el hilo actual se **bloquea** hasta que el candado se libere.

Al finalizar el bloque, el hilo **libera** el candado.

Métodos sincronizados

```
public class ContadorSincronizado {
    private int c = 0;

    public void incrementar() {
        synchronized(this) {
            c++;
        }
    }

    public int valor() {
        synchronized(this) {
            return c;
        }
    }
}
```

Azúcar sintáctica:

```
public class ContadorSincronizado {
    private int c = 0;

    public synchronized void incrementar() {
        c++;
    }

    public synchronized int valor() {
        return c;
    }
}
```

Candados explícitos

También es posible utilizar candados explícitos provistos por la biblioteca estándar.

La interfaz `Lock` provee los métodos `lock()` y `unlock()`, ofreciendo un control más fino sobre la adquisición y liberación del candado.

```
public class ContadorSincronizado {
    private int c = 0;
    private final ReentrantLock lock = new ReentrantLock();

    public void incrementar() {
        lock.lock();
        c++;
        lock.unlock();
    }

    public int valor() {
        int v;
        lock.lock();
        v = c;
        lock.unlock();
        return v;
    }
}
```

Interbloqueos o Deadlocks

Un **deadlock** ocurre cuando dos o más hilos están bloqueados permanentemente, esperando por un recurso que nunca será liberado.

```
final Object lock1 = new Object();
final Object lock2 = new Object();

Thread hiloA = new Thread(() -> {
    synchronized (lock1) {
        System.out.println("Hilo A: Adquirió lock 1");
        try { Thread.sleep(100); } catch (InterruptedException e) {}
        System.out.println("Hilo A: Esperando por lock 2");
        synchronized (lock2) { System.out.println("Hilo A: Adquirió lock 2"); }
    }
});

Thread hiloB = new Thread(() -> {
    synchronized (lock2) {
        System.out.println("Hilo B: Adquirió lock 2");
        try { Thread.sleep(100); } catch (InterruptedException e) {}
        System.out.println("Hilo B: Esperando por lock 1");
        synchronized (lock1) { System.out.println("Hilo B: Adquirió lock 1"); }
    }
});

hiloA.start(); hiloB.start();
```

Esta situación es el caso más simple de un interbloqueo, donde múltiples hilos esperan indefinidamente debido a una dependencia cíclica.

Interbloqueos o Deadlocks (cont.)

Una posible solución: asegurar un orden global para adquirir múltiples candados.

```
final Object lock1 = new Object();
final Object lock2 = new Object();

Thread hiloA = new Thread(() -> {
    synchronized (lock1) {
        System.out.println("Hilo A: Adquirió lock 1");
        try { Thread.sleep(100); } catch (InterruptedException e) {}
        System.out.println("Hilo A: Esperando por lock 2");
        synchronized (lock2) { System.out.println("Hilo A: Adquirió lock 2"); }
    }
});

Thread hiloB = new Thread(() -> {
    synchronized (lock1) { // Cambiado lock2 por lock1
        System.out.println("Hilo B: Adquirió lock 1");
        try { Thread.sleep(100); } catch (InterruptedException e) {}
        System.out.println("Hilo B: Esperando por lock 2");
        synchronized (lock2) { System.out.println("Hilo B: Adquirió lock 2"); }
    }
});

hiloA.start(); hiloB.start();
```

Variables de condición

Una **variable de condición** permite a un hilo suspender su ejecución hasta que se cumpla una condición específica.

El método `wait` de la clase `Object`:

1. Libera el candado intrínseco del objeto (se asume que está adquirido).
2. Bloquea el hilo hasta que sea *notificado* o *interrumpido* desde otro hilo.
3. Cuando el hilo se despierta, vuelve a adquirir el candado intrínseco.

El método `notify` despierta a uno de los hilos que están esperando en el objeto (si hay alguno).

El método `notifyAll` despierta a todos los hilos que están esperando en el objeto (si hay alguno).

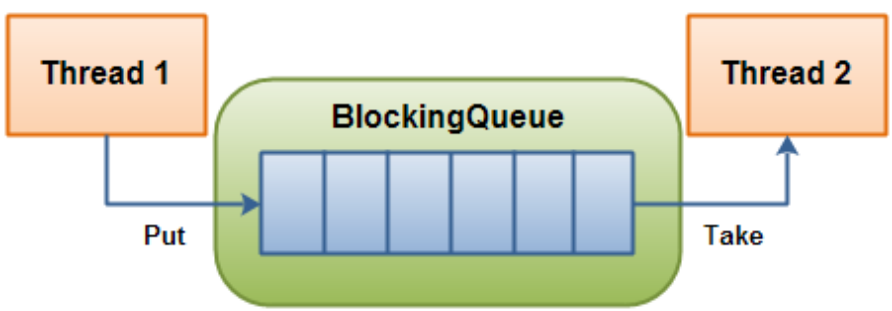
```java
public class CasillaDeCorreo {
    private Object paquete;
    private boolean vacio;

    public synchronized void depositar(Object p) throws InterruptedException {
        while (!vacio) {
            wait(); // Bloquea el hilo hasta que la casilla esté vacía
        }
        paquete = p;
        vacio = false;
        notify(); // Despierta a uno de los hilos que esperan
    }

    public synchronized Object retirar() throws InterruptedException {
        while (vacio) {
            wait(); // Bloquea el hilo hasta que haya un paquete
        }
        Object p = paquete;
        paquete = null;
        vacio = true;
        notify(); // Despierta a uno de los hilos que esperan
        return p;
    }
}
```

Colecciones concurrentes

La biblioteca estándar de Java provee varias colecciones diseñadas para ser usadas en entornos concurrentes, evitando la necesidad de sincronización manual.

Algunos ejemplos:

- `CopyOnWriteArrayList`: Una lista que crea una copia del arreglo subyacente en cada operación de escritura. Ideal para escenarios con muchas lecturas y pocas escrituras.
- `ConcurrentHashMap`: Un mapa hash que permite acceso concurrente eficiente. Divide el mapa en segmentos para minimizar la contención entre hilos.
- `BlockingQueue`: Una cola que soporta operaciones que esperan hasta que la cola no esté vacía al retirar elementos, o hasta que haya espacio disponible al insertar elementos.



Variables atómicas

Una **transacción atómica** es una serie **indivisible** e **irreducible** de operaciones, de modo que **o todas ocurren, o ninguna ocurre**.

La JVM garantiza que algunas operaciones son atómicas (por ejemplo, la lectura y escritura de referencias y algunos tipos primitivos).

Sin embargo, operaciones más complejas (como `x++`) no son atómicas, ya que involucran múltiples pasos.

La biblioteca estándar de Java provee clases en el paquete `java.util.concurrent.atomic` que permiten operaciones atómicas sobre variables, evitando la necesidad de sincronización explícita.

```
AtomicInteger c = new AtomicInteger(0);
c.incrementAndGet(); // Operación atómica
```

Computación asíncrona

Una función **sincrónica** bloquea el hilo que la invoca hasta que la operación finaliza, mientras que una función **asíncrona** retorna inmediatamente, permitiendo que el hilo invocante continúe su ejecución.

La clase `Future` representa el resultado de una computación asíncrona.

```
long sumar(long[] array) {
    long r = 0;
    for (long v : array) {
        r += v;
    }
    return r;
}

Future<Long> sumarAsincronico(long[] array) {
    CompletableFuture<Long> r = new CompletableFuture<>();
    Thread t = new Thread(() -> {
        long suma = sumar(array);
        r.complete(suma);
    });
    t.start();
    return r;
}
```

```
Future<Long> f = sumarAsincronico(array);
// Hacer otras cosas...
long resultado = f.get(); // Bloquea hasta que el resultado esté
listo
```

Ejecutores

La interfaz `ExecutorService` define un mecanismo genérico para gestionar un grupo de hilos para ejecutar computaciones asíncronas.

```
long sumar(long[] array) {
    long r = 0;
    for (long v : array) {
        r += v;
    }
    return r;
}

// En cualquier momento, como máximo 4 hilos estarán activos procesando
// tareas. Si se envían tareas adicionales cuando todos los hilos están
// activos, esperarán en la cola hasta que un hilo esté disponible.
ExecutorService executor = Executors.newFixedThreadPool(4);

Future<Long> sumarAsincronico(long[] array) {
    return executor.submit(() -> sumar(array));
}
```

```
Future<Long> f = sumarAsincronico(array);
// Hacer otras cosas...
long resultado = f.get();
```

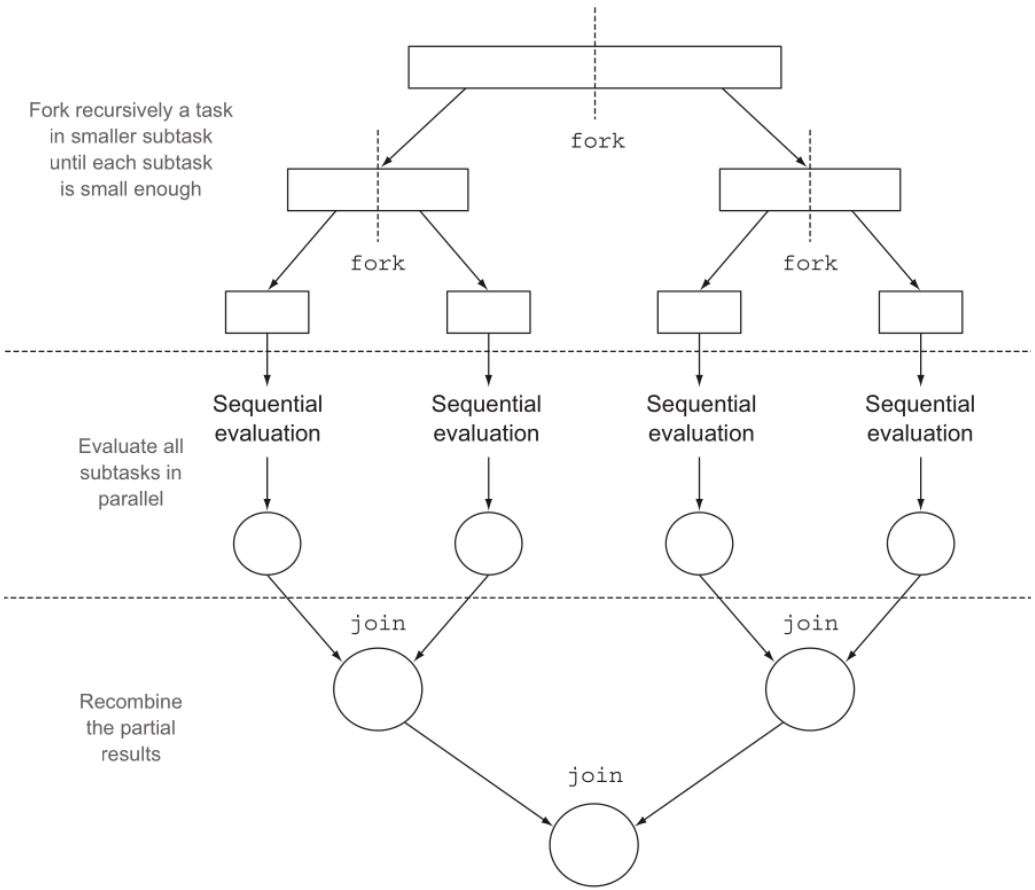
Patrón Fork/Join

Es una estrategia inspirada en *divide & conquer* para paralelizar tareas.

La clase `ForkJoinPool` es una implementación de la interfaz `ExecutorService` que automáticamente aplica esta estrategia.

Ejemplo:

```
class SortTask extends RecursiveAction {...}

var executor = new ForkJoinPool();
executor.execute(new SortTask(array));
```



www.**ingenieria**.uba.ar

/ingenieriauba

/FIUBAoficial