1. Programas, Procesos, Servicios e Hilos

Programa

Un programa es un elemento estatico. Es un conjunto de instrucciones escritas en un lenguaje de programación. Solo describe qué hacer, pero no realiza acciones hasta que se ejecuta.

Proceso

Un proceso es una instancia en ejecución de un programa.
Al iniciar un programa, el sistema operativo crea un proceso que tiene sus propios recursos (como memoria, contador de programa y registros) y puede ejecutarse de forma independiente de otras instancias.
Cada proceso es aislado, es decir, no comparte memoria con otros procesos, y el sistema operativo se encarga de su administración.

Sistema Operativo.

El sistema operativo es el encargado de la gestión de procesos. Los crea, los elimina y provee los instrumentos que permiten la ejecución y también la comunicación entre ellos.

Servicio y Daemon

Un servicio es un tipo especial de proceso que opera en segundo plano, es decir, sin interactuar directamente con el usuario.
En Windows, se llaman "servicios", y en Linux se llaman "daemons".

Por ejemplo, el servicio de firewall o el antivirus son servicios en segundo plano que se ejecutan para proporcionar funciones específicas sin intervención del usuario.

Hilo (Thread)

Un hilo es la unidad de ejecución dentro de un proceso.
A diferencia de los procesos, los hilos pueden compartir el mismo espacio de memoria, lo cual es eficiente en cuanto a recursos.
Sin embargo, como los hilos pueden acceder a los mismos datos, puede haber conflictos si no se manejan correctamente. Los hilos se dividen en dos niveles:

- **Hilos de usuario:** se crean en aplicaciones y se gestionan a través del lenguaje de programación (como Java).
- **Hilos de sistema:** creados por el sistema operativo para optimizar la ejecución de los hilos de usuario.

2. Estados de un Proceso

Los procesos cambian de estado según su interacción con el procesador y otros recursos del sistema. Los principales estados son:

- **Nuevo:** El proceso acaba de ser creado.
- **Listo:** El proceso está preparado para ejecutarse y espera su turno.
- **En ejecución:** El proceso está utilizando el procesador activamente.
- **Bloqueado:** El proceso está esperando que ocurra un evento externo, como la finalización de una operación de entrada/salida.
- **Terminado:** El proceso ha finalizado, y el sistema liberará los recursos que estaba utilizando.

3. Multiproceso y Multiprogramación

- **Multiproceso:** Es la capacidad de un sistema de tener múltiples procesadores o núcleos, lo que permite ejecutar varios procesos simultáneamente.

- **Multiprogramación:** Es la habilidad de un sistema monoprocesador para alternar rápidamente entre varios procesos, simulando ejecución simultánea, lo que maximiza el uso del procesador.

4. Ejecución de Procesos en Java

Java proporciona clases como `ProcessBuilder` y `Process` en el paquete `java.lang` para la creación y gestión de procesos del sistema operativo:

- **ProcessBuilder**: Permite construir y ejecutar un proceso del sistema (por ejemplo, abrir la calculadora o ejecutar comandos en la consola).

Ejemplo de código:

```
Process pCalc = new ProcessBuilder("calc").start();
System.out.println("Lanzado proceso calculadora con java. PID: " +
pCalc.pid());
```

- **Runtime.exec()**: Permite ejecutar comandos directamente en el sistema operativo, similar a `ProcessBuilder`, aunque tiene menos opciones configurables.

Ejemplo de ejecución de comando `tasklist`:

```
String command = "tasklist";
Process pCommand = Runtime.getRuntime().exec(command);
BufferedReader reader = new BufferedReader(new
InputStreamReader(pCommand.getInputStream()));
```

Este código ejecuta el comando `tasklist` en Windows para mostrar los procesos activos.

5. Ejemplo Avanzado: Ejecutar un Proceso Java en un Subproceso

Usar `ProcessBuilder` para ejecutar una clase de Java como un subproceso. Este enfoque es útil cuando queremos lanzar una tarea en paralelo sin bloquear el proceso principal. Aquí hay un código de ejemplo:

Clase `SubProceso`

Esta clase realiza la suma de todos los números en un intervalo y muestra el resultado. El código obtiene los límites del intervalo de los argumentos (`args`) que recibe en su método `main`.

Clase `Main`

Esta clase usa `ProcessBuilder` para ejecutar **`SubProceso`** como un subproceso.

```
java.lang.ProcessBuilder pb = new java.lang.ProcessBuilder(
"java", "-cp",classPath, className, n1.toString(),
n2.toString()).redirectOutput(Redirect.INHERIT);
```

Esta línea crea un proceso que ejecuta la clase **`SubProceso`**, pasándole los límites del intervalo. `Redirect.INHERIT` permite que la salida del subproceso se muestre en la consola del proceso principal.

Resumen

El sistema operativo gestiona la ejecución de procesos y hilos para aprovechar al máximo los recursos del procesador. Java facilita la creación de procesos y subprocesos para realizar tareas paralelas y mejorar la eficiencia del sistema. Estos mecanismos son clave en aplicaciones que requieren realizar múltiples tareas a la vez o ejecutar comandos del sistema en segundo plano.

3. Concurrencia

La **concurrencia** permite que múltiples aplicaciones o procesos se ejecuten al mismo tiempo, lo cual ejecuta diferentes procesos simultáneamente o de manera intercalada, creando la ilusión de que ocurren a la vez.

1. **Multiprogramación y Programación Paralela**:
    - **Multiprogramación**: En sistemas con un solo procesador, los procesos se alternan de manera intercalada.
    - **Programación Paralela**: En sistemas con varios procesadores, los procesos pueden ejecutarse en paralelo, en distintos procesadores.

3.1 Ventajas

- **Mejora del rendimiento**: La concurrencia puede incrementar la velocidad de ejecución.

- **Flexibilidad y escalabilidad**: Permite una asignación dinámica de recursos y la posibilidad de aumentar la potencia del sistema añadiendo nuevos procesadores.

- **Redundancia y tolerancia a fallos**: Al tener múltiples procesadores, el fallo de uno no detiene el sistema.

3.2 Inconvenientes

- **Exclusión mutua**: Varios procesos no deben modificar simultáneamente una misma variable compartida para evitar inconsistencias de datos. Para esto, se emplea el concepto de **región crítica** y herramientas como semáforos y monitores.

- **Sincronización**: Los procesos deben coordinarse para evitar que uno dependa de un estado que el otro aún no ha alcanzado. La sincronización puede gestionarse mediante semáforos, eventos, o monitores.

4. Programación Paralela

La **programación paralela** se refiere a la concurrencia en sistemas multiprocesador, donde los procesos realmente se ejecutan al mismo tiempo en diferentes procesadores, lo cual disminuye el tiempo de ejecución total.

4.1 Ventajas

- **Ejecución simultánea**: Permite que múltiples tareas se ejecuten al mismo tiempo.
- **Reducción del tiempo total de ejecución**.
- **Resolución de problemas complejos**: Facilita el procesamiento de grandes cantidades de datos y operaciones complejas.
- **Uso de recursos remotos**: Se pueden aprovechar recursos distribuidos, como los de redes WAN o Internet.

4.2 Inconvenientes

- **Complejidad en el desarrollo**: Los programas paralelos son más difíciles de escribir y gestionar.
- **Consumo de energía**: Los sistemas paralelos suelen requerir un alto consumo energético.
- **Sincronización**: La comunicación y sincronización entre subtareas es esencial pero compleja.

---

5. Programación Distribuida

La **programación distribuida** es un tipo especial de programación paralela, donde los procesos se ejecutan en una red de computadoras independientes, que funcionan como un solo sistema desde la perspectiva del usuario.

1. **Sistemas distribuidos**:
    - Compuestos por varios ordenadores conectados en red que colaboran en tareas comunes.
    - Un ejemplo de sistema distribuido es una red de oficinas de un banco que comparte datos y procesos, lo cual permite a los usuarios realizar transacciones en cualquier sucursal.
2. **Modelos de Comunicación**:
    - **Sockets**: Puntos de conexión para la comunicación entre procesos.
    - **Llamada de procedimientos remotos (RPC)**: Permite que un programa cliente invoque procedimientos en un servidor remoto.
    - **Invocación remota de objetos (RMI)**: Los objetos en distintos procesos pueden invocar métodos entre sí; Java RMI es una implementación de este modelo.

5.1 Ventajas

- **Compartición de recursos y datos**: Facilita el acceso y la gestión de recursos distribuidos.
- **Escalabilidad**: Posibilidad de crecimiento incremental.
- **Alta disponibilidad y tolerancia a fallos**.
- **Flexibilidad**: Permite distribuir la carga de trabajo entre diferentes computadoras, mejorando el rendimiento general.

5.2 Inconvenientes

- **Complejidad**: Requiere software especializado para gestionar el sistema.
- **Problemas de red**: La pérdida de mensajes y la saturación de tráfico pueden afectar el rendimiento.
- **Seguridad**: Vulnerable a ataques como la denegación de servicio (DDoS), que pueden interrumpir el funcionamiento del sistema.

6. Hilos

La programación multihilo es un enfoque que permite ejecutar varias tareas de manera concurrente dentro de un mismo programa.
Cada hilo es una unidad de procesamiento dentro de un proceso que comparte los recursos de memoria. Los hilos pueden realizar diferentes tareas al mismo tiempo.

Conceptos clave:

1. **Hilos (Threads):** Un hilo es la unidad más pequeña de ejecución dentro de un proceso. Todos los procesos contienen al menos un hilo, pero pueden tener varios.
   Los hilos comparten los recursos del proceso, como la memoria, lo que permite que cualquier hilo modifique los datos que los demás hilos pueden ver.

2. **Gestión de hilos en Java:**
   - Cuando se arranca un programa Java, el hilo principal (el que ejecuta el método `main`) es el primero en ejecutarse. Este hilo puede crear otros hilos para realizar tareas simultáneas.
   -
   - Los hilos pueden ser gestionados directamente creando instancias de la clase `Thread` o utilizando la interfaz `Runnable`.

3. **Definir e iniciar hilos:**
   - **Usando la interfaz `Runnable`:** La interfaz define un método `run()`, el cual contiene el código que se ejecutará en el hilo. Un objeto `Runnable` es pasado al constructor de `Thread` y se inicia con el método `start()`.
   - **Usando la clase `Thread`:** Se puede crear una subclase de `Thread` y sobrescribir el método `run()`. Al igual que con `Runnable`, se llama a `start()` para iniciar el hilo.

4. **Control de la ejecución de los hilos:**
   - **`sleep()`:** Suspende la ejecución de un hilo por un tiempo determinado. Esto puede ser útil para simular retrasos o liberar el tiempo del procesador para otros hilos. **También se puede usar para sincronizar la ejecución de hilos.**

   - **Interrupciones (interrupt):** Permite interrumpir un hilo que está ejecutando un proceso. El hilo interrumpido puede manejar esta interrupción de manera que termine su ejecución de forma controlada.

   - **`join()`:** Permite que un hilo espere hasta que otro hilo termine su ejecución. Esto asegura que los hilos se ejecuten en un orden específico si es necesario.

5. **Prioridades de los hilos:**
   - Los hilos en Java tienen una prioridad que va de 1 (mínima) a 10 (máxima), y la prioridad por defecto es 5.

- Sin embargo, no se garantiza que un hilo de mayor prioridad se ejecute antes que otro de menor prioridad, ya que la gestión de prioridades depende del sistema operativo y la JVM.

6. **Hilos demonios (Daemon Threads):**
   - Son hilos de baja prioridad que ejecutan tareas en segundo plano, como el recolector de basura (garbage collector) en Java. Los hilos demonios se terminan automáticamente cuando no quedan más hilos de usuario activos.
   - Se puede convertir un hilo en demonio utilizando `setDaemon(true)`, pero esta propiedad no puede modificarse una vez el hilo ha sido iniciado.

Explicación detallada:

1. **Definir e iniciar hilos:** En Java, los hilos pueden ser definidos de dos formas:

   - Implementando la interfaz `Runnable`, lo que permite que un objeto se ejecute en su propio hilo. Esta es la forma más flexible, ya que un objeto `Runnable` puede heredar de cualquier clase.

   - Subclassificando la clase `Thread` y sobrescribiendo el método `run()`. Esto es más restrictivo porque obliga a heredar de la clase `Thread`, pero puede ser útil para tareas simples.

   Una vez que un hilo ha sido definido, se inicia con el método `start()`, que invoca internamente el método `run()`.

2. **Método `sleep()`:** Cuando se usa `sleep()`, el hilo actual suspende su ejecución por un tiempo determinado. Esto no implica que el hilo deje de consumir recursos; simplemente no hace nada durante ese período. Por ejemplo, si un hilo se detiene cada 3 segundos, otros hilos pueden ejecutarse durante ese tiempo.

3. **Interrupciones y método `interrupt()`:** Si se interrumpe un hilo, este puede manejar la interrupción (por ejemplo, finalizar su ejecución). **Si el hilo está en espera (por ejemplo, usando `sleep()`), lanzará una excepción `InterruptedException`.**

4. **El método `join()`:** Si un hilo quiere esperar a que otro termine su ejecución antes de continuar, puede utilizar `join()`. Esto puede ser útil para garantizar que ciertos hilos terminen antes que otros en un programa que dependa del orden de ejecución.
5. **Hilos demonios:** Los hilos demonios son aquellos que se ejecutan en segundo plano y no impiden que el programa termine.
   Su uso es común en tareas de mantenimiento, como la recolección de basura. Estos hilos terminan automáticamente cuando no hay más hilos de usuario activos.

7. Sincronización

Los hilos pueden compartir recursos, lo que puede causar **interferencia entre hilos** y **errores de consistencia de memoria**.
La **sincronización** se utiliza para prevenir estos problemas, pero puede introducir **contención**, lo que puede ralentizar la ejecución de los hilos o incluso suspenderla.

2. Problemas Comunes de Sincronización:

Interferencia entre hilos:

Este problema se presenta cuando varios hilos acceden y modifican una variable compartida sin control, lo que puede resultar en valores inconsistentes.

Errores de consistencia de memoria:

Cuando diferentes hilos leen y escriben en el mismo dato sin una sincronización adecuada, pueden surgir errores de consistencia.
Para evitarlo, se utiliza la relación de **happens-before**, que asegura que los efectos de una operación realizada por un hilo son visibles para otros hilos en el orden correcto.

3. Solución a los Problemas de Sincronización:

Sección Crítica:

Una **sección crítica** es un bloque de código que solo puede ser ejecutado por un hilo a la vez. Esto se logra usando el **algoritmo de exclusión mutua** (mutex).

En Java, se puede marcar métodos con la palabra clave **`synchronized`** para asegurarse de que no haya acceso simultáneo a recursos compartidos.

Esto asegura que los datos sean consistentes y evita la interferencia entre hilos.

Sincronización mediante el modificador `synchronized`:

Al declarar métodos sincronizados (`synchronized`), se asegura que solo un hilo pueda ejecutar ese método a la vez, bloqueando a los demás hilos que intenten acceder al mismo recurso.

La sincronización también establece relaciones **happens-before**, asegurando la visibilidad de los cambios en el objeto.

Bloqueos Intrínsecos y Sincronización:

Cada objeto en Java tiene un **bloqueo intrínseco** asociado, que se obtiene al invocar un método sincronizado.

Esto asegura que ningún otro hilo pueda acceder al mismo objeto mientras un hilo esté ejecutando código sincronizado.

Resumen si un hilo entra a un método syncroniced de un objeto y otro hilo quiere entrar a otro método del mismo objeto no podrá acceder y quedara en espera.

Los métodos **estáticos** tienen su propio bloqueo, asociado a la clase y no a las instancias.

4. Sincronización Reentrante:

Java permite la **sincronización reentrante**, lo que significa que un hilo puede adquirir un bloqueo que ya posee.

Esto es útil cuando el código sincronizado invoca de manera indirecta otros métodos sincronizados sin que el hilo se bloquee a sí mismo.

Evita un bloqueo entre métodos syncroniced

5. Acceso Atómico:

En programación, una **acción atómica** es aquella que se ejecuta completamente sin interrupciones.
En Java, ciertas operaciones en variables de tipo primitivo (excepto `long` y `double`) son atómicas. Usar variables **volátiles** también ayuda a garantizar la visibilidad de los cambios entre hilos, ya que la escritura en una variable volátil establece una relación **happens-before**.

6. Deadlocks (Interbloqueos):

El **deadlock** ocurre cuando dos o más hilos están esperando indefinidamente unos a otros para liberar recursos.
Un ejemplo clásico es el problema entre dos amigos (Pedro y Pablo) que se bloquean mutuamente al esperar que el otro complete una acción. Este tipo de situaciones pueden resolverse **ordenando** los objetos para que siempre se adquieran los bloqueos en un orden consistente, evitando así el ciclo de espera circular que causa el deadlock.

6.2 Inanición (Starvation)

La **inanición** ocurre cuando un hilo no puede obtener acceso regular a los recursos necesarios para avanzar en su ejecución. Esto puede suceder si otros hilos, especialmente aquellos con un mayor tiempo de ejecución o que acceden de forma sincronizada a los mismos recursos, bloquean el acceso a los recursos compartidos. Los hilos que están "bloqueados" pueden no obtener nunca los recursos que necesitan y, por lo tanto, no avanzan en su ejecución.

Ejemplo: Si un hilo tiene un acceso prolongado a un objeto o recurso (por ejemplo, a través de un método sincronizado) y otros hilos también requieren acceso a ese mismo recurso, el hilo que ya está accediendo al recurso puede seguir bloqueándolo durante mucho tiempo, lo que lleva a la inanición de los otros hilos.

6.3 Livelock

Un **livelock** es una situación similar al **deadlock** (bloqueo mutuo), pero con una diferencia crucial: los hilos no están bloqueados de manera permanente, sino que están ocupados respondiendo continuamente a las acciones de otros hilos sin poder avanzar. Aunque no están "bloqueados", los hilos siguen cambiando su estado de manera constante, lo que impide que alguno de ellos termine la tarea.

Ejemplo de livelock: Imagina que dos personas, Pedro y Pablo, intentan pasar por un pasillo estrecho. Cada vez que uno se mueve para dar paso al otro, el otro hace lo mismo, pero nunca consiguen pasar, ya que se siguen moviendo en direcciones opuestas. Esto es un livelock, ya que ambos están ocupados en una "danza" de movimientos, pero no logran avanzar.

7 Guarded Blocks en Java

Un **guarded block** es un patrón de diseño en programación concurrente en el que un hilo ejecuta un bloque de código solo cuando se cumple una condición.
Este tipo de bloque se utiliza cuando un hilo espera un evento o condición específica para continuar su ejecución. Sin embargo, si el hilo no puede continuar inmediatamente, se debe suspender para evitar el desperdicio de recursos mientras espera.

Uso de `wait` y `notify`:

- **`wait()`**: Un hilo que no puede continuar su ejecución debido a que la condición no se cumple debe invocar `wait()` para suspender su ejecución. El hilo libera el bloqueo y entra en estado de espera hasta que otro hilo lo notifique.
- **`notify()`**: Un hilo que cambia el estado de la condición invoca `notify()` (o `notifyAll()`) para despertar a uno o más hilos que están esperando.

Ejemplo de un guarded block:

```
public synchronized void guardedJoy() {
    while (!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy has been achieved!");
}
```

En este ejemplo, el hilo se suspende con `wait()` hasta que otro hilo invoque `notifyAll()` para indicar que la condición ha cambiado (es decir, `joy` es verdadero). El uso de `synchronized` asegura que el hilo obtiene el bloqueo necesario para interactuar de manera segura con el objeto compartido.

8 Objetos Inmutables

Un **objeto inmutable** es un objeto cuyo estado no puede cambiar después de haber sido creado. Los objetos inmutables son muy útiles en ambientes concurrentes

porque, al no poder cambiar su estado, no existe el riesgo de que diferentes hilos interfieran con su información. Esto elimina la necesidad de sincronización para garantizar que el objeto no se corrompa.

Ventajas de los objetos inmutables:

1. **Seguridad en entornos concurrentes**: Como no pueden cambiar su estado, no se producen condiciones de carrera.
2. **Simplicidad**: Son más fáciles de entender y usar porque no cambian después de su creación.
3. **Menor sobrecarga en la recolección de basura (Garbage Collection)**: No es necesario gestionar el estado mutable y los bloqueos relacionados, lo que puede reducir la carga en la memoria.

Características clave de la clase `ImmutableRGB`:

1. **Atributos finales**: Los atributos son `final` para evitar modificaciones después de la creación del objeto.
2. **No tiene setters**: No hay métodos que permitan modificar el estado de los objetos una vez que se han creado.
3. **Métodos que retornan nuevos objetos**: El método `invert()` crea un nuevo objeto `ImmutableRGB` en lugar de modificar el objeto existente.

Reglas para crear objetos inmutables:

1. No se deben implementar métodos "setter".
2. Los atributos deben ser `final` y `private`.
3. La clase debe ser `final` para evitar que se pueda extender.
4. Si los atributos contienen referencias a objetos mutables, se deben tomar precauciones para evitar que esos objetos cambien. Esto se logra copiando las referencias y almacenando las copias en lugar de las referencias originales.

9. Patrones de Sincronización

A lo largo de la historia, se han propuesto distintas soluciones para resolver los problemas de sincronización entre procesos, adaptándose a la complejidad de cada caso, el número de secciones críticas, la cantidad de procesos y la interdependencia entre ellos. Los mecanismos de sincronización de secciones críticas deben cumplir con estos criterios básicos:

1. **Exclusión mutua**: Garantizar que solo un proceso esté en la sección crítica a la vez.

2. **No inanición (starvation)**: Asegurar que todos los procesos puedan acceder eventualmente, sin esperas indefinidas.

3. **No interbloqueo (deadlock)**: Evitar que procesos fuera de la sección crítica bloqueen el acceso de otros a la misma.

4. **Independencia del hardware**: El mecanismo debe funcionar independientemente del número de procesadores y la velocidad de los procesos.

Los patrones más comunes para resolver problemas de sincronización son los **semáforos** y los **monitores**.

9.1 Semáforos

Un **semáforo** es una técnica de sincronización de memoria compartida que regula el acceso de los procesos a la sección crítica, asegurando la exclusión mutua y evitando conflictos en sistemas concurrentes.

Un semáforo tiene tres operaciones principales:

1. **Inicializar**: Establece el estado del semáforo como liberado (verde) o bloqueado (rojo).
2. **Liberar**: Cambia el semáforo a verde, permitiendo que los procesos en espera accedan.
3. **Bloquear**: Coloca el semáforo en rojo; si ya está bloqueado, el proceso debe esperar en una cola.

Los semáforos pueden configurarse para exclusión mutua o para establecer un sistema de turnos en el acceso a recursos. En Java, el paquete `java.util.concurrent` proporciona la clase `Semaphore`, que facilita la implementación de semáforos. A continuación, se presenta un ejemplo de cómo controlar la ejecución de hilos con un límite de cuatro simultáneos usando `Semaphore`:

```
import java.util.concurrent.Semaphore;

public class SemaphoreUsage implements Runnable {
    private static final int AVAILABLE_THREADS = 4;
    private static final Semaphore semaphore = new
Semaphore(AVAILABLE_THREADS);
    private final String name;

    public SemaphoreUsage(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        try {
            semaphore.acquire();  // Solicita permiso al semáforo
            System.out.println("Executing process: " + name);
            Thread.sleep(1000);
            System.out.println("End: " + name);
            semaphore.release();  // Libera el semáforo
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

```
    public static void main(String[] args) {
        for (int i = 0; i < 20; i++) {
            new Thread(new SemaphoreUsage(Integer.toString(i +
1))).start();
        }
    }
}
```

Además de los semáforos, Java ofrece herramientas como `CountDownLatch`, `CyclicBarrier` y `Phaser` para gestionar sincronización más compleja. A continuación se muestra un ejemplo usando `CountDownLatch` para coordinar que los hilos se ejecuten en un orden específico (ej., escribir "Hello world"):

```
import java.util.concurrent.CountDownLatch;

public class CountDownUsage implements Runnable {
    private String msg;
    private CountDownLatch countDown;

    public CountDownUsage(String msg, CountDownLatch countDown) {
        this.msg = msg;
        this.countDown = countDown;
    }

    public static void main(String[] args) {
        final CountDownLatch countDownPhase1 = new
CountDownLatch(1);
        final CountDownLatch countDownPhase2 = new
CountDownLatch(1);

        new Thread(new CountDownUsage("Hello ",
countDownPhase1)).start();
        Thread t2 = new Thread(new CountDownUsage("world!",
countDownPhase2));

        try {
            countDownPhase1.await();
            t2.start();
            countDownPhase2.await();
            System.out.println("\n*** the end ***");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void run() {
        try {
            Thread.sleep(1000);
            System.out.print(msg);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        countDown.countDown();
    }
}
```

CountDownLatch vs CyclicBarrier vs Phaser

Característica	*CountDownLatch*	*CyclicBarrier*	*Phaser*
Propósito	*Esperar a que se completen ciertas tareas iniciales antes de que otros hilos puedan continuar.*	*Sincronizar un conjunto de hilos en un punto de barrera, para que todos esperen a que lleguen los demás antes de continuar.*	*Sincronizar hilos en múltiples fases, permitiendo tareas en distintas etapas o fases de ejecución.*
Reutilización	*Único uso, no se puede reutilizar después de que el contador llega a cero.*	*Reutilizable automáticamente una vez que todos los hilos han alcanzado la barrera y han sido liberados.*	*Reutilizable, permite que los hilos pasen por varias fases o ciclos.*
Contador de espera	*Valor fijo establecido en la creación (ejemplo: new CountDownLatch(3)).*	*Número fijo de hilos debe alcanzar la barrera (ejemplo: new CyclicBarrier(3)).*	*No tiene un límite fijo, permite que los hilos se registren y se sincronicen dinámicamente en cada fase.*
Funcionalidad principal	*Disminuye el contador con countDown() cada vez que un hilo completa su tarea. Cuando llega a cero, todos los hilos en espera son liberados con await().*	*Todos los hilos llaman a await() y esperan hasta que el número requerido de hilos llegue a la barrera; luego todos continúan.*	*Los hilos pueden registrarse y pasar por varias fases de sincronización, permitiendo un control flexible en cada fase.*
Acción en cada ciclo o fase	*No aplica.*	*Opcional: se puede especificar una acción (Runnable) que se ejecutará cuando todos los hilos alcancen la barrera en cada ciclo.*	*Opcional: permite ejecutar acciones en cada fase, similar a CyclicBarrier, pero con mayor flexibilidad.*
Ideal para	*Situaciones donde un conjunto de tareas iniciales deben completarse antes de que continúe otro conjunto de tareas.*	*Situaciones donde un grupo fijo de hilos debe trabajar en fases cíclicas, sincronizándose en una barrera común antes de proceder a la siguiente etapa.*	*Situaciones con tareas que tienen fases múltiples o cíclicas, con un número dinámico de hilos en cada fase.*
Ejemplo de uso	*Iniciar una aplicación solo después de que se completen varias tareas de inicialización.*	*Coordinación de hilos en algoritmos paralelos o simulaciones donde los hilos necesitan sincronizarse en puntos comunes.*	*Procesos que necesitan pasar por múltiples fases, como simulaciones complejas con etapas de preparación y ejecución.*
Ejemplo de código	*CountDownLatch latch = new CountDownLatch(3); latch.countDown(); latch.await();*	*CyclicBarrier barrier = new CyclicBarrier(3); barrier.await();*	*Phaser phaser = new Phaser(); phaser.register(); phaser.arriveAndAwaitAdvance();*

9.2 Monitores

Otra técnica para la sincronización de procesos es el uso de **monitores**. Un monitor es una abstracción que gestiona el acceso de los procesos a un recurso compartido en exclusión mutua, encapsulando un conjunto de métodos y datos. Los procesos solo pueden interactuar con el recurso compartido mediante los métodos del monitor, garantizando la exclusión mutua sin que el programador gestione los bloqueos directamente.

Un monitor se puede visualizar como una habitación con puerta cerrada, donde solo un proceso a la vez puede ingresar para utilizar el recurso compartido. Los procesos que desean acceder deben "llamar" al monitor, el cual gestiona el acceso de forma segura y ordenada.

Los monitores utilizan **variables de condición** para gestionar la sincronización:

- **sendWait**: Un proceso que espera a que una condición se cumpla abandona temporalmente el monitor y espera en una cola asociada a su variable de condición.
- **sendSignal**: Libera un proceso de la cola de espera y lo prepara para ingresar al monitor. Si no hay procesos esperando, no tiene efecto.

Un monitor consta de los siguientes elementos:

1. **Variables y métodos internos**: Solo accesibles dentro del monitor y no modificables externamente.
2. **Código de inicialización**: Establece los valores iniciales del monitor al ser creado.
3. **Métodos externos**: Accesibles para procesos que necesitan acceder al recurso.
4. **Cola de procesos**: Procesos en espera que serán liberados al cumplirse las condiciones definidas.

En Java, podemos implementar monitores mediante métodos `synchronized`, que garantizan la exclusión mutua, y los métodos `wait` y `notifyAll`, que permiten la sincronización entre hilos.

8. El Modelo Productor-Consumidor

El **problema productor-consumidor** es un ejemplo clásico de sincronización en entornos de multiprocesos. Involucra dos procesos que comparten un **buffer de tamaño limitado**:

- El **productor** genera productos y los almacena en el buffer.
- El **consumidor** toma productos del buffer.

Problema:

- El productor no debe intentar añadir más productos si el buffer está lleno.
- El consumidor no debe intentar tomar productos si el buffer está vacío.

Solución:

Ambos procesos deben sincronizarse para operar en el buffer sin interferencias. Esto se logra haciendo que el productor y el consumidor:

- Se “despierten” o “duerman” según el estado del buffer.

En Java, se usa `synchronized`, `wait()` y `notifyAll()` para implementar esta sincronización. Si el buffer está lleno, el productor se "duerme" hasta que el consumidor tome un producto y lo notifique. Si el buffer está vacío, el consumidor se duerme hasta que el productor le notifique que un producto ha sido agregado.

Ejemplo en Código:

- La clase `Drop` simula el buffer, permitiendo almacenar solo un mensaje a la vez.
- El productor (clase `Producer`) envía mensajes al consumidor y se detiene por intervalos aleatorios para simular condiciones reales.
- El consumidor (clase `Consumer`) toma mensajes y los imprime hasta que recibe la señal de finalización (`"."`).

Este mecanismo asegura que el productor y el consumidor trabajen en armonía sin interferir entre sí, evitando un acceso simultáneo que pueda llevar a errores.

---

9. Concurrencia de Alto Nivel en Java

Java proporciona una serie de clases en el paquete `java.util.concurrent.locks` que permiten una sincronización más avanzada y flexible que los bloques `synchronized`. Estas clases permiten a los hilos comunicarse de forma más eficiente y evitan problemas de bloqueo mutuo o **deadlock**.

9.1 Locks

La interfaz **Lock** en Java funciona de manera similar a los bloqueos implícitos de `synchronized`, pero ofrece más flexibilidad:

- **tryLock()**: Permite al hilo intentar obtener un bloqueo sin quedar esperando indefinidamente.
- **lockInterruptibly()**: Permite al hilo abandonar el bloqueo si se recibe una interrupción antes de adquirirlo.

Ejemplo de Uso con `ReentrantLock` para Evitar Deadlocks

La clase `ReentrantLock` implementa la interfaz `Lock`, y permite gestionar el bloqueo de forma más controlada:

- La clase `Friend` contiene un `Lock` individual para cada amigo.
- Cuando un amigo intenta saludar a otro (método `tryBow()`), intenta adquirir ambos `Locks` (el suyo y el del otro).
    - Si logra ambos, procede con el saludo y luego libera los `Locks`.
    - Si solo logra uno, lo libera inmediatamente para evitar un **deadlock**.

Código Ejemplo: Deadlock Seguro

En la clase `Bow`, cada saludo se ejecuta en un hilo distinto. El código intenta obtener ambos bloqueos de manera que si uno no está disponible, se libera el que sí fue bloqueado, evitando así situaciones de bloqueo mutuo entre hilos que intentan acceder a los mismos recursos.

9.2 Executors

En el ámbito de la programación concurrente en Java, el uso de **executors** facilita la gestión de hilos al separar la lógica de creación y control de los hilos de la lógica de la aplicación. La idea es que los ejecutores administren hilos para procesar tareas concurrentemente y de forma optimizada. Los hilos se crean de forma eficiente y los executors permiten al programador enfocarse en dividir el trabajo sin preocuparse por la administración directa de los hilos.

9.2.1 Interfaces Executor

Java proporciona tres interfaces clave para gestionar tareas con executors en el paquete `java.util.concurrent`:

- **Executor**: Es la interfaz básica y permite lanzar tareas de forma general con el método `execute`.
- **ExecutorService**: Una subinterfaz de `Executor`, que añade métodos para gestionar el ciclo de vida de las tareas y controlar su finalización.
- **ScheduledExecutorService**: Una subinterfaz de `ExecutorService`, que permite programar la ejecución de tareas en intervalos específicos o tras un retraso inicial.

Interfaz Executor

El método principal de la interfaz `Executor` es `execute()`, que se usa para enviar tareas a un hilo. Por ejemplo:

```
java
Copiar código
(new Thread(r)).start();
```

se reemplaza por

```
java
Copiar código
e.execute(r);
```

Este cambio simplifica el envío de tareas a hilos sin la necesidad de crearlos manualmente. Un executor puede hacer que una tarea se ejecute inmediatamente o esperar hasta que un hilo trabajador esté disponible.

Interfaz ExecutorService

Además del método `execute`, `ExecutorService` incluye `submit`, que permite que las tareas devuelvan un valor a través de objetos de tipo `Callable`. Esto devuelve un objeto `Future`, que se usa para obtener el valor calculado cuando la tarea finaliza. También permite gestionar grandes conjuntos de tareas y realizar un cierre controlado del executor.

Interfaz ScheduledExecutorService

Esta interfaz extiende `ExecutorService` con el método `schedule`, que permite programar una tarea para que se ejecute después de un cierto tiempo, o periódicamente con los métodos `scheduleAtFixedRate` y `scheduleWithFixedDelay`.

9.2.2 Thread Pools

Los **pools de hilos** (thread pools) permiten reutilizar hilos ya existentes, evitando el coste de creación y destrucción de hilos repetitivamente. Al asignar tareas en paralelo, el pool reduce la carga de trabajo en el sistema.

Existen tres clases principales para gestionar pools:

- **ThreadPoolExecutor**: Es el gestor genérico que maneja tareas independientes. Utiliza una cola de tareas para organizar el trabajo.
- **ScheduledThreadPoolExecutor**: Permite programar tareas periódicas o con retraso.
- **ForkJoinPool**: Adecuado para tareas paralelas que pueden descomponerse en subproblemas más pequeños, utilizando el modelo de divide y vencerás.

Configuración del ThreadPoolExecutor

La clase `Executors` proporciona métodos estáticos para crear instancias de `ThreadPoolExecutor`:

- **newCachedThreadPool()**: Crea un pool que crea hilos según necesidad y reutiliza los hilos inactivos.
- **newFixedThreadPool(int numThreads)**: Crea un pool con un número fijo de hilos, donde las tareas se agregan a una cola y se ejecutan cuando un hilo queda libre.
- **newSingleThreadExecutor()**: Crea un pool con un solo hilo, para procesar todas las tareas en secuencia.

Cada configuración se adapta a diferentes necesidades de concurrencia según la carga y el sistema donde se ejecuta.

Métodos Importantes de ThreadPoolExecutor

- **execute**: Ejecuta una tarea en algún momento futuro.
- **invokeAll**: Ejecuta una colección de tareas y devuelve una lista de objetos `Future` que representan los resultados.
- **getPoolSize**: Devuelve el número actual de hilos en el pool.
- **getCompletedTaskCount**: Retorna el número de tareas que han completado su ejecución.

Ejemplo de Uso: ThreadPoolExecutor

El siguiente ejemplo crea un `ThreadPoolExecutor` con tres hilos para ejecutar diez multiplicaciones aleatorias:

```
java
Copiar código
ThreadPoolExecutor executor = (ThreadPoolExecutor)
Executors.newFixedThreadPool(3);
List<Multiplication> taskList = new ArrayList<>();
for (int i = 0; i < 10; i++) {
    int op1 = (int) (Math.random() * 10);
    int op2 = (int) (Math.random() * 10);
    taskList.add(new Multiplication(op1, op2));
}
List<Future<Integer>> resultList = executor.invokeAll(taskList);
executor.shutdown();
for (int i = 0; i < resultList.size(); i++) {
    System.out.println("Result of task " + i + " is:" +
resultList.get(i).get());
}
```

Ejemplo de Uso: ScheduledThreadPoolExecutor

A continuación, un ejemplo que usa `ScheduledThreadPoolExecutor` para verificar una carpeta periódicamente:

```
java
Copiar código
ScheduledExecutorService executor =
Executors.newScheduledThreadPool(1);
WatcherTask task = new WatcherTask("/ruta/a/monitorear");
```

```
ScheduledFuture<?> result = executor.scheduleAtFixedRate(task, 1, 5,
TimeUnit.SECONDS);
try {
    Thread.sleep(60000); // Espera un minuto
} catch (InterruptedException e) {
    e.printStackTrace();
}
executor.shutdown();
```

Aquí, `scheduleAtFixedRate` inicia la tarea de monitoreo de archivos después de un segundo y luego la ejecuta cada cinco segundos, sin interrupciones entre ejecuciones.

El uso de **executors** y **thread pools** permite crear aplicaciones concurrentes eficientes, escalables y fáciles de mantener, y proporciona un control avanzado sobre la ejecución de tareas y la utilización de recursos del sistema.

9.2.3 Fork-Join

El framework Fork-Join es una extensión de los patrones de concurrencia de Java basados en el uso de `Executor`, diseñado específicamente para dividir y procesar tareas de manera recursiva y paralela. Esto permite mejorar la eficiencia del procesamiento cuando se tienen tareas que pueden dividirse en sub-tareas independientes, ideales para computación intensiva y algoritmos que pueden resolverse de forma recursiva.

1. **ForkJoinPool**: es el gestor de hilos central en este framework, que ejecuta tareas utilizando el algoritmo de robo de trabajo (*work-stealing*). Este algoritmo permite que los hilos activos roben tareas en espera de otros hilos inactivos o con menos carga, maximizando la utilización de los núcleos del procesador.
2. **ForkJoinTask**: es una clase abstracta que representa las tareas que se ejecutan en el `ForkJoinPool`, proporcionando métodos como `fork()` y `join()`. Estos métodos ayudan a dividir las tareas (`fork`) y a unirlas nuevamente (`join`) una vez completadas. El método `fork()` envía la tarea a la cola de ejecución, y `join()` bloquea al hilo hasta que la tarea se complete, permitiendo que el `ForkJoinPool` aproveche el bloqueo para reasignar tareas.
3. **Clases RecursiveTask y RecursiveAction**:
    - `RecursiveTask`: Se usa para tareas que devuelven un valor, y es útil en problemas como el cálculo de un valor máximo en un arreglo.
    - `RecursiveAction`: Se utiliza cuando la tarea no necesita devolver un valor. Incluye el método `invokeAll`, que ayuda a ejecutar múltiples sub-tareas de manera paralela sin necesidad de combinar resultados.

Ejemplo: Búsqueda del Máximo en un Arreglo de Enteros

En el siguiente ejemplo se utiliza `RecursiveTask` para encontrar el número máximo en un arreglo de enteros mediante una combinación de búsqueda iterativa y recursiva. La tarea se divide en sub-tareas hasta que la porción del arreglo sea lo suficientemente pequeña como para ser procesada iterativamente.

```
java
Copiar código
import java.util.Random;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class MaxNumberTask extends RecursiveTask<Integer> {
    // Tamaño total del array y umbral para usar búsqueda iterativa
    final static int TOTAL = 100_000_000;
    final static int THRESHOLD = 10_000_000;

    int[] numbers;
    int first;
    int length;

    // Constructor para inicializar la tarea
    public MaxNumberTask(int[] numbers, int first, int length) {
        this.numbers = numbers;
        this.first = first;
        this.length = length;
    }

    // Método para obtener el máximo iterativamente
    private int getMaxIterative() {
        int max = Integer.MIN_VALUE;
        int high = first + length;
        for (int i = first; i < high; i++) {
            if (numbers[i] > max) {
                max = numbers[i];
            }
        }
        return max;
    }

    // Método para obtener el máximo de forma recursiva
    private int getMaxRecursive() {
        int mid = length / 2;
        MaxNumberTask task1 = new MaxNumberTask(numbers, first, mid);
        task1.fork();
        MaxNumberTask task2 = new MaxNumberTask(numbers, first + mid,
length - mid);
        task2.fork();
        return Math.max(task1.join(), task2.join());
    }

    // Método `compute` para ejecutar la tarea
    @Override
    protected Integer compute() {
        if (length > THRESHOLD) {
            return getMaxRecursive();
        } else {
            return getMaxIterative();
        }
    }

    // Generar arreglo aleatorio de enteros
    static int[] generateArray(int size) {
        int[] numbers = new int[size];
        Random r = new Random(System.currentTimeMillis());
        for (int i = 0; i < numbers.length; i++) {
            numbers[i] = r.nextInt();
```

```
        }
        return numbers;
    }

    public static void main(String[] args) {
        int[] numbers = generateArray(TOTAL);

        // Búsqueda secuencial iterativa
        MaxNumberTask task = new MaxNumberTask(numbers, 0, TOTAL);
        long start = System.currentTimeMillis();
        int max = task.getMaxIterative();
        long end = System.currentTimeMillis();
        System.out.println("Max number (" + max + ") iterative found
in " + (end - start) + " ms");

        // Búsqueda recursiva en paralelo con ForkJoin
        ForkJoinPool pool = new ForkJoinPool();
        start = System.currentTimeMillis();
        Integer res = pool.invoke(task);
        end = System.currentTimeMillis();
        System.out.println("Max number (" + res + ") recursive
parallel found in " + (end - start) + " ms");
    }
}
```

Explicación del Código:

1. **generateArray**: Método que crea un arreglo de números aleatorios para la prueba.
2. **getMaxIterative**: Método que obtiene el valor máximo de manera iterativa dentro de una sección del arreglo.
3. **getMaxRecursive**: Método que divide el arreglo en dos mitades y usa `fork()` para iniciar cada mitad como una sub-tarea, utilizando `join()` para combinar los resultados.
4. **compute**: Método de `RecursiveTask` que determina si el arreglo es suficientemente pequeño como para calcular el máximo de manera iterativa o si debe seguir dividiéndose.
5. **main**: Crea una instancia del arreglo y ejecuta tanto la búsqueda iterativa como la recursiva usando `ForkJoinPool`.

Colecciones Concurrentes y Variables Atómicas

Además de `ForkJoin`, Java proporciona colecciones concurrentes en el paquete `java.util.concurrent`, como `BlockingQueue` (FIFO), `ConcurrentMap` y `ConcurrentNavigableMap`, diseñadas para evitar problemas de consistencia de memoria en entornos multi-hilo.

Variables Atómicas

La clase `AtomicInteger` permite manejar enteros con operaciones atómicas como `incrementAndGet` sin necesidad de sincronización. Esto permite evitar bloqueos en métodos críticos al incrementar, decrementar o leer valores de forma segura en aplicaciones concurrentes.