
# PROGRAMACIÓN CONCURRENTE MULTIHILO

## PARADIGMAS DE PROGRAMACIÓN II

Carlos Rojas Sánchez

Licenciatura en Informática

Universidad del Mar

# Contenido

# Conceptos generales

# ¿Qué es la Programación Concurrente?

- Permite ejecutar múltiples tareas lógicas de forma simultánea o intercalada.
- Mejora el rendimiento, la eficiencia y la capacidad de respuesta de los programas.
- Se logra mediante el uso de **hilos** (threads).

# Concurrencia vs Paralelismo

| Concurrencia | Paralelismo |
| --- | --- |
| Intercala la ejecución de múltiples tareas. | Ejecuta múltiples tareas exactamente al mismo tiempo. |
| Funciona en un solo núcleo. | Requiere múltiples núcleos para ejecutarse realmente en paralelo. |
| Enfoque lógico. | Enfoque físico. |

# ¿Qué es un Hilo (Thread)?

- Unidad básica de ejecución dentro de un proceso.
- Varios hilos pueden compartir memoria y recursos del proceso.
- Los hilos pueden ejecutarse de forma concurrente.

# Ventajas del Multihilo

- Mejor uso de CPU multinúcleo.
- Mayor rendimiento en tareas pesadas.
- Interfaz gráfica más fluida.
- Permite tareas en segundo plano.

# Riesgos del Multihilo

- Condiciones de carrera (Race conditions).
- Interbloqueos (Deadlocks).
- Inanición (Starvation).
- Dificultad para depurar y mantener.

# Código embebido: hilo.cs

```
using System;
using System.Threading;

class Program {
 static void Main() {
  Thread t = new Thread(MetodoSecundario);
  t.Start(); // Inicia hilo

  for (int i = 0; i < 5; i++) {
    Console.WriteLine("Hilo principal: " + i);
    Thread.Sleep(500);
  }
 }

 static void MetodoSecundario() {
  for (int i = 0; i < 5; i++) {
    Console.WriteLine("Hilo secundario: " + i);
    Thread.Sleep(500);
  }
 }
}
```

# APIs para programación multihilo

- C# / .NET: `Thread`, `Task`, `async/await`, `Parallel`
- Java: `Thread`, `Runnable`, `ExecutorService`
- Python: `threading`, `asyncio`, `multiprocessing`
- C++: `std::thread`, `pthread`, `OpenMP`

# Aplicaciones comunes

- Juegos con múltiples entidades (IA, enemigos, colisiones).
- Aplicaciones con interfaz gráfica.
- Servidores web (manejo de múltiples clientes).
- Procesamiento de archivos en segundo plano.

# Resumen

- La programación concurrente multihilo es poderosa pero compleja.
- Aprovecha al máximo los procesadores actuales.
- Requiere una gestión cuidadosa de los recursos compartidos.

# Concepto de hilo

# Hilos en Programación

- Un **hilo (thread)** es una secuencia independiente de instrucciones dentro de un programa.
- Todos los hilos de un proceso comparten:
  - El mismo espacio de direcciones.
  - Recursos del sistema (archivos, memoria, variables globales).
- Se usan para ejecutar varias tareas de forma **concurrente o paralela**.

# Proceso vs Hilo

| Proceso | Hilo |
| --- | --- |
| Unidad de ejecución independiente. | Subunidad dentro de un proceso. |
| Tiene su propia memoria y recursos. | Comparte memoria y recursos del proceso. |
| Más costoso de crear. | Más ligero y rápido de crear. |
| Comunicación más compleja. | Comunicación sencilla (memoria compartida). |

# Creación de Hilos en C#

```
// Importar espacio de nombres
using System.Threading;

// Crear y ejecutar un hilo
Thread t = new Thread(MetodoSecundario);
t.Start(); // Inicia el hilo

void MetodoSecundario() {
    Console.WriteLine("Hola desde un hilo");
}
```

- Se utiliza la clase `Thread` del espacio de nombres `System.Threading`.
- El hilo ejecuta un método específico.

# Hilo principal vs hilo secundario

```
using System;
using System.Threading;

class Program {
 static void Main() {
  Thread t = new Thread(Secundario);
  t.Start();

  for (int i = 0; i < 3; i++) {
    Console.WriteLine("Principal: " + i);
    Thread.Sleep(500);
  }
 }

 static void Secundario() {
  for (int i = 0; i < 3; i++) {
    Console.WriteLine("Secundario: " + i);
    Thread.Sleep(500);
  }
 }
}
```

# Uso de Thread.Sleep()

- El método `Thread.Sleep(ms)` detiene temporalmente la ejecución del hilo actual.
- Sirve para simular tareas que consumen tiempo (como IO, cálculo, etc.).
- Permite observar la concurrencia entre hilos en ejemplos simples.

**Ejemplo:** `Thread.Sleep(1000);` detiene el hilo por 1 segundo.

# Estados de un Hilo

- **Unstarted**: El hilo ha sido creado pero aún no se ha iniciado.
- **Running**: El hilo está ejecutándose.
- **WaitSleepJoin**: El hilo está detenido temporalmente (por `Sleep` o `Join`).
- **Suspended / Blocked**: Esperando recursos compartidos.
- **Stopped / Aborted**: El hilo ha terminado su ejecución.

# ¿Por qué usar hilos?

- Ejecutar tareas en segundo plano sin congelar la aplicación.
- Mejorar el rendimiento aprovechando CPUs multinúcleo.
- Separar tareas independientes (descarga, cálculo, entrada/salida).
- Aplicaciones más receptivas e interactivas.

# Precauciones al usar hilos

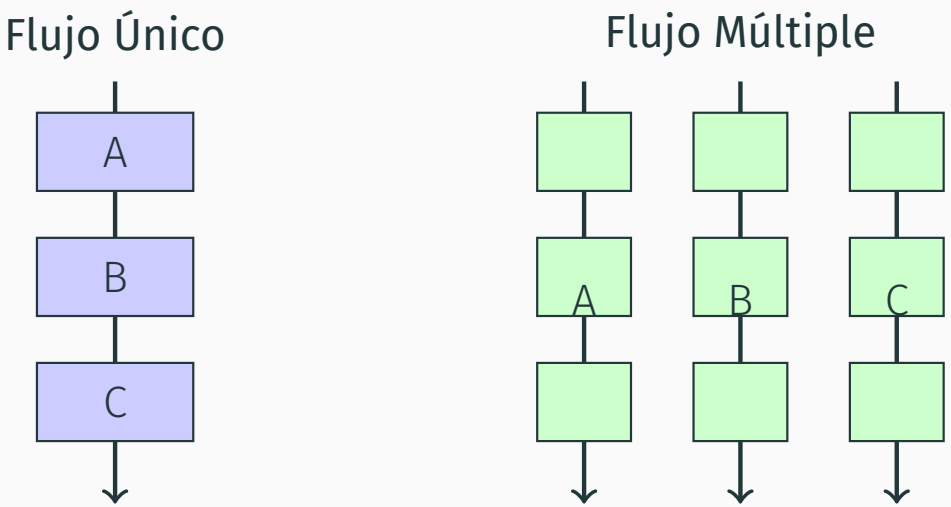
- **Race conditions:** cuando dos hilos acceden al mismo recurso al mismo tiempo.
- **Deadlocks:** cuando varios hilos esperan recursos bloqueados.
- **Sincronización:** necesaria para evitar errores en variables compartidas.
- **Dificultad para depurar.**

# Comparación de un programa de flujo único contra uno de flujo múltiple

# Flujo Único vs Flujo Múltiple

| Flujo Único (Monohilo) | Flujo Múltiple (Multihilo) |
|---|---|
| Ejecuta una tarea a la vez. | Ejecuta varias tareas simultáneamente. |
| Secuencial: bloquea mientras espera. | Concurrente: tareas independientes avanzan en paralelo. |
| Fácil de depurar. | Requiere cuidado con sincronización. |
| No aprovecha varios núcleos. | Aprovecha CPUs multinúcleo. |
| Ejemplo: Calculadora simple. | Ejemplo: Reproductor con GUI + carga en segundo plano. |

# Representación visual



# Creación y control de hilos

# Creación de un Hilo en C#

- Se utiliza la clase `Thread` del espacio de nombres `System.Threading`.
- La instancia de `Thread` recibe un delegado al método que ejecutará.
- Luego se invoca `Start()` para iniciar el hilo.

```
using System.Threading;

Thread hilo = new Thread(MetodoSecundario);
hilo.Start();
```

# Método que ejecuta el hilo

```
static void MetodoSecundario() {
  for (int i = 1; i <= 3; i++) {
    Console.WriteLine("Mensaje del hilo: " + i);
    Thread.Sleep(500); // Simula trabajo
  }
}
```

- El método debe tener firma compatible con `ThreadStart`.
- Puede ejecutar cualquier tarea: cálculos, descargas, etc.

# Control básico de hilos

- `Start()`: inicia la ejecución del hilo.
- `Join()`: espera a que el hilo termine.
- `Sleep(ms)`: pausa temporal del hilo actual.

```
Thread t = new Thread(Metodo);
t.Start();

t.Join(); // Espera que termine
Console.WriteLine("Hilo terminado");
```

# Pasar parámetros a un hilo

```
Thread t = new Thread(new ParameterizedThreadStart(
    MostrarMensaje));
t.Start("Hola mundo");

static void MostrarMensaje(object mensaje) {
  Console.WriteLine("Mensaje: " + mensaje);
}
```

- Se usa `ParameterizedThreadStart`.
- El método debe aceptar un `object` como parámetro.

# Crear múltiples hilos

```
for (int i = 1; i <= 3; i++) {
  int n = i;
  Thread t = new Thread(() => {
    Console.WriteLine("Hilo " + n);
  });
  t.Start();
}
```

- Cada hilo puede ejecutar una tarea distinta o la misma con distintos datos.
- Se puede usar expresión lambda para simplificar.

# Sleep y Join

```
Thread t = new Thread(() => {
  Thread.Sleep(2000);
  Console.WriteLine("Hilo terminado");
});

t.Start();

t.Join(); // Espera que t termine
Console.WriteLine("Fin del programa");
```

- `Sleep`: simula trabajo o pausa.
- `Join`: sincroniza el hilo con el flujo principal.

# Resumen

- Los hilos se crean con la clase `Thread`.
- Se controlan usando `Start()`, `Sleep()`, `Join()`.
- Se pueden pasar parámetros mediante `ParameterizedThreadStart`.
- Controlar la ejecución permite sincronizar tareas concurrentes.

# Sincronización de hilos

# ¿Qué es la sincronización de hilos?

- Es el proceso de controlar el acceso a recursos compartidos por múltiples hilos.
- Evita que dos o más hilos modifiquen al mismo tiempo una misma variable, archivo, memoria, etc.
- Sin sincronización, pueden ocurrir errores como:
  - Condiciones de carrera (*race conditions*)
  - Resultados inconsistentes
  - Fallos aleatorios

# Problema sin sincronización

```
int contador = 0;

void Incrementar() {
  for (int i = 0; i < 1000; i++) {
    contador++; // Acceso no seguro
  }
}

Thread t1 = new Thread(Incrementar);
Thread t2 = new Thread(Incrementar);
t1.Start(); t2.Start();
t1.Join(); t2.Join();

Console.WriteLine(contador); // ¿2000? No siempre
```

- Varios hilos acceden y modifican ‘contador‘ al mismo tiempo.
- El resultado final es impredecible.

# Solución: sincronización con `lock`

```
int contador = 0;
object candado = new object();

void Incrementar() {
  for (int i = 0; i < 1000; i++) {
    lock (candado) {
      contador++; // Acceso seguro
    }
  }
}
```

- `lock(obj)` asegura que solo un hilo entre al bloque a la vez.
- El objeto `candado` se usa como clave de bloqueo.

# ¿Cómo funciona `lock`?

- Si un hilo entra al bloque `lock`, los demás deben esperar.
- Evita que dos hilos entren al mismo bloque crítico simultáneamente.
- Es una forma simple de proteger secciones críticas del código.
- Internamente usa una estructura de sincronización llamada **monitor**.

**Importante:** No usar objetos mutables como candado (por ejemplo, strings compartidos).

# Alternativas a `lock` en C#

- `Monitor.Enter / Monitor.Exit` – equivalente manual de ‘lock‘.
- `Mutex` – sincronización entre procesos.
- `Semaphore / SemaphoreSlim` – controlar acceso a recursos limitados.
- `ReaderWriterLockSlim` – para múltiples lectores y un solo escritor.
- `Interlocked` – operaciones atómicas (incremento, decremento, etc.).

# Uso de `Interlocked` para operaciones simples

```
using System.Threading;

int contador = 0;

void Incrementar() {
  for (int i = 0; i < 1000; i++) {
    Interlocked.Increment(ref contador);
  }
}
```

- Ideal para operaciones simples sobre variables enteras.
- Más rápido que `lock`, pero solo para casos limitados.

# Resumen

- Los hilos pueden interferir si acceden a recursos compartidos sin control.
- `lock` permite sincronizar fácilmente el acceso a secciones críticas.
- Existen herramientas más avanzadas según el escenario: `Mutex`, `Semaphore`, etc.
- Siempre evalúa la necesidad de sincronizar para evitar errores difíciles de depurar.