UT3: Programación Multihilo en Java

Un recorrido por los conceptos y aplicaciones de la programación concurrente

Presentador: [Tu Nombre]

Fecha: 30 de octubre de 2025

¿Qué es un hilo?

Definición

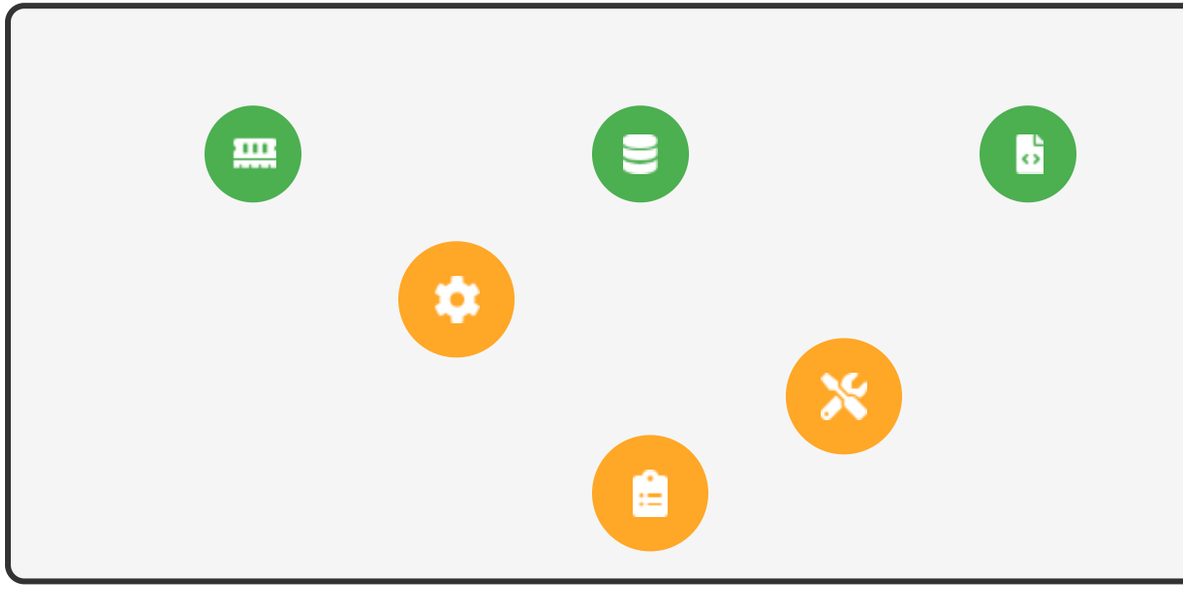
Un hilo (thread) es una secuencia de ejecución independiente dentro de un proceso.

Permite que una aplicación realice múltiples tareas de forma simultánea o aparentemente simultánea.

Características

- Un proceso puede tener múltiples hilos de ejecución
- Todos los hilos dentro de un proceso comparten el mismo espacio de memoria y recursos
- Cada hilo tiene su propio estado, registro de programación y pila

Analogía: Fábrica con Trabajadores



En la analogía de la fábrica:

- La **fábrica** representa el **proceso**
- Los **trabajadores** representan los **hilos**
- Los **recursos** de la fábrica (memoria, máquinas) son compartidos por todos los trabajadores

Hilos vs. Procesos

Hilos

Componentes de un proceso que comparten su espacio de memoria y recursos.

Procesos

Entidades independientes con su propio espacio de memoria aislado.

Característica	Hilos	Procesos
Independencia	Dependientes del proceso padre	Independientes
Espacio de Memoria	Comparten el mismo espacio de memoria (código, datos, heap)	Propio espacio de memoria aislado
Comunicación	Rápida y directa (acceso a memoria compartida)	Compleja y costosa (requiere mecanismos de IPC)
Coste de Creación	Menor coste (más ligeros)	Mayor coste (más pesados)
Cambio de Contexto	Más rápido	Más lento

Ventajas del Multihilo

Mejora de Responsividad

Permite que la interfaz de usuario permanezca interactiva mientras se ejecutan tareas largas en segundo plano.

Procesadores Multinúcleo

Facilita la ejecución de subtareas independientes en diferentes núcleos de CPU, optimizando el uso del hardware.

Eficiencia en E/S

Otros hilos pueden continuar ejecutándose mientras uno espera la finalización de una operación de entrada/salida.

Menor Coste

Los hilos son más ligeros que los procesos, lo que reduce la sobrecarga del sistema y mejora el rendimiento.

Comunicación Eficiente

La memoria compartida permite una comunicación más rápida y directa entre las diferentes partes de la aplicación.

Rendimiento Mejorado

La programación multihilo mejora el rendimiento general de las aplicaciones al permitir una mejor utilización de los recursos del sistema.

Creación de Hilos en Java

Java ofrece dos principales abstracciones para trabajar con concurrencia:

Extendiendo Thread

```java
public class MiHiloExtendido extends Thread {
    @Override
    public void run() {
        System.out.println("Hola desde un hilo extendido");
    }

    public static void main(String[] args) {
        MiHiloExtendido hilo = new MiHiloExtendido();
        hilo.start(); // Inicia la ejecución del hilo
    }
}
```

- ✅ Se crea una nueva clase que extiende Thread
- ✅ Se sobrescribe el método run() con el código a ejecutar
- ❌ Limita la herencia de la clase del hilo a otra clase

Implementando Runnable

```java
public class MiTareaRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Hola desde un hilo implementando Runna
    }

    public static void main(String[] args) {
        MiTareaRunnable tarea = new MiTareaRunnable();
        Thread hilo = new Thread(tarea);
        hilo.start(); // Inicia la ejecución del hilo
    }
}
```

- ✅ Se crea una clase que implementa Runnable
- ✅ Se pasa la instancia de Runnable al constructor de Thread
- ✅ Método preferido porque promueve la composición sobre la herencia

💡 La implementación de Runnable es el método preferido en Java para crear hilos

La Clase Thread

¿Qué es Thread?

La clase **java.lang.Thread** es la representación de un hilo de ejecución en Java.

Permite crear, controlar y gestionar el estado de los hilos.

Extender Thread

- Se crea una nueva clase que extiende **Thread**
- Se sobrescribe el método **run()** con el código que ejecutará el hilo
- Se llama al método **start()** para iniciar la ejecución del hilo

Ejemplo de Código

```
public class MiHiloExtendido extends Thread {
  @Override
  public void run() {
    System.out.println("Hola desde un hilo
extendido: " +
      Thread.currentThread().getName());
  }

  public static void main(String[] args) {
    MiHiloExtendido hilo = new MiHiloExtendido();
    hilo.start(); // Inicia la ejecución del hilo
  }
}
```

Flujo de Ejecución

La Interfaz Runnable

Descripción

Implementar la interfaz **java.lang.Runnable** es el método preferido para crear hilos en Java.

La interfaz **Runnable** define un único método **run()** que contiene el código que el hilo ejecutará.

Ejemplo de Implementación

```
public class MiTareaRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Hola desde un hilo
implementando Runnable: " +
Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        MiTareaRunnable tarea = new MiTareaRunnable();
        Thread hilo = new Thread(tarea); // Se pasa la
instancia de Runnable al constructor de Thread
        hilo.start(); // Inicia la ejecución del hilo
    }
}
```

Ventajas sobre la Extensión de Thread

Promueve la Composición sobre la Herencia

Permite que la clase del hilo herede de otra clase si es necesario, superando la limitación de la herencia múltiple en Java.

Mayor Flexibilidad

Una clase puede implementar múltiples interfaces, pero solo puede extender de una clase. Runnable ofrece mayor flexibilidad.

Mejor Modelación de Objetos

A menudo es más natural que una clase "sea un hilo" (implementar Runnable) que "extender un hilo" (heredar de Thread).

Nota: La interfaz Runnable es más utilizada en aplicaciones concurrentes modernas y es el método preferido por la comunidad Java.

Métodos Esenciales de Thread

start()

Inicia la ejecución del hilo. Llama internamente al método `run()` en un hilo separado.

Cambia el estado del hilo de NEW a RUNNABLE

run()

Contiene el código que será ejecutado por el hilo. Puede ser sobrescrito por las clases que extienden Thread o implementan Runnable.

Método que contiene la lógica de ejecución del hilo

sleep(long millis)

Pausa la ejecución del hilo durante el tiempo especificado en milisegundos. El hilo permanece en estado RUNNABLE pero inactivo.

Cambia el estado a TIMED_WAITING

join()

Permite que el hilo actual espere a que el hilo llamado termine su ejecución. Puede bloquearse hasta que el hilo termine o transcurra un tiempo límite.

Cambia el estado a WAITING o TIMED_WAITING

yield()

Sugiere al planificador que el hilo actual ceda el control del procesador a otro hilo. El hilo vuelve a la cola de listos y puede ser reprogramado inmediatamente.

Vuelve a la cola de ejecución (RUNNABLE)

Otros Métodos

- `stop()`: Detiene la ejecución del hilo (obsoleto)
- `suspend()`: Suspende el hilo (obsoleto)
- `resume()`: Reanuda un hilo suspendido (obsoleto)
- `isAlive()`: Verifica si el hilo está en ejecución

Ciclo de Vida de un Hilo

¿Qué es el ciclo de vida de un hilo?

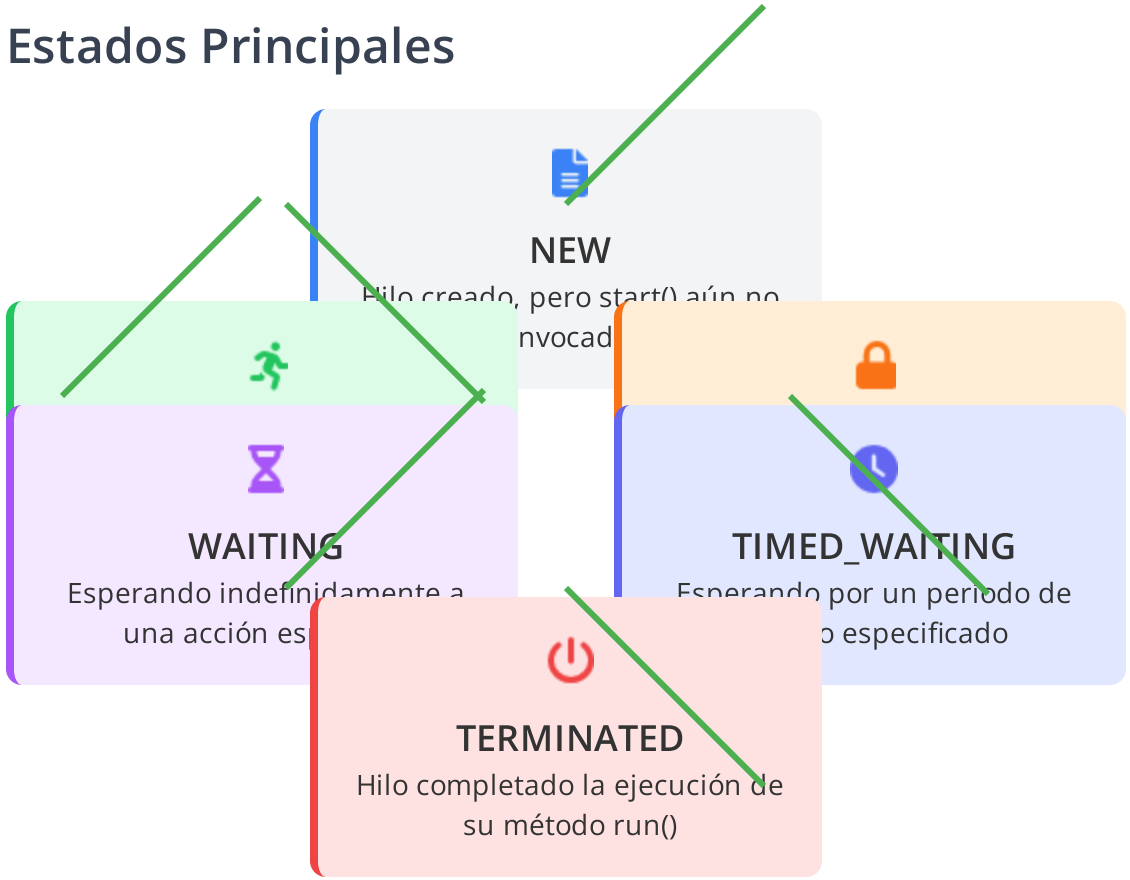
Un hilo, desde su creación hasta su finalización, atraviesa una serie de estados bien definidos.

Comprender este ciclo de vida es fundamental para diseñar, depurar y optimizar aplicaciones multihilo.

Importancia del ciclo de vida

- Permite entender cómo la JVM y el sistema operativo gestionan la ejecución concurrente
- Facilita el diagnóstico y solución de problemas comunes en programación multihilo
- Permite escribir código que controle adecuadamente el comportamiento de los hilos

Estados Principales



Estados de un Hilo

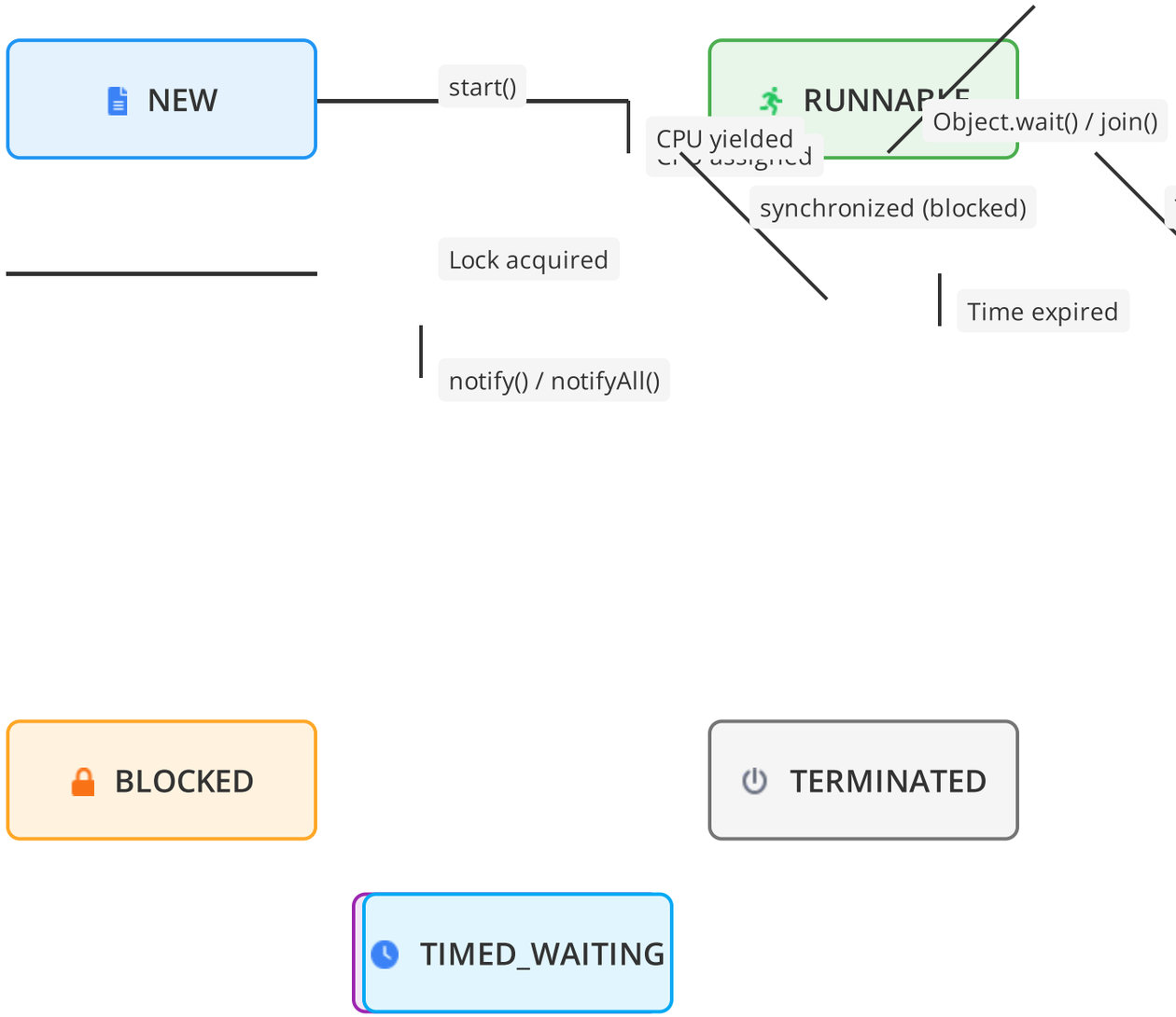
En Java, los estados de un hilo se representan mediante la enumeración **Thread.State**. Estos estados son gestionados por el planificador de la JVM/SO y son cruciales para entender el comportamiento de un hilo.

NEW

Estado: Nuevo

El hilo ha sido creado (instanciado `Thread`

Transiciones entre Estados



Métodos que Afectan las Transiciones

- **st** run() finished ejecución del hilo, pasando de NEW a RUNNABLE
- **stop()**: No se recomienda usar, puede dejar hilos en estados inconsistentes
- **synchronized**: Bloquea acceso a secciones críticas, puede causar transiciones a BLOCKED
- **wait()**: Pone el hilo en espera indefinida, transición a WAITING
- **sleep(long)**: Pone el hilo en espera con tiempo límite, transición a TIMED_WAITING

Notas Importantes

- Un hilo en TERMINATED no puede ser reiniciado
- Las transiciones entre estados son gestionadas por el planificador de la JVM/SO
- Un hilo en BLOCKED, WAITING o TIMED_WAITING no consume tiempo de CPU

Problemas de Concurrencia

Introducción

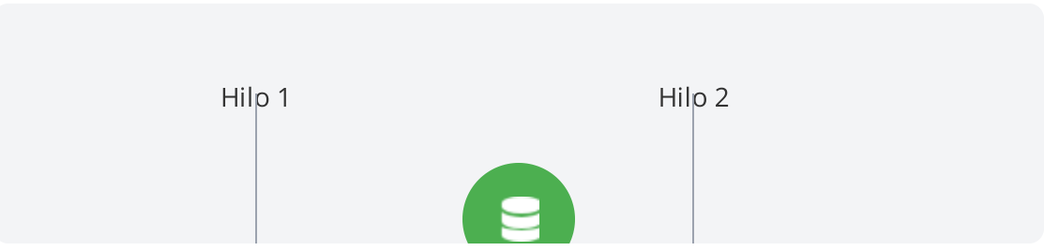
La capacidad de los hilos para compartir el mismo espacio de memoria es una de sus mayores ventajas, ya que facilita la comunicación y el acceso a datos comunes.

Sin embargo, esta característica es también la principal fuente de errores complejos y difíciles de depurar en la programación multihilo si no se gestiona con el debido cuidado.

Consecuencias

- Resultados impredecibles en la ejecución
- Corrupción de datos
- Pérdida de consistencia en la información

Categorías de Problemas



Condición de Carrera

Cuando dos o más hilos acceden o modifican un recurso compartido, y el resultado depende del orden impredecible de ejecución.

Inconsistencia de Memoria

Cuando diferentes hilos tienen "vistas" distintas del mismo dato compartido debido a cachés locales y falta de coordinación.

Interbloqueo e Inanición

Situaciones donde los hilos se bloquean mutuamente esperando recursos, o nunca obtienen acceso a los recursos necesarios.

Condición de Carrera

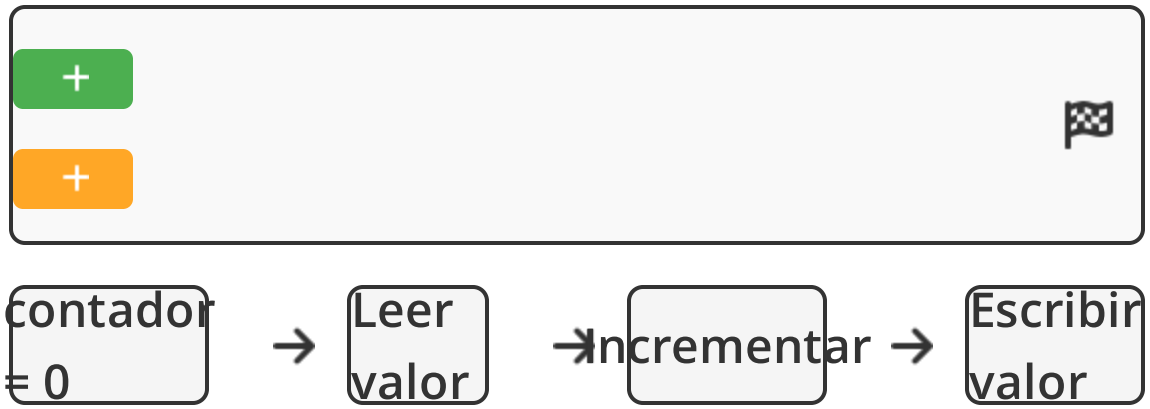
¿Qué es una condición de carrera?

Ocurre cuando dos o más hilos acceden o modifican un recurso compartido, y el resultado final depende del orden impredecible en que se ejecutan las operaciones de los hilos.

¿Por qué es problemático?

- ⚠ Resultados impredecibles y dependen del tiempo de ejecución
- ⚠ Puede llevar a datos corruptos y resultados incorrectos
- ⚠ Difícil de detectar y depurar debido a la naturaleza no determinista

Ejemplo: Incremento de un Contador




```
public class Contador {
    private int cuenta = 0;

    public void incrementar() {
        cuenta++; // Esta operación no es atómica
    }

    public int getCuenta() {
        return cuenta;
    }
}
```

La operación `cuenta++` no es atómica y se compone de tres pasos:

1. Leer el valor actual
2. Incrementarlo
3. Escribir el nuevo valor

Si dos hilos intentan incrementar el contador simultáneamente, pueden perderse incrementos.

Inconsistencia de Memoria

¿Qué es la Inconsistencia de Memoria?

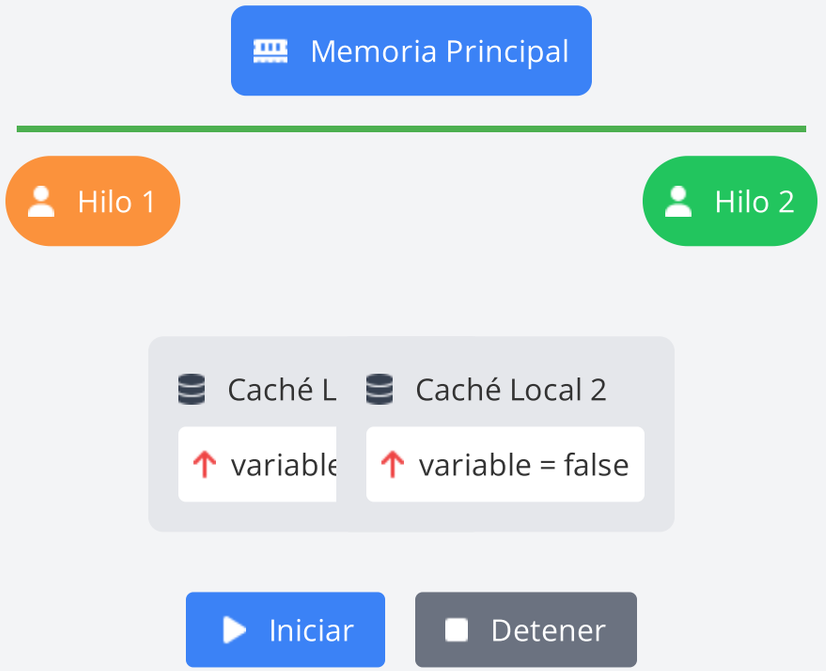
La inconsistencia de memoria surge cuando diferentes hilos tienen "vistas" distintas del mismo dato compartido.

Esto ocurre porque los procesadores modernos utilizan cachés locales para mejorar el rendimiento.

Problema de Visibilidad

- Un hilo puede modificar una variable en su caché local
- Este cambio puede no ser inmediatamente visible para otros hilos
- Los hilos pueden estar trabajando con versiones desactualizadas de los datos

Representación Visual



Ejemplo Práctico

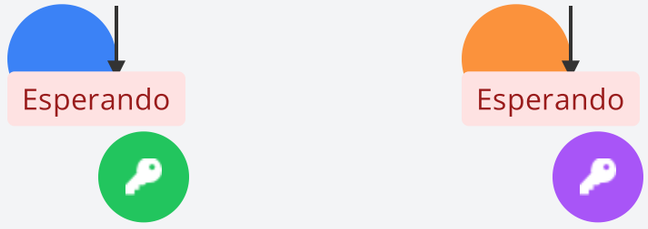
Si el Hilo A escribe `variableCompartida = true;`

El Hilo B podría seguir viendo `false` si su caché no se ha actualizado.

Interbloqueo e Inanición

Interbloqueo (Deadlock)

Situación en la que dos o más hilos se bloquean mutuamente, esperando indefinidamente un recurso que el otro hilo posee.

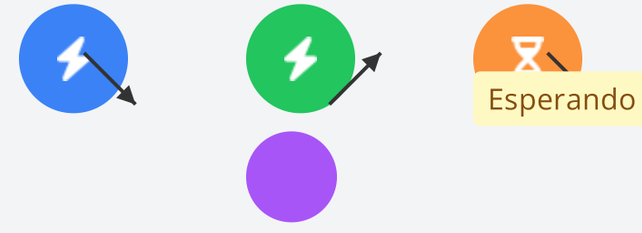


Causas y Consecuencias

- Recursos compartidos sin una estrategia de bloqueo adecuada
- Orden de adquisición de bloqueos incorrecto
- El sistema se queda sin respuesta (no hay progreso)

Inanición (Starvation)

Situación en la que un hilo nunca obtiene acceso a un recurso o a la CPU porque otros hilos, a menudo de mayor prioridad o que acaparan el recurso, lo impiden constantemente.



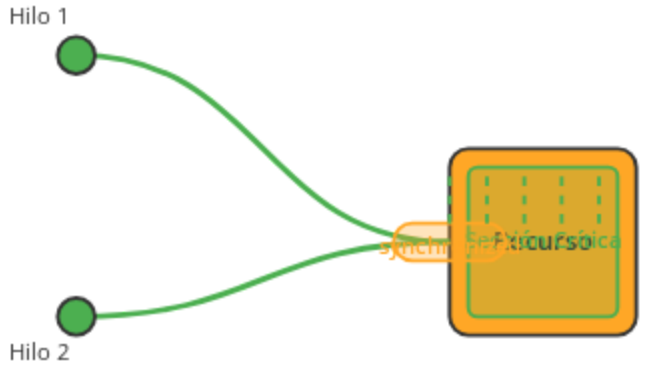
Causas y Consecuencias

- Prioridades desequilibradas entre hilos
- Acaparamiento de recursos por parte de hilos de mayor prioridad
- El hilo con menor prioridad nunca logra ejecutarse o adquirir los recursos necesarios

Sincronización de Hilos

¿Qué es la Sincronización?

La sincronización es el conjunto de mecanismos esenciales en la programación multihilo para gestionar de forma segura el acceso a los recursos compartidos.



Propósito

- Prevenir problemas de concurrencia como condiciones de carrera, inconsistencia de memoria y deadlocks
- Garantizar la integridad de los datos cuando múltiples hilos acceden a recursos compartidos
- Coordinar la ejecución de hilos para que trabajen juntos de manera eficiente y predecible

Mecanismos de Sincronización

synchronized
Bloques y métodos sincronizados

Locks Explícitos
ReentrantLock, ReadWriteLock

Operaciones Atómicas
AtomicInteger, AtomicReference

Semáforos
Control de acceso a recursos

Exclusión Mutua

¿Qué es la Exclusión Mutua?

Un principio fundamental en programación concurrente que asegura que solo un hilo puede ejecutar una **sección crítica** a la vez.

Una **sección crítica** es cualquier parte del código que accede o modifica un recurso compartido.

Importancia

- Previene condiciones de carrera
- Garantiza la integridad de los datos compartidos
- Protege los recursos críticos contra accesos concurrentes

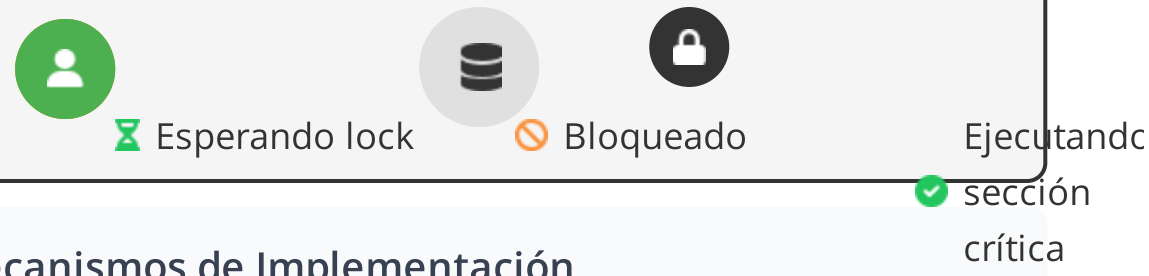
Implementación en Java

```
public class Contador {
    private int cuenta = 0;

    // Solo un hilo puede ejecutar este método a la vez
    public synchronized void incrementar() {
        cuenta++; // Sección crítica
    }

    public synchronized int getCuenta() {
        return cuenta;
    }
}
```

Visualización de Exclusión Mutua



Mecanismos de Implementación

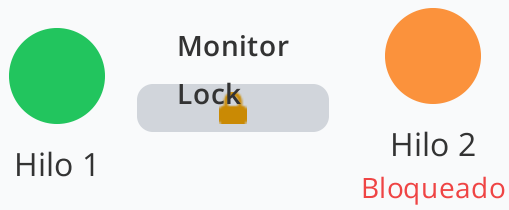
- **synchronized**: Bloqueos automáticos basados en objetos
- **ReentrantLock**: Implementación explícita con mayor control

La Palabra Clave synchronized

¿Qué es synchronized?

La palabra clave **synchronized** es el mecanismo principal para lograr la exclusión mutua en Java.

Cuando un hilo entra en un bloque o método **synchronized**, adquiere un "monitor lock" sobre un objeto. Si otro hilo intenta acceder a una sección sincronizada protegida por el mismo monitor lock, se bloqueará hasta que el primer hilo libere el lock.



Métodos Sincronizados

Cuando se aplica a un método de instancia, el lock se adquiere sobre la instancia (**this**) del objeto. Para métodos estáticos, el lock se adquiere sobre el objeto **Class** de la clase.

```
public class Contador {
    private int cuenta = 0;

    // El lock se adquiere sobre la instancia de Conta
    public synchronized void incrementar() {
        cuenta++;
    }

    public synchronized int getCuenta() {
        return cuenta;
    }
}
```

Bloques Sincronizados

Permite sincronizar solo una parte específica del código, lo que ofrece mayor granularidad y puede mejorar la concurrencia.

```
public class EscritorLog {
    private final Object lock = new Object(); // Objet

    public void escribirLog(String mensaje) {
        // Solo un hilo puede ejecutar este bloque a l
        synchronized (lock) {
            System.out.println("Hilo actual: " + Threa
            // Simular escritura en archivo
        }
    }
}
```

Importante: Todos los hilos deben sincronizar sobre el mismo objeto de bloqueo para que la exclusión mutua funcione correctamente.

Otros Mecanismos de Sincronización

Locks Explícitos

Permiten mayor control que **synchronized**:

- Intentar adquirir un lock sin bloquearse (`tryLock()`)
- Establecer tiempos de espera para adquirir un lock
- Adquirir y liberar locks en diferentes bloques de código

Operaciones Atómicas

Clases para operaciones atómicas sobre tipos de datos primitivos:

`AtomicInteger` `AtomicLong` `AtomicReference`

Más eficientes que **synchronized** para operaciones simples como incrementos o actualizaciones condicionales.

Semáforos

Controlan el número de hilos que pueden acceder a un recurso simultáneamente.

Un semáforo se inicializa con un número de "permisos"; los hilos deben adquirir un permiso antes de acceder al recurso.

Monitores

Construcciones de alto nivel que encapsulan un recurso compartido, los procedimientos para manipularlo y mecanismos de espera/notificación.

- Implementan el concepto de monitor con `synchronized` y `Object.wait/notify`
- Proporcionan exclusión mutua y mecanismos para coordinar hilos

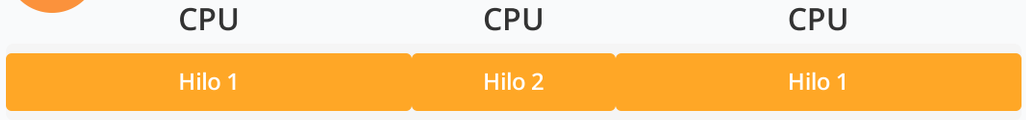
Planificación de Hilos

La gestión de cuándo y cómo los hilos obtienen acceso a la CPU es una tarea fundamental del planificador de hilos, que puede ser parte del sistema operativo (SO) o de la Máquina Virtual de Java (JVM).

Multihilo Apropiativo

- El SO o la JVM tiene el control total sobre la ejecución de los hilos
- Puede suspender un hilo en cualquier momento (expropiación de CPU)
- Un hilo se ejecuta por un "quantum" de tiempo o es interrumpido si llega un hilo de mayor prioridad
- Este es el modelo más común en los sistemas operativos modernos y en la JVM de Java
- Ofrece mejor respuesta del sistema y equilibrio de carga

Multihilo Cooperativo



- Los hilos ceden el control voluntariamente, llamando a métodos como yield() o al bloquearse
- No hay interrupciones forzadas por parte del planificador
- Menor sobrecarga del planificador debido a la ausencia de interrupciones
- Mayor control explícito sobre los puntos donde se cede el control
- Riesgo de inanición y bloqueo del sistema si un hilo no coopera

En Java, se utiliza el modelo apropiativo con prioridades para la planificación de hilos.

Prioridades en Java

Sistema de Prioridades

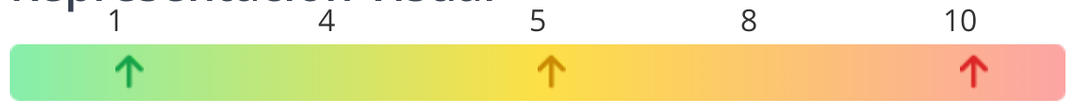
Java utiliza un modelo apropiativo con un sistema de prioridades que sugiere la importancia relativa de un hilo.

- **Thread.MIN_PRIORITY (1)**
  Prioridad mínima
- **Thread.NORM_PRIORITY (5)**
  Prioridad por defecto
- **Thread.MAX_PRIORITY (10)**
  Prioridad máxima

Métodos para Manipular Prioridades

```
public final void setPriority(int priority)
public final int getPriority()
```

Representación Visual



Limitaciones y Consideraciones

- Las prioridades son solo una **sugerencia** al sistema operativo
- La interpretación y uso puede variar entre diferentes JVMs y sistemas operativos
- No se debe depender exclusivamente de las prioridades para garantizar la corrección de un programa multihilo

Recomendación: Utiliza la sincronización como mecanismo principal para la corrección, no las prioridades.

Depuración Multihilo

Puntos de Interrupción en IDEs

- Permite pausar la ejecución de hilos específicos o todos los hilos
- Inspección de la pila de llamadas y variables (locales y compartidas)
- Funcionalidades avanzadas como "watchpoints" que pausan cuando una variable cambia

Análisis de Thread Dumps

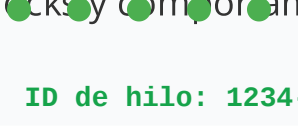
- Herramientas como `jstack` obtienen el estado de todos los hilos
- Útil para identificar deadlocks, hilos bloqueados o en espera
- Permite analizar la pila de llamadas de cada hilo en un momento dado

Logging Detallado

- Inserción de sentencias de log con ID/nombre del hilo
- Registro de marcas de tiempo y estado de variables
- Herramienta invaluable para reconstruir la secuencia de eventos

Herramientas de Profiling

- Java VisualVM, JProfiler y otras herramientas de monitorización
- Identificación de cuellos de botella y uso de CPU
- Análisis de contención de locks y comportamiento en tiempo de ejecución



Documentación en Aplicaciones Multihilo

La documentación es **crítica** en el desarrollo de aplicaciones multihilo debido a su complejidad inherente. Facilita el mantenimiento, la comprensión y la extensión segura del código.

Diseño de Concurrencia

Documenta cómo se han dividido las tareas en hilos y cómo interactúan entre sí. Incluye diagramas de flujo que muestren la colaboración entre componentes concurrentes.

Suposiciones sobre Ejecución

Registra suposiciones sobre la planificación de hilos, prioridades y comportamiento del sistema. Evita depender exclusivamente de las prioridades para la corrección.

Protocolos de Sincronización

Describe los mecanismos de sincronización utilizados (locks, semáforos, etc.) y cómo se aplican. Explica las secciones críticas y los recursos compartidos.

Compartición de Recursos

Documenta qué recursos comparten los hilos y cómo se garantiza la integridad de los datos. Incluye información sobre la visibilidad de variables y mecanismos de sincronización.

Mejores Prácticas

- Documenta las invariantes de concurrency
- Actualiza la documentación con el código
- Usa comentarios claros sobre secciones críticas

Proyecto Práctico: Contador Concurrente

Objetivo del Proyecto

Demostrar experimentalmente la existencia de una condición de carrera en un contador compartido y cómo la sincronización en Java puede resolver este problema.

Estructura del Proyecto

- Clase `Contador` sin sincronización
- Múltiples hilos intentando incrementar el contador
- Visualización de la condición de carrera
- Solución usando `synchronized`

Problema: Contador Sin Sincronización

500,000

⚠ Operación no atómica: `cuenta++` (lee, incrementa, escribe)

Solución: Contador con Sincronización

500,000

```
public synchronized void incrementar() {
    cuenta++;
}
```

✓ Exclusión mutua: solo un hilo puede ejecutar el método a la vez

Resultados Esperados

- ✗ Sin sincronización: Cuenta incorrecta (pérdida de incrementos)
- ✓ Con sincronización: Cuenta correcta (todos los incrementos registrados)

El Problema: Contador Sin Sincronización

Código con Problema

```
public class Contador { private int cuenta = 0; public void incrementar() { cuenta++; // Esta operación no es atómica } public int getCuenta() { return cuenta; } }
```

```
public class AppSinSincronizar { public static void main(String[] args) throws InterruptedException { Contador contador = new Contador(); int numHilos = 5; int incrementosPorHilo = 100000; HiloIncrementador[] hilos = new HiloIncrementador[numHilos]; for (int i = 0; i < numHilos; i++) { hilos[i] = new HiloIncrementador(contador, incrementosPorHilo); hilos[i].start(); } for (int i = 0; i < numHilos; i++) { hilos[i].join(); // Esperar a que todos los hilos terminen } int cuentaEsperada = numHilos * incrementosPorHilo; System.out.println("Cuenta final esperada: " + cuentaEsperada); System.out.println("Cuenta final obtenida: " + contador.getCuenta()); } }
```

Análisis del Problema

¿Qué hace `cuenta++`?

La operación `cuenta++` no es atómica. En realidad se compone de tres pasos:

1. Leer el valor actual de `cuenta
2. Incrementarlo
3. Escribir el nuevo valor

Condición de Carrera

Cuando múltiples hilos ejecutan `cuenta++` simultáneamente, es posible que:

- Múltiples hilos lean el mismo valor inicial
- Incrementen ese valor
- Escriban el mismo resultado
- Perdiendo incrementos

Resultado

Al ejecutar `AppSinSincronizar.java`

La Solución: Sincronización

Implementación

La solución a la condición de carrera es aplicar la palabra clave **synchronized** al método **incrementar()**:

```
public class Contador {
    private int cuenta = 0;

    // Método sincronizado
    public synchronized void incrementar() {
        cuenta++;
    }

    public int getCuenta() {
        return cuenta;
    }
}
```

¿Cómo funciona?

- El método **synchronized** garantiza la exclusión mutua
- Cada objeto en Java tiene un bloqueo intrínseco (monitor)
- Un hilo debe adquirir el bloqueo antes de ejecutar el método

Ejecución Sincronizada

Hilo 1 ejecuta incrementar()

↓

Hilo 2 espera bloqueado

↓

Hilo 1 libera el bloqueo

Beneficios de la Solución

- Garantiza que solo un hilo puede modificar la cuenta a la vez
- Evita la condición de carrera y la corrupción de datos
- El resultado será consistente: Cuenta final obtenida = Cuenta final esperada

Conclusiones

Conceptos Fundamentales

Comprender los hilos, su ciclo de vida y los estados es crucial para desarrollar aplicaciones concurrentes eficientes.

Depuración

La depuración multihilo presenta desafíos adicionales; técnicas como puntos de interrupción, logging y análisis de volcados de hilos son esenciales.

Sincronización

La sincronización es esencial para prevenir condiciones de carrera e inconsistencias de memoria, garantizando la integridad de los datos.

Documentación

Una documentación clara y detallada es crucial para el mantenimiento y la comprensión de la complejidad inherente a las aplicaciones multihilo.

Mecanismos de Sincronización

Java ofrece diversos mecanismos desde synchronized hasta clases avanzadas en java.util.concurrent.

Beneficios

La programación multihilo mejora la responsividad de la UI, aprovecha procesadores multinúcleo y optimiza operaciones de entrada/salida.

> La clave para aplicaciones robustas y eficientes con programación multihilo