



Departamento de Programación
Facultad de Informática
Universidad Nacional del Comahue



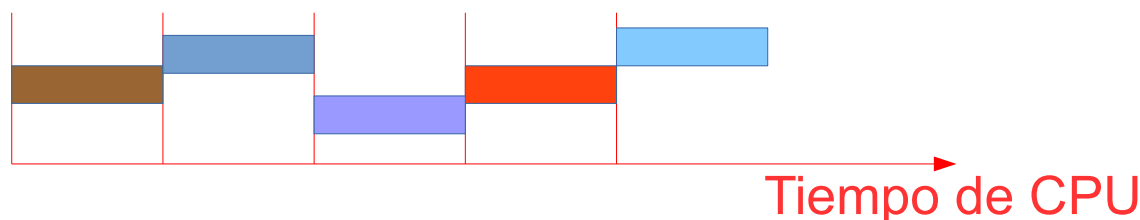
Programación Concurrente



Características – Mecanismos para la exclusión mutua

Hardware

- Sistema monoprocesador: La concurrencia se produce gestionando el tiempo de procesador para cada proceso.



- Sistemas multiprocesador: Un proceso en cada procesador

- Con memoria compartida (procesamiento paralelo)

Fuertemente acoplados

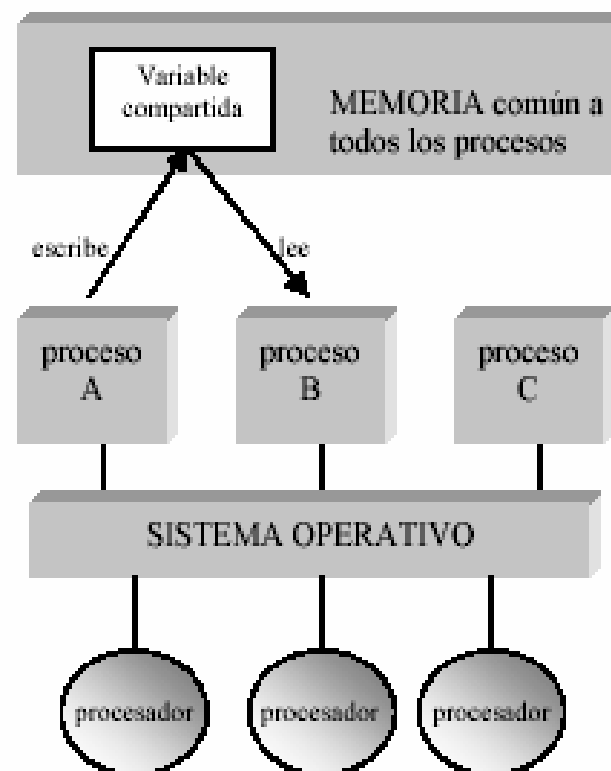
- Memoria local a cada procesador (sist.distribuidos)

Debilmente acoplados

Sistemas estrechamente acoplados (Multiprocesadores)

La sincronización y comunicación entre procesos se suele hacer mediante variables compartidas. Procesadores comparten memoria y reloj.

- **Ventaja:** aumento de velocidad de procesamiento con bajo coste.
- **Inconveniente:** Escalable sólo hasta decenas o centenares de procesadores

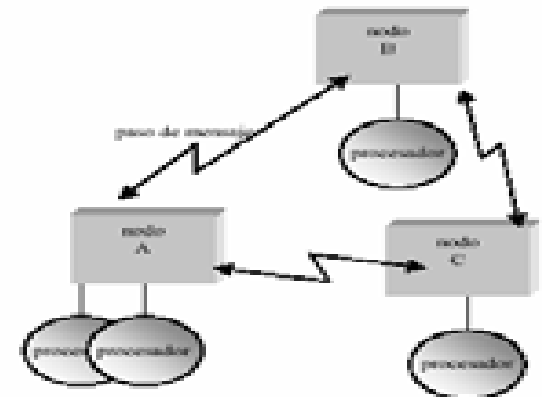


Sistemas débilmente acoplados (Distribuidos)

- Múltiples procesadores conectados mediante una red
- Los procesadores no comparten memoria ni reloj
- Los sistemas conectados pueden ser de cualquier tipo
- Escalable hasta millones de procesadores (ej. Internet)

La forma natural de comunicar y sincronizar procesos es mediante el uso de paso de mensajes.

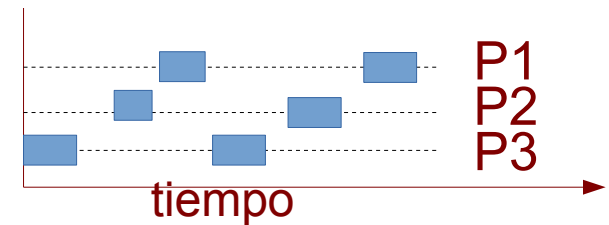
- **Ventaja:** compartición de recursos dispersos, aumento de velocidad de ejecución, escalabilidad ilimitada, mayor fiabilidad, alta disponibilidad



Concurrencia vs. Paralelismo

– Concurrencia “interleaved”

- Procesamiento simultáneo lógicamente
- Ejecución intercalada (cant procesadores < cant procesos)
- “Seudo-paralelismo”



– Concurrencia simultánea

- Procesamiento simultáneo físicamente
- Requiere un sistema multiprocesador o multicore
- Paralelismo “full” (igual cantidad de procesadores que procesos)





Un programa concurrente

Puede ser ejecutado por...

- **Multiprogramación:**
los procesos comparten uno o más procesadores
- **Multiprocesamiento:**
cada proceso corre en su propio procesador pero con **memoria compartida**
- **Procesamiento distribuido:**
cada proceso corre en su propio procesador **conectado a los otros a través de una red**



Sistemas de naturaleza concurrente

- **Sistemas de control:** Captura de datos, análisis y actuación (sistemas de tiempo real).
- **Servidores web** que son capaces de atender varias peticiones concurrentemente, servidores de chat, email, etc.
- **Aplicaciones basadas en GUI:** El usuario hace varias peticiones a la aplicación gráfica (Navegador web).
- **Simulación**, o sea programas que modelan sistemas físicos con autonomía.
- Sistemas **Gestores de Bases de Datos**.
- Sistemas **operativos** (controlan la ejecución de los usuarios en varios procesadores, los dispositivos de E/S, etc)



Objetivos de los sistemas concurrentes

- **Ajustar el modelo** de arquitectura de hardware y software al problema del mundo real a resolver.
- **El mundo real ES CONCURRENTE**
- **Incrementar la performance**, mejorando los tiempos de respuesta de los sistemas de procesamiento de datos, a través de un enfoque diferente de la arquitectura física y lógica de las soluciones.
- Se puede mejorar la velocidad de ejecución, mejor utilización de la CPU de cada procesador, y explotación de la concurrencia inherente a la mayoría de los problemas reales.



Incumbencias de la PC

- Los procesos pueden “competir” o “colaborar” entre sí por los recursos del sistema.

La programación concurrente (PC) se encarga del estudio de las nociones de ejecución concurrente, así como de sus **problemas de comunicación y sincronización.**

Lenguajes concurrentes

Incorporan características que permiten expresar la concurrencia directamente.

Incluyen mecanismos de sincronización

Incluyen mecanismos de comunicación entre procesos



Cómo expresar la concurrencia

- Sentencia concurrente:

cobegin

P; Q; R

coend;

- Objetos que representan procesos:

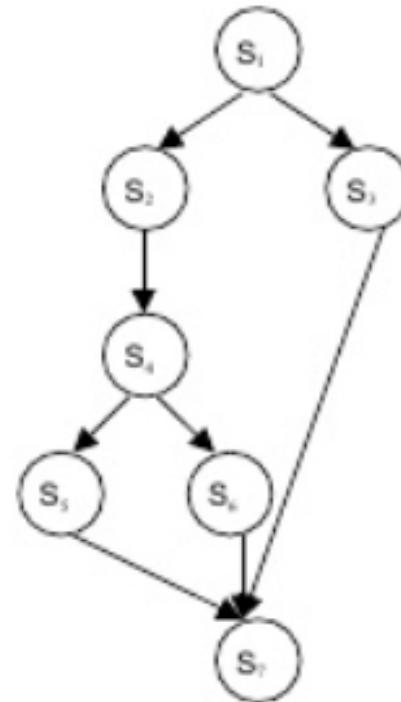
task A is begin P; end;

task B is begin Q; end;

task C is begin R; end;

- Notación gráfica:

Grafos de precedencia



Problemas de la PC

¿qué significa?

- ¿Pueden P1 y P2 ejecutarse de forma concurrente y determinista?
- ¿Cómo puedo saberlo?

```
proceso P1;  
    var i: integer;  
begin  
    for i:=1 to 5 do ...  
end;  
  
proceso P2;  
    var j: integer;  
begin  
    for j:=6 to 15 do ...  
end;
```

depende...
¿sobre quién actúa P1?
¿sobre quién actúa P2?



Problemas de la PC

- ¿Pueden P1 y P2 ejecutarse de forma concurrente y determinista?

Completamos P1 y P2

```
proceso P1;  
    var i: integer;  
begin  
    for i:=1 to 5 do x:= x+1;  
end;  
  
proceso P2;  
    var j: integer;  
begin  
    for j:=6 to 15 do x:= x+1;  
end;
```

```
Proceso P0  
x= 0;  
cobegin  
    P1;  
    P2;  
coend;  
Escribir x  
end;
```

x es una *variable compartida*
P1 actúa sobre x,
con acciones de
lectura y escritura
P2 también

Problemas de la PC

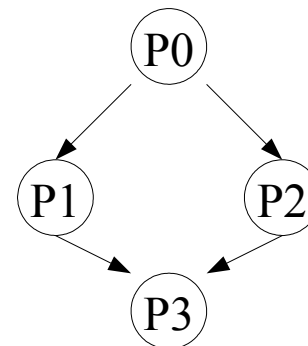
- Es más difícil analizar y verificar un algoritmo concurrente por el no determinismo.
- Que existan varias posibilidades de salida NO significa necesariamente que un programa concurrente sea incorrecto

```
Proceso P0  
x := 100;  
End;
```

```
proceso P1;  
x := x+10;  
end;
```

```
proceso P2;  
  Si x>100 escribir(x)  
  Sino escribir (x-50)  
end;
```

```
Proceso P3  
escribir ('fin')
```



¿Cual es la salida?

Problemas de la PC

Proceso P0

```
x:= 100;  
end;
```

Proceso P1

```
x:= x + 10;  
end;
```

proceso P2;

```
Si x>100 escribir(x)  
Sino escribir (x-50)  
end;
```

Proceso P3

```
escribir ('fin')  
end
```

Problemas de la PC

Proceso P0

```
x := 100;  
end;
```

Proceso P1

```
x := x + 10;  
end;
```

proceso P2;

```
Si x > 100 escribir(x)  
Sino escribir (x-50)  
end;
```

Proceso P3

```
escribir ('fin')  
end
```

- Se ejecuta P0, P1, P2, P3

$x=100$, $x = x+10$ (110), $\text{escribir}(x) \rightarrow 110$,
 escribir "fin"

- Se ejecuta P0, P2, P1, P3

$x=100$, $x > 100$ (no), $\text{escribir}(x-50) \rightarrow 50$
 escribir "fin"

- Se ejecuta P0, P2 (solo $x > 100$), P1,
continúa P2, P3

$x=100$, $x > 100$ (no), $x = x+10$ (110),
 $\text{escribir}(x-50) \rightarrow 60$
 escribir "fin"

INDETERMINISMO

Concurrencia en Java

Proceso P0

```
x := 100;  
end;
```

Proceso P1

```
x := x + 10;  
end;
```

proceso P2;

```
Si x > 100 escribir(x)  
Sino escribir (x-50)  
end;
```

Proceso P3

```
escribir ('fin')  
end
```

Para implementarlo en Java ...
tendría que considerar:

- un hilo para P1 y
- otro hilo para P2
- para P0 y P3 no necesito crear hilos,
se ocupa de ellos el
hilo principal de ejecución



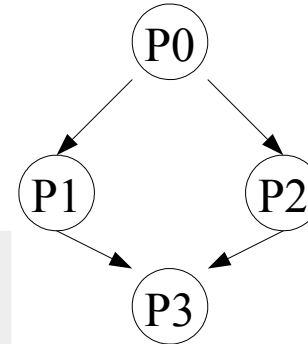
El *hilo principal de ejecución* corresponde a la ejecución del método `main` de cualquier programa Java.

Condiciones para la concurrencia

```
Proceso P0                Proceso P3;
x:= 0;                    escribir('fin');
End;                      end;

proceso P1;
    var i: integer;
begin
    for i:=1 to 100 do x:= x+1;
end;

proceso P2;
    var j: integer;
begin
    for j:=6 to 150 do x:= x+1;
end;
```



P1 y P2 no son independientes, comparten la variable x





Concurrencia en Java

```
package hilos;
public class Datos {
    private int dato = 0;
    public Datos(int nro) {
        dato = nro;
    }
    public int getDato() {
        return dato;
    }
    public void setDato(int valor) {
        dato = valor;
    }
    public boolean verificar(int valor) {
        return dato > valor;
    }
}
```

Procesos compartiendo algo

```
package hilos;

public class TestMiHilo{
    public static void main(String[] args) {

        Datos x= new Datos(100);
        ProcesoUno pUno = new ProcesoUno(x);
        ProcesoDos pDos = new ProcesoDos(x);
        Thread hilo1 = new Thread(pDos);
        Thread hilo2 = new Thread(pUno);
        hilo1.start();
        hilo2.start();
        System.out.println("fin");
        System.out.println("vuelta en el
main");
    }
}
```

Procesos compartiendo algo

Se creará un hilo Principal de ejecución que se ejecutará concurrentemente con los otros

hilos creados

```
package hilos;
```

```
public class TestMiHilo{  
    public static void main(String[] args) {  
  
        Datos x= new Datos(100);  
        ProcesoUno pUno = new ProcesoUno(x);  
        ProcesoDos pDos = new ProcesoDos(x);  
        Thread hilo1 = new Thread(pDos);  
        Thread hilo2 = new Thread(pUno);  
        hilo1.start();  
        hilo2.start();  
        System.out.println("fin");  
        System.out.println("vuelta en el main");  
    }  
}
```

Concurrencia en Java

```
package hilos;
```

```
public class TestMiHilo{  
    public static void main(String[] args)  
    {  
        Datos x= new Datos(100);  
        ProcesoUno pUno = new ProcesoUno(x);  
        ProcesoDos pDos = new ProcesoDos(x);  
        Thread hilo1 = new Thread(pDos);  
        Thread hilo2 = new Thread(pUno);  
        hilo1.start();  
        hilo2.start();  
        System.out.println("fin");  
        System.out.println("vuelta en el main");  
    }  
}
```

Se crean 2 objetos Thread,
pero ... no hay hilos de *ejecución*
hasta que se envia el mensaje
start a los objetos

Los hilos reciben
el mensaje de
“*estar listos*”,
pasan a estado
“*runnable*”

esperando por su turno para ejecutar el método *run()*



Concurrencia en Java

```
package hilos;

public class ProcesoUno implements Runnable{

    private Datos unDato;

    public ProcesoUno(Datos unD) {
        unDato = unD;
    }

    public void run() {
        System.out.println("estoy en ProcesoUno");
        if (unDato.getDato() > 100)
            System.out.println(unDato.getDato());
        else
            System.out.println(unDato.getDato()-50);
    }
}
```



Concurrencia en Java

```
package hilos;

public class ProcesoDos implements Runnable{
    private Datos unDato;

    public ProcesoDos(Datos unD) {
        unDato = unD;
    }

    public void run() {
        System.out.println("estoy en ProcesoDos");
        unDato.setDato(unDato.getDato()+10);
    }
}
```




Problemas en lenguajes de alto nivel

- En **ejecuciones concurrentes**
 - Diferentes posibilidades en cuanto al **orden de ejecución**
 - El resultado **puede** ser **incorrecto**
 - Pueden ser necesarias **ciertas restricciones** al **orden de ejecución**



Generalmente existen partes de código con **variables compartidas** y que deben ejecutarse en **exclusión mutua**

Problemas en lenguajes de alto nivel

Generalmente existen partes de código con **variables compartidas** y que deben ejecutarse en **exclusión mutua**, es decir tomar las operaciones que actúan sobre la variable compartida como **atómicas**.

...

contador.incrementar();

...



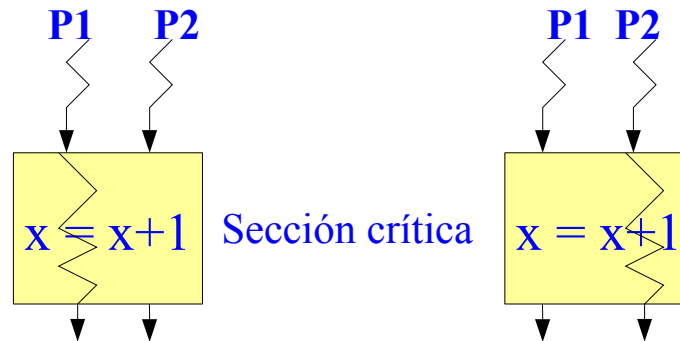
Sección crítica

Hay que “controlar” el acceso a la variable (recurso) compartida

Problemas de la PC

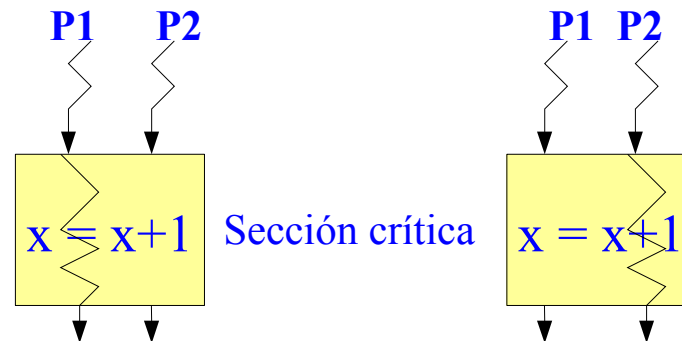
Exclusión mutua

- Sección crítica: porción de código con variables compartidas y que debe ejecutarse en exclusión mutua



Problema de la PC

Sección crítica: porción de código con variables compartidas y que debe ejecutarse en exclusión mutua



- **exclusión mutua** para acceder a la variable compartida x y asegurar que la variable va a quedar en estado consistente (Sincronización por competencia)

Sección crítica

- El código se divide en las siguientes secciones

SECCION DE ENTRADA

SECCION CRITICA

SECCION DE SALIDA

SECCION RESTANTE

- Existen partes de código con **variables compartidas** y que deben ejecutarse en **exclusión mutua (MUTEX)**



Sección crítica

- **Problema:** Garantizar que los procesos involucrados puedan operar sin generar ningún tipo de inconsistencia.

El segmento de código en el que un proceso puede modificar variables compartidas con otros procesos se denomina **sección crítica**

- **Sección de entrada**, se solicita el acceso a la sección crítica.
- **Sección crítica**, en la que se realiza la modificación efectiva de los datos compartidos.
- **Sección de salida**, en la que típicamente se hará explícita la salida de la sección crítica.
- **Sección restante**, que comprende el resto del código fuente.



Sección crítica



- Para entrar a la sección crítica **de manera segura** se debe cumplir:
- **Exclusión mutua**, si un proceso está en su sección crítica, entonces ningún otro proceso puede ejecutar su sección crítica.
- **Progreso**, todos los procesos que no estén en su sección de salida podrán participar en la decisión de quién es el siguiente en ejecutar su sección crítica.
- **Espera limitada**, todo proceso debería poder entrar en algún momento a la sección crítica



Exclusión mutua

Con la aplicación de mecanismos de exclusión mutua se **evita** que más de un **proceso** a la vez ingrese a las **secciones críticas**.

Se logra la sincronización entre los procesos que compiten por un recurso

Exclusión mutua

Las **zonas críticas** son secciones del programa que sólo debe ejecutar un *hilo* en un momento dado, o habría problemas

Protocolo

si está ocupado, espero (estado bloqueado)

si está libre:

Entro

cierro por dentro

actúo

salgo

si hay alguien esperando, que entre

si no, queda abierto/liberado

Mecanismos para Exclusión Mútua

Mecanismos

```
graph TD; A[Mecanismos] --> B[Semáforos]; A --> C[Métodos sincronizados]; A --> D[Cerrojos – lock explícito];
```

Semáforos

Métodos sincronizados

Cerrojos – lock explícito



Ejemplo contador

Clases:

Contador (proceso disparador),
ProcesoI (hilos), Dato,

```
public class Contador
{
    public static void main(...) {
        Dato unDato;
        ProcesoI p1;
        ProcesoI p2;
        //crea unDato y lo inicializa
        //crea hilos, los ejecuta y luego los finaliza
        // muestra el valor final de unDato
    }
}
```



Ejemplo contador

Classes:

Contador (proceso disparador),
ProcesoI (hilos), Dato,

```
public class Dato {  
    private int dato;  
    public Dato(int nro) {...}  
    public int getDato() {...}  
    public void incrementar()  
        {dato++; }  
}
```

```
public class Contador  
    public static void main(...) {  
        Dato unDato;  
        ProcesoI p1;  
        ProcesoI p2;  
        //crea unDato y lo inicializa  
        //crea hilos, los ejecuta y luego los finaliza  
        // muestra el valor final de unDato  
    }
```

Ejemplo contador

```
public class ProcesoI implements
Runnable{
    private Datos unD;
    //crea e inicializa unD

    public ProcesoI(Dato unD){..}

    public void run(){
        //incrementa 10000 veces }
}
```

Classes:

Contador (proceso disparador),
ProcesoI (hilos), Dato,

```
public class Dato {
    private int dato;
    public Dato(int nro){..}
    public int getDato(){..}
    public void incrementar()
        {dato++; }
}
```

```
public class Contador
{
    public static void main(...){
        Dato unDato;
        ProcesoI p1;
        ProcesoI p2;
        //crea unDato y lo inicializa
        //crea hilos, los ejecuta y luego los finaliza
        // muestra el valor final de unDato
    }
}
```



Ejemplo contador

```
public class ProcesoI implements
Runnable{
    private Datos unDato;

    public ProcesoI(Datos unD) {
        unDato = unD;
    }
    public void run() {
        for (int i=1; i<10000; i++){
            unDato.incrementar();
        }
    }
}
```

```
public class Datos {
    private int dato;

    public Datos(int nro) {
        dato = nro;
    }
    public int getDato() {
        return dato;
    }
    public void incrementar()
    {
        dato++;
    }
}
```



```
public class Contador {  
    public static void main(String[] args) {  
        Dato elContador = new Dato(0);  
        ProcesoI p1= new ProcesoI(elContador);  
        ProcesoI p2= new ProcesoI(elContador);  
  
        Thread h1= new Thread(p1);  
        Thread h2= new Thread(p2);  
  
        h1.start(); h2.start();  
        h1.join();  h2.join();  
        System.out.println("en main "+ elContador.getDato());  
    }  
}
```

Try {
 ...
} catch (...) {...}



Código ejecutado varias veces

Resultados diferentes, cercanos al 20000 pero no justo el 20000,

Explicación:

- ambos hilos ejecutan el método *run()*
- la acción de incrementar ($++$ valor) **no es atómica**
- puede pasar que cuando le toque el turno de ejecución a un hilo, el otro hilo sea interrumpido **justo después de haber recuperado el valor**
- pero antes de modificarlo
- entonces se **pierden incrementos**


```
public class Contador {  
    public static void main(String[] args){  
        Dato elContador = new Dato(0);  
        ProcesoI p1= new ProcesoI(elContador);  
        ProcesoI p2= new ProcesoI(elContador);  
  
        Thread h1= new Thread(p1);  
        Thread h2= new Thread(p2);  
  
        h1.start(); h2.start();  
        h1.join();  h2.join();  
        System.out.println("en main "+ elContador.getDato());  
    }  
}
```

excepcion

```
ProcesoI pp = new ProcesoI (elContador)
```

```
Thread h1= new Thread(pp)  
Thread h2 = new Thread(pp)
```



Cómo resolver el problema?

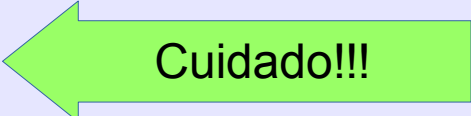
Sincronizar el acceso al recurso (la variable compartida) para lograr la exclusión mutua

```
public class Datos {  
    private int dato;  
  
    public Datos(int nro) {  
        dato = nro;  
    }  
  
    public int synchronized getDato() {  
        return dato;  
    }  
  
    public void synchronized incrementar() {  
        dato++;  
    }  
  
    public void synchronized set(int valor) {  
        Dato= valor;  
    }  
}
```

Métodos sincronizados

- Sincroniza todo el método

Utiliza el objeto de la clase que posee el método para sincronizar

```
public synchronized void incrementar  
throws InterruptedException {  
  
    dato++;  
    Thread.sleep( ...)   
}
```

Cada objeto en Java tiene un “lock” o llave implícito, disponible para lograr la sincronización