Sincronización en C# y Java

Sistemas Distribuidos
Grado de Ingeniería Informática.

ETSI ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA

DEPARTAMENTO DE TECNOLOGÍAS DE LA INFORMACIÓN

Universidad de Huelva

Contenido

1. Conceptos.
2. Procesos Pesados vs. Ligeros.
3. Hebras en c#.
    a. Sincronización con Monitores.
    b. Sincronización de Métodos.
    c. Sincronización mediante operaciones Atómicas.
4. Hebras en Java.
    a. Sincronización con Monitores.

Conceptos

- *Programa* es un archivo ejecutable residente en un dispositivo de almacenamiento permanente. Es ejecutado mediante el SO.
- *Proceso* es un programa en ejecución. Los procesos pueden crear procesos hijos clonándose mediante llamadas específicas al SO.
- Un proceso es un programa en ejecución, que se ejecuta secuencialmente (no más de una instrucción a la vez).
- El SO tiene una abstracción del mismo con la siguiente información:

- Identificación del proceso.
- Identificación del proceso padre.
- Información sobre el usuario y grupo.
- Estado del procesador.
- Información de control de proceso.
  - Información del planificador.
  - Memoria asignada.
  - Recursos asignados.

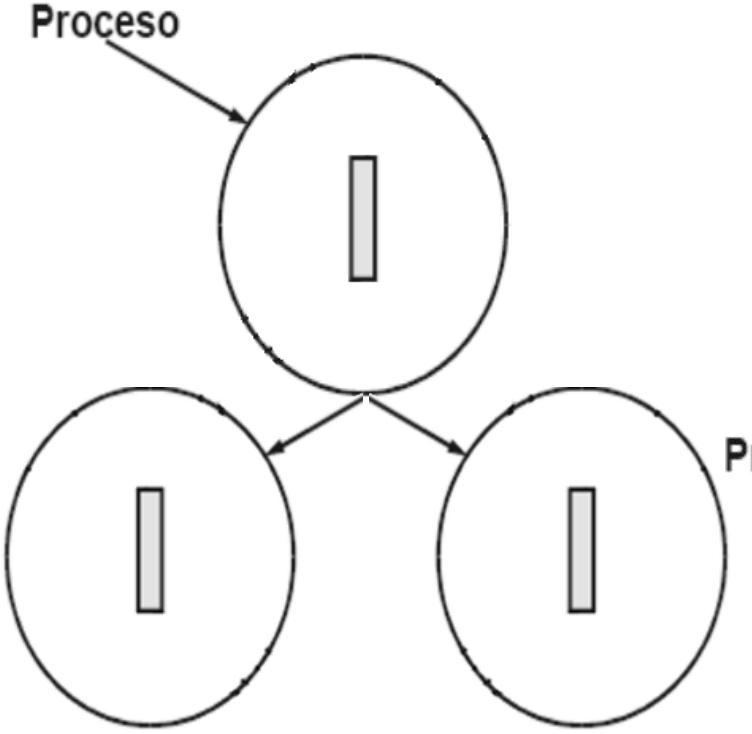
Proceso Pesado vs. Ligero

- *Proceso Pesado*: La clonación de procesos supone:
  - ☹ Copia de todos los recursos del proceso.
  - ☹ Coste de comunicaciones entre procesos

- *Proceso Ligero*: Es un flujo de control perteneciente a un proceso y se le suele denominar también hebras o hilos
  - ☺ La sobrecarga de recursos en su creación es menor. Solo se clona las variables compartidas.
  - ☺ El coste de comunicación entre los hilos es prácticamente nulo
  - ☺ Cada hilo pertenece a un proceso pesado
  - ☺ Todos los hilos comparten su espacio de direccionamiento
  - ☺ Cada hilo dispone de su propia política de planificación, pila y contador de programa

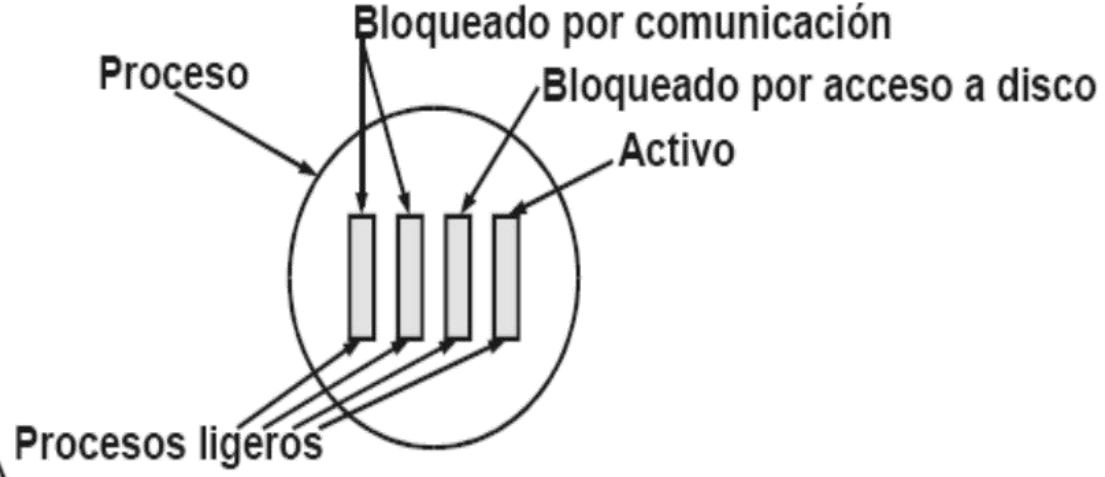
Proceso Pesado vs. Ligero

Proceso Pesado



Clonación Completa

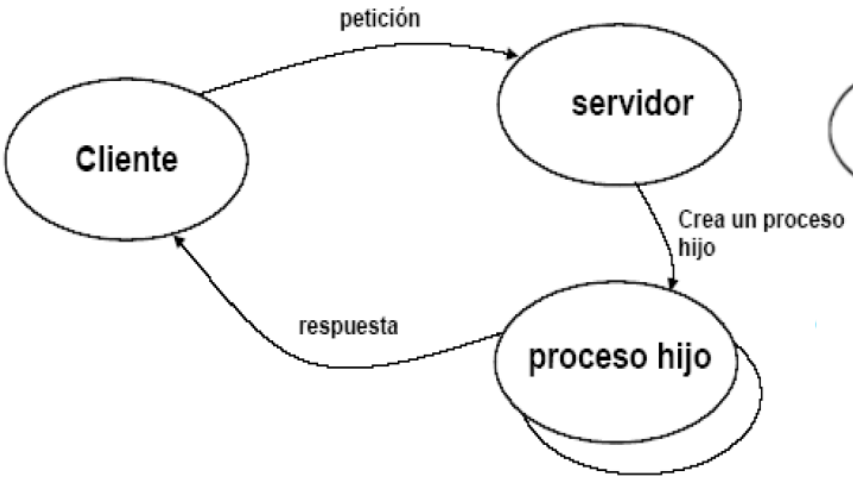
Proceso Ligero



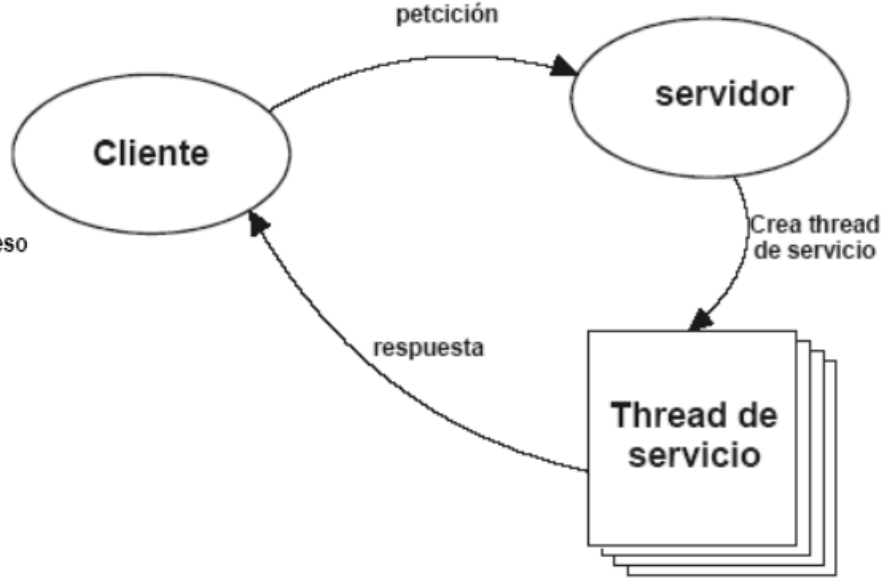
Clonación Parcial,
Solo las variables compartidas

Proceso Pesado vs. Ligero

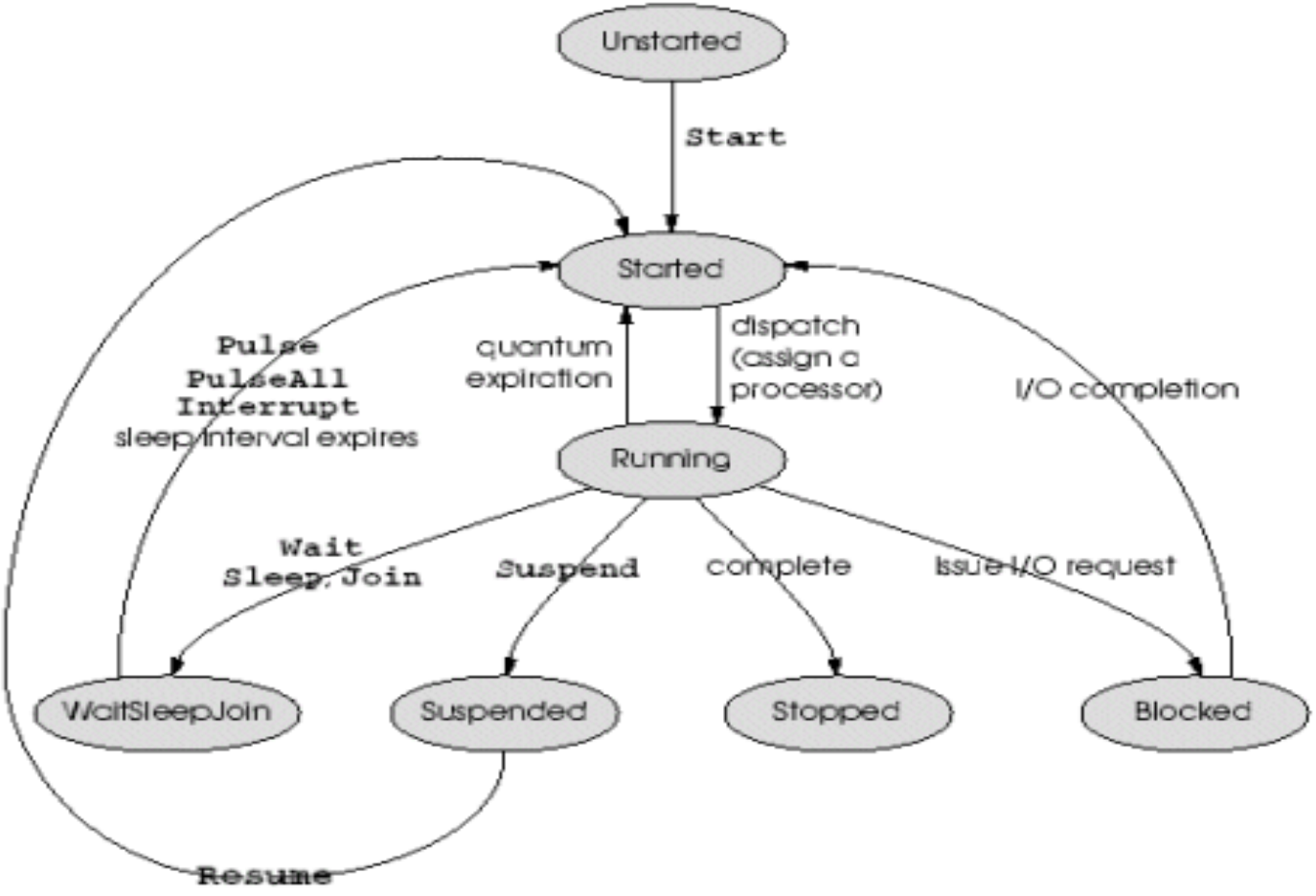
Proceso Pesado



Proceso Ligero



Ciclo de vida de las Hebras



Hebras en c#

- Creación. Se puede crear una hebra utilizando:

1. Creando un delegado del tiempo *ThreadStart*. El delegado almacena la referencia al método que se ejecutara en un hilo. Este método no puede tener parámetros y no debe devolver valores.

2. Un objeto *Thread* que ejecute el subproceso que enlaza el delegado.

```
using System.Threading;
...
...
void M() {
... Acciones ...
}
...
ThreadStart ts = new ThreadStart(M);
Thread t = new Thread(ts);
t.Start();
```

Hebras en c#

- Detención temporal de la ejecución de una hebra.
  - ✓*Sleep*: detiene la ejecución de la hebra actual durante un número determinado de milisegundos.
  - ✓*Suspend*: detiene de forma indefinida la ejecución de la hebra (hasta que se llame al método *Resume*). Este método está obsoleto, y no debería emplearse para sincronizar hebras, pues no hay forma de saber qué está haciendo el hilo cuando se suspende.
  - ✓*Join*: detiene la ejecución de la hebra hasta que finalice otra hebra.
- Finalización de la ejecución de una hebra
  - ✓*Abort*: interrumpe inmediatamente la ejecución de la hebra y genera una excepción de tipo *ThreadAbortException*.
  - ✓*Interrupt*: interrumpe la ejecución de la hebra cuando la hebra esté detenida y genera una excepción de tipo *ThreadInterruptedException* .

Ejemplo

- Detención temporal de la ejecución de una hebra.

```
static void Main(string[] args){
   Thread.CurrentThread.Name = "Principal";
   Thread h1 = new Thread(new ThreadStart(RutinaHilo1));
   h1.Name = "Hilo1";
   h1.Start();
   h1.Join(); //espero por el hilo 1
   Console.WriteLine("Se acabó la Main()");
}
static void RutinaHilo1(){
    for (int i = 1; i < 10; i++){
         Console.WriteLine("Soy el hilo 1");
         Thread.Sleep(1000);
    }
}
```

Ejemplo

- Finalización de la ejecución de una hebra.

```
static void Main(string[] args){
    Thread h1 = new Thread(new ThreadStart(RutinaHilo1));
    h1.Start();
    . . .
    h1.Abort();
}
static void RutinaHilo1(){
    try{
        for (int i = 1; i < 10; i++){
            Console.WriteLine("Soy el hilo 1");
            Thread.Sleep(1000);
         }
    }catch (Exception e){
        Console.WriteLine("Atrapada excepción {0} en hilo {1}",
                e.ToString(), Thread.CurrentThread.Name);
    }
}
```

Sincronización de Hebras: Monitores

- El Modelo de sincronización utilizado en la plataforma .NET es análogo al utilizado por el lenguaje de programación Java:
  - Los objetos Monitor tienen la capacidad de sincronizar el acceso a una región de código mediante la obtención y liberación de un *bloqueo* en un objeto concreto con los métodos *Monitor.Enter*, *Monitor.TryEnter*, *Monitor.Exit*.
  - Mecanismos para bloquear la ejecución de una hebra hasta que se cumpla una condición se realiza mediante los métodos *Monitor.Wait*, *Monitor.Pulse* y *Monitor.PulseAll*.

Sincronización de Hebras: Monitores

- Uso del método *Monitor.Enter*.

```
private object x;
. . .
Monitor.Enter(x);
try {
    // Clodigo que se necesita proteger mediante el monitor.
}
finally {
    // Siempre utilizar Finally para asergurarnos que se sale del
    // Monitor.
    Monitor.Exit(x);
}
```

Sincronización de Hebras: Monitores

- Uso del método *Monitor.TryEnter*.

```
if (Monitor.TryEnter(this, 300)) { //Intenta entrar en 300 segundos
    try {
        // Clodigo que se necesita proteger mediante el monitor.
    }
    finally {
        Monitor.Exit(this);
    }
}
else {
    // Código que se ejecutará si el time out se cumple
}
```

Sincronización de Hebras: Monitores

- Exclusión mutua con *lock*.

```
void M() {
   // Entrada en la sección crítica
   ...
   Monitor.Enter(this);
   //Sección crítica
   ...
   // Salida de la sección crítica
   Monitor.Exit(this);
   ...
}
...
```

```
void M() {
   // Entrada en la sección crítica
   ...
   lock(this){
   //Sección crítica
   ...
   // Salida de la sección crítica
   }
   ...
}
...
```

Sincronización de Hebras: Condiciones

- Uso del método *Monitor.Wait*(obj).
  - ✓Queda a la espera de que otra hebra invoque al método *Monitor.Pulse*(obj) o *Monitor.PulseAll*(obj).
  - ✓Una llamada a *Monitor.Wait*(obj) estará dentro de una sección crítica definida sobre el objeto obj de lo contrario se produce una excepción de tipo *SynchronizationLockException*.
  - ✓La hebra desbloquea el acceso al objeto obj de forma que otras hebras puedan acceder a dicho objeto

Sincronización de Hebras: Condiciones

- Uso del método *Monitor.Pulse*(obj).
  - ✓Despierta a una hebra que esté esperando al objeto obj. La hebra recibe una notificación del cambio de estado y puede reanudar su ejecución en cuanto recupere el acceso en exclusiva al objeto obj.

  - ✓No se garantiza que la hebra que sale de su espera cumpla la condición que la llevó a esperar. Por tanto, habrá que volver a comprobar si se cumple o no la condición:

```
while (!cond)
        Monitor.Wait(obj);
```

- Uso del método *Monitor.PulseAll*(obj).
  - ✓Despierta a todas las hebras que estén esperando al objeto obj.

Ejemplo Productor/Consumidor en C#

```
class Buffer {
   int[] buf = new int[10];
   int head = 0, tail = 0;
   int n = 0;

   void put(int data) {
      lock(this){
         while (n == 10)
            Monitor.Wait(this);
         buf[tail] = data;
         tail = (tail+1)%10;
         n++;
         Monitor.PulseAll(this);
      }
   }

   int get() {
      lock(this){
      while (n == 0)
         Monitor.Wait(this);
      int data = buf[head];
      head = (head+1)%10;
      n--;
      Monitor.PulseAll(this);
      return data;
      }
   }
}
```

Sincronización de Métodos

- Si se pretende sincronizar un método entero, en vez de usar bloqueos en cada método de tipo *lock* o *Monitor*, se puede usar un atributo para marcar el método entero como sincronizado:

```
using System.Runtime.CompilerServices;

[MethodImpl(MethodImplOptions.Synchronized)]
public void metodosincronizado()
{
   // cuerpo del método sincronizado
}
```

- Con esto se consigue sincronizar el acceso al método completo con menos tiempo y esfuerzo.

Sincronización Mediante Interlocked

- Los métodos de esta clase permiten realizar operaciones sobre variables considerándolas como atómicas.
- Los métodos *Increment* y *Decrement* aumentan o disminuyen una variable y almacenan el valor resultante en una única operación. En la mayoría de los equipos, el incremento de una variable no es una operación atómica.
- El método *Exchange* intercambia los valores de las variables especificadas atómicamente.
- El método *CompareExchange* combina dos operaciones: *compara* dos valores y *almacena* un tercer valor en una de las variables, en función del resultado de la comparación. Las operaciones de comparación e intercambio se realizan como una operación atómica.
- El método *add* agrega dos enteros y reemplaza el primer entero con la suma, como una operación atómica.

Sincronización Mediante Interlocked

```
class CountClass{
    static int unsafeInstanceCount = 0;
    static int   safeInstanceCount = 0;

  ..........
    public CountClass()    {
        unsafeInstanceCount++;
        Interlocked.Increment(ref safeInstanceCount);
    }

    ~CountClass()    {
        unsafeInstanceCount--;
        Interlocked.Decrement(ref safeInstanceCount);
    }
}
```

Hebras en Java

- Creación. Se puede crear de dos maneras:
  1. Heredando de la clase Thread.
  2. Implementando la interfaz Runnable.

```
Class MyThread extend Thread{
  public void run(){
    //Código que se ejecuta en una
    //hebra
  }
}

class Ejemplo {
  MyThread Hilo = new MyThread();
  t.Start();
}
```

```
Class MyThread implements Runnable{
   Thread Hilo;
   public void start(){
      Hilo=new Thread(this);
      Hilo.start();
   }
   public void run(){
    //Código que se ejecuta en una
    //hebra
  }
}
class Ejemplo {
  MyThread Hilo = new MyThread();
  t.Start();
}
```

Hebras en Java

- Detención temporal de la ejecución de una hebra.
  - ✓*Sleep*: detiene la ejecución de la hebra actual durante un número determinado de milisegundos.
  - ✓*Suspend*: detiene de forma indefinida la ejecución de la hebra (hasta que se llame al método *Resume*). *Método obsoleto.*
  - ✓*wait*: detiene la ejecución de la hebra hasta que otra realice un *notify* o *notifyAll*. Permite además opcionalmente un *timeout*.
  - ✓*join*: detiene la ejecución de la hebra hasta que otra se muera. Permite además opcionalmente un *timeout*.
- Finalización de la ejecución de una hebra.
  - ✓*Interrupt*: interrumpe la ejecución de la hebra cuando la hebra esté detenida y genera una excepción de tipo *InterruptedException* .

Sincronización de Hebras: Monitores en Java

- Java utiliza un monitor para cada objeto que posea un método declarado *synchronized*.
- Cuando una hebra está ejecutando un método declarado *synchronized* se convierte en propietaria del *monitor* asociado al objeto, evitando que cualquier otra hebra ejecute el mismo u otro método declarado como *synchronized*.
- Al igual que c#, una hebra dormida con *wait*, cuando despierta por la ejecución por otra hebra del método *notify* o *notifyAll*, no garantiza que la condición que la llevó a dormirse en un principio, se cumpla nuevamente, por lo que deberá comprobar nuevamente dicha condición. Para ello es normal englobar la operación *wait* en un bucle que compruebe las condiciones antes de entrar en la sección crítica.

```
while (!cond)
    wait();
```

Ejemplo Productor/Consumidor en Java

```
public class Buffer {
    private int[] Datos;
    private int Tamanio, Cabeza, Cola, NElementos;
    public Buffer(int Tamanio) {
          Datos=new int[Tamanio];
          this.Tamanio=Tamanio;
          Cabeza=Cola=NElementos=0;
    }
    public synchronized void Poner(int Dato) {
        while (NElementos==Tamanio){
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("Poner interrumpido");
            }
        }
        Datos[Cola]=Dato;
        Cola=(Cola+1)%Tamanio;
        NElementos++;
        notify();
    }
```

Ejemplo Productor/Consumidor en Java

```
    public synchronized int Sacar() {
        while (NElementos==0) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("Sacar interrumpido");
            }
        }
        int Elemento=Datos[Cabeza];
        Cabeza=(Cabeza+1)%Tamanio;
        NElementos--;
        notify();
        return Elemento;
    }
}
```