



Departamento de Programación
Facultad de Informática
Universidad Nacional del Comahue



Programación Concurrente



Sincronización I: competencia

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

Clases:

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 }
    }
}
```

Clases:
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++;
        // dato = dato + 1
    }
}
```

Ejemplo contador

```
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 (dato++) **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**

Problemas en lenguajes de alto nivel

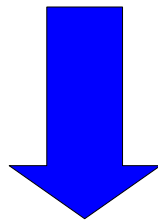
Generalmente existen partes de código con **variables compartidas** y que deben ejecutarse en **exclusión mutua**,

es decir tomar las operaciones que actúan sobre la variable compartida como **atómicas**.

...

datos.incrementar();

...



Sección crítica

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

Problemas de la PC: otro ejemplo

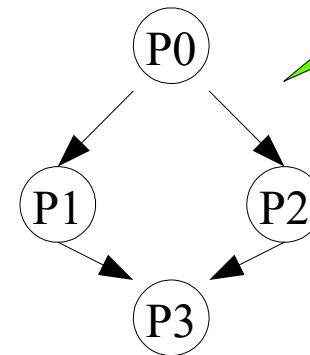
- Es más difícil analizar y verificar un algoritmo concurrente por el no determinismo.
- Que existan varias posibilidades de salida NO significa necesariamente que un programa concurrente sea incorrecto

```
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')
```



Grafo de
precedencia

¿Cual es la salida?

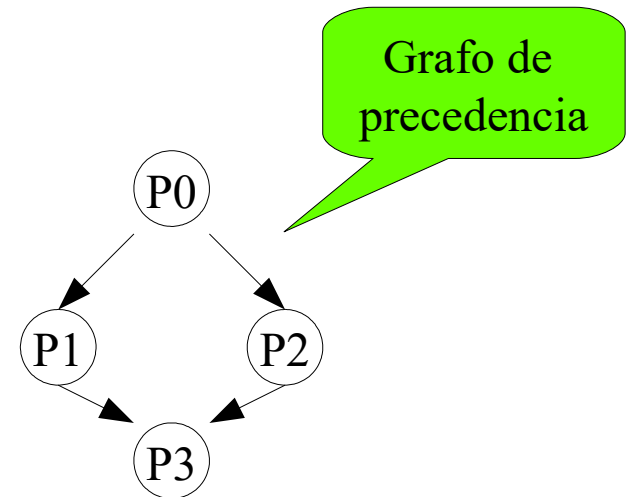
Representación gráfica

En el grafo de precedencia:

- un nodo por cada proceso/hilo
- el arco indica una relación de precedencia en la ejecución de los hilos.

En este caso:

- P0 debe terminar su ejecución para que P1 y P2 puedan comenzar a ejecutarse
- P1 y P2 pueden ejecutarse concurrentemente, tal vez con restricciones
- P3 debe esperar a que P1 y P2 terminen para poder ejecutarse



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), *escribir*(x) \longrightarrow 110,
escribir "fin"

- Se ejecuta P0, P2, P1, P3

$x=100$, $x > 100$ (no), *escribir* (x-50) \longrightarrow 50
escribir "fin"

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

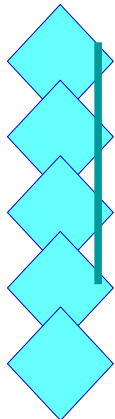
$x=100$, $x > 100$ (no), $x = x+10$ (110),
escribir (x-50) \longrightarrow 60,
escribir "fin"

INDETERMINISMO

Lo implementamos 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

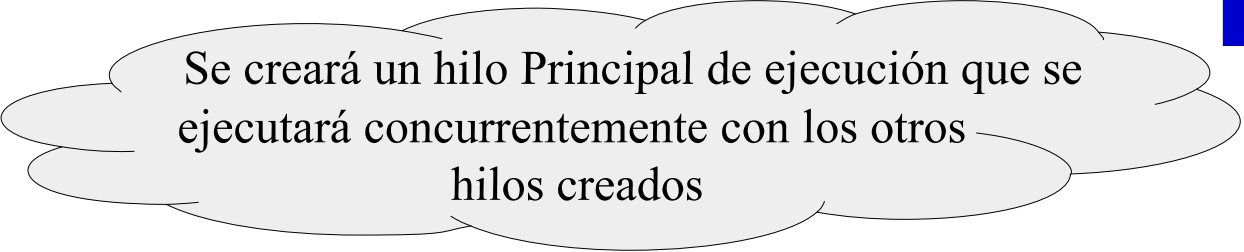


```
package hilos;

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, con runnable 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**



En el ejemplo anterior hay 2 resultados correctos 110 y 50.
El resultado 60 es incorrecto ¿¿????

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

Sección crítica

- El código se divide en las siguientes secciones

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

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.
 - **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 la **Exclusión mutua**, si un proceso está en su sección crítica, entonces ningún otro proceso puede ejecutar su sección crítica.

Y además ... es importante

- **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

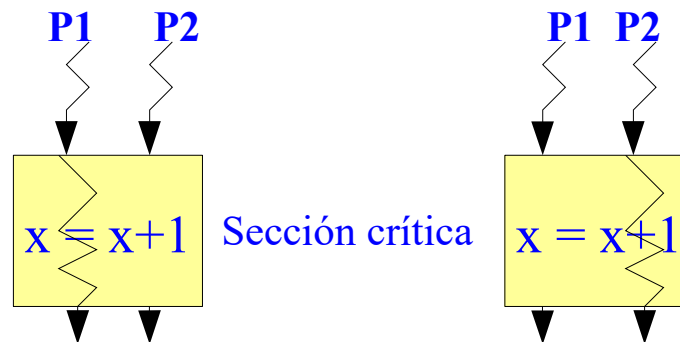
Con la aplicación de mecanismos de exclusión mutua se **evita** que más de un **proceso** a la vez ingrese a las **secciones críticas**.

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

SINCRONIZACION POR COMPETENCIA

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**)

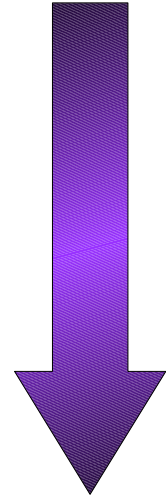


Continuamos con ... problemas de PC

- Trabajamos la concurrencia en un escenario de memoria compartida.
 - Los hilos tienen, en general, que consultar y actualizar variables compartidas.
 - Las acciones entre las tareas involucradas pueden entrelazarse en cualquier orden, y al trabajar sobre las mismas variables pueden producir inconsistencia ----> *condiciones de carrera* (es decir cual es el hilo que gana la carrera en el acceso al dato compartido)
 - Es necesario trabajar con “bloqueos” para proporcionar “exclusión mutua” en el acceso al área compartida, a la que llamamos “sección crítica”
 - El lenguaje de programación provee la forma de producir esos bloqueos, para permitir que un hilo tome el control del área compartida, y pueda tener la exclusividad de trabajo mientras dure el bloqueo.
 - Otros hilos no podrán leer ni escribir sobre ese área compartida hasta que se libere el bloqueo.
- 

Propiedades que deben cumplir los PC

- Seguridad
 - recursos compartidos Thread Safe
 - no producir inconsistencias
- Viveza
 - todos los hilos deben poder progresar en sus acciones



Sincronización y comunicación entre los hilos