



Departamento de Programación
Facultad de Informática
Universidad Nacional del Comahue



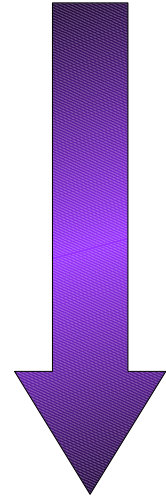
Programación Concurrente



Sincronización I: competencia – Exclusion mutua

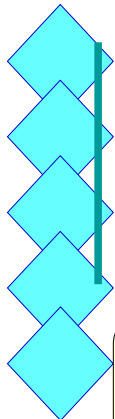
Propiedades que deben cumplir los PC

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



Sincronización y comunicación entre los hilos

Mecanismos para Exclusión Mútua



Mecanismos

Métodos y bloques sincronizados
lock implícito

Semáforos

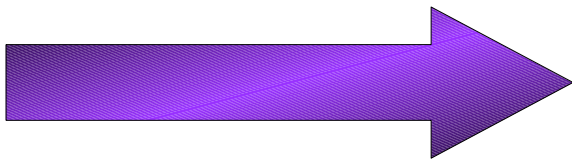
Cerrojos – lock explícito

Cómo resolver el problema?



Sincronizar el acceso al recurso

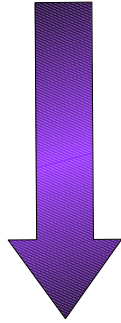
(la variable compartida)
para lograr la exclusión mutua



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

Cómo sincronizar?

Usar la palabra **synchronized**



La parte sincronizada
puede ser ejecutada por
un sólo hilo por vez



el hilo necesita
la llave para acceder
al código sincronizado.

Cómo sincronizar?





Exclusion mutua en Java - Synchronized

- Java utiliza **synchronized** para lograr la exclusion mutua sobre los objetos.
- Cada objeto tiene su propio lock.
- Existen 2 posibilidades para sincronizar objetos
 - El bloque sincronizados (synchronized)
 - **Los métodos sincronizados (synchronized)**
- Cada vez que un hilo intenta ejecutar un método sincronizado sobre un objeto lo puede hacer sólo si no hay algún otro hilo ejecutando un método sincronizado sobre el mismo objeto



Exclusion mutua en Java - Synchronized

- Al intentar ejecutar un metodo sincronizado sobre un objeto se obtiene el lock implícitamente. Al terminar la ejecución del método el lock es liberado implícitamente también.
- Un **hilo** que intenta ejecutar un método sincronizado sobre un objeto cuyo lock ya está en poder de otro hilo **es suspendido** y puesto en espera hasta que el lock del objeto es liberado.
- El lock se libera cuando el hilo que lo tiene tomado:
termina la ejecución del método / ejecuta un return / lanza una excepción.



Exclusión mutua en java – Synchronized

- El mecanismo de sincronización funciona si TODOS los accesos a los *datos delicados* ocurren dentro de métodos sincronizados, es decir con exclusión mutua
- Los datos delicados protegidos por métodos sincronizados deben ser privados

Cómo resolver el problema?

P3 en listos

P1

tiene la CPU
...
miP.metodoUno
(obtiene el lock)

pierde la CPU, pero
mantiene el lock
(espera en *listos*)

recupera la CPU

termina y libera el lock

pierde la CPU

```
public class Prueba {  
    private ...;  
    ...
```

```
    public synchronized void  
        metodoUno ()  
    { ...  
      ...  
      ...  
    }
```

```
    public synchronized void  
        metodoDos (int valor)  
    { ...  
      ...  
    }  
}
```

espera la CPU, esta en *listos*
...

obtiene la CPU, **miP.metodoDos**
(no puede obtener el lock, lo tiene P1)
(espera en *bloqueados*, libera la CPU)

se desbloquea y pasa a *listos*

obtiene la CPU y el lock
y ejecuta metodoDos

libera el lock

...

P2

Cómo resolver el problema?

P3 en listos

P1

tiene la CPU
...
miP1.metodoUno
(obtiene el lock)

pierde la CPU, pero
mantiene el lock
(espera en *listos*)

recupera la CPU

termina y libera el lock

pierde la CPU

```
public class Prueba {  
    private ...;  
    ...
```

```
    public synchronized void  
        metodoUno ()  
    { ...  
      ...  
      ...  
    }
```

```
    public synchronized void  
        metodoDos (int valor)  
    { ...  
      ...  
    }  
}
```

espera la CPU, esta en *listos*
...

obtiene la CPU, miP2.metodoDos
(no puede obtener el lock, lo tiene P1)
(espera en *bloqueados*, libera la CPU)

se desbloquea y pasa a *listos*

obtiene la CPU y el lock
y ejecuta metodoDos

libera el lock

...

P2



Exclusion mutua en Java -Synchronized

- Cada instancia de **Object** y sus subclases posee bandera de bloqueo (**lock implícito**)
- Los tipos primitivos (no objetos) solo pueden bloquearse a través de los objetos que los encierran
- No pueden sincronizarse variables individuales
- Los objetos arreglos cuyos elementos son tipos primitivos pueden bloquearse, pero sus elementos NO

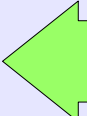
Sicronización – Exclusion mutua

```
public class Datos {  
    private int dato;  
  
    public Datos(int nro){  
        dato = nro;  
    }  
    public synchronized int getDato(){  
        return dato;  
    }  
    public synchronized void incrementar(){  
        dato ++;  
    }  
    public synchronized void 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( ...)  Cuidado!!!  
}
```

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

Sicronización – Exclusion mutua

```
public class Datos {  
    private int dato;  
  
    public Datos(int nro){  
        dato = nro;  
    }  
  
    public synchronized int getDato(){  
        return dato;  
    }  
  
    public synchronized void incrementar(){  
        this.set(this.getDato() + 1);  
    }  
  
    public synchronized void set(int valor){  
        dato= valor;  
    }  
}
```

Qué sucede si
cambiamos el código
del método
incrementar como se
muestra ...?

```
temp = this.getDato();  
this.set (temp + 1);
```

Sicronización – Exclusion mutua

```
public class Datos {  
    private int dato;  
  
    public Datos(int nro){  
        dato = nro;  
    }  
    public synchronized int getDato(){  
        return dato;  
    }  
    Public synchronized void incrementar(){  
        this.set(this.getDato() + 1);  
    }  
    public synchronized void set(int valor) throws InterruptedException {  
        dato= valor;  
        Thread.sleep(50000);  
    }  
}
```



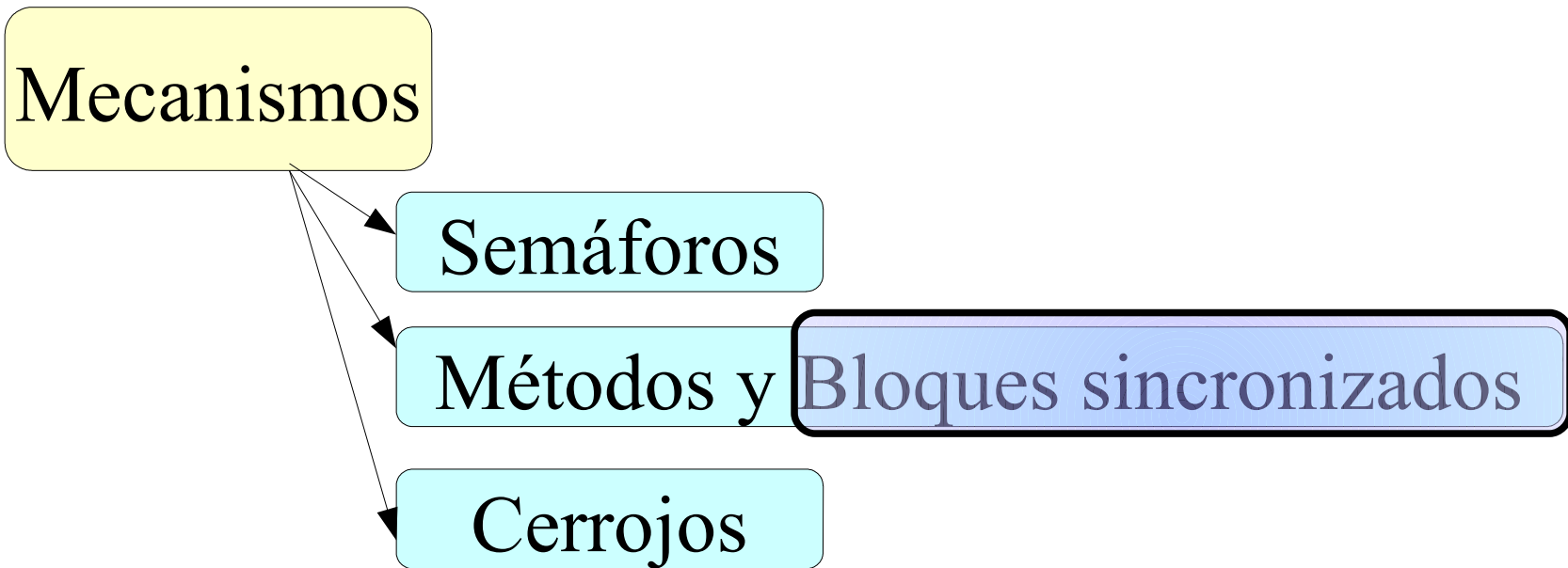
Sicronización – Exclusion mutua

```
public class Datos {  
    private int dato;  
  
    public Datos(int nro){  
        dato = nro;  
    }  
    public synchronized int getDato(){  
        return dato;  
    }  
    public synchronized void incrementar(){  
        this.set(this.getDato() + 1);  
    }  
    public synchronized void set(int valor) throws InterruptedException {  
        dato = valor;  
        Thread.sleep(50000);  
    }  
}
```

InterruptedException es una excepcion de tipo “chequeada”, entonces hay que capturarla (con un try/catch) o definirla (con un throws...)

Cuidado!!!

Exclusión mútua



Bloques sincronizados

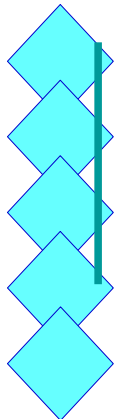
- El objeto es el que se utiliza como llave para acceder a la sección crítica

```
synchronized (objeto) {  
    // zona de exclusión mutua  
}
```

el objeto es sobre el que
se debe
sincronizar

```
synchronized (cc) {  
    int valor = cc.getN(id);  
    valor++;  
    cc.setN(id, valor);  
}
```

Synchronized en Java



```
public synchronized void metodo() {  
    // codigo del metodo aca  
}
```



```
public void metodo() {  
    synchronized(this) {  
        // codigo del metodo aca  
    }  
}
```

```
public static synchronized void metodo() {  
    // codigo del metodo aca  
}
```



```
public static void metodo() {  
    synchronized(MiClase.class) {  
        // codigo del metodo aca  
    }  
}
```

Bloque sincronizado

- ¿Cuándo sincronizamos un bloque?
 - Cuando el método tiene parte de su código como sección crítica
 - Cuando hay 2 o mas secciones críticas en el método, separadas por secciones NO criticas, para evitar bloqueos innecesarios.

```
public .... metodo (..) {  
    ...  
    ...  
    synchronized(this.) {  
        ...  
    }  
    ...  
    ...  
    synchronized(this) {  
        ...  
    }  
    ...  
    ...  
}
```

Métodos y bloques sincronizados

- La posesión de un lock es único por hilo.

```
public synchronized void metodoUno() {  
    // código del método aca  
}
```

```
public synchronized void metodoDos() {  
    ...  
    this.metodoUno()  
    ...  
}
```

hilo 1 ejecuta:

```
...  
miDato.metodoDos()  
...
```

Al ejecutar `metodoDos`, *hilo 1* obtiene el lock de `miDato`.
Cuando desde `metodoDos` se invoca a `metodoUno()`,
el hilo NO necesita volver a obtener el lock, porque ya lo tiene
en su poder
(`metodoUno` y `metodoDos` están en la misma clase)

Metodos y bloques sincronizados

- CUIDADO !!!

```
Clase UNO  
public synchronized void metodoUno() {  
    // codigo del metodo aca  
}
```

```
hilo 1 ejecuta:  
...  
    miDato.metodoDos()  
...
```

```
Clase DOS  
public synchronized void metodoDos() {  
    UNO miVar  
    ...  
    miVar.metodoUno()  
    ...  
}
```

Al ejecutar metodoDos *hilo 1* obtiene el lock de *miDato*.
Cuando desde metodoDos se invoca a metodoUno(), el *hilo 1*, necesita obtener el lock de *miVar*, pero NO LIBERA el lock de *miDato* que ya tiene en su poder

Metodos y bloques sincronizados

- CUIDADO !!!

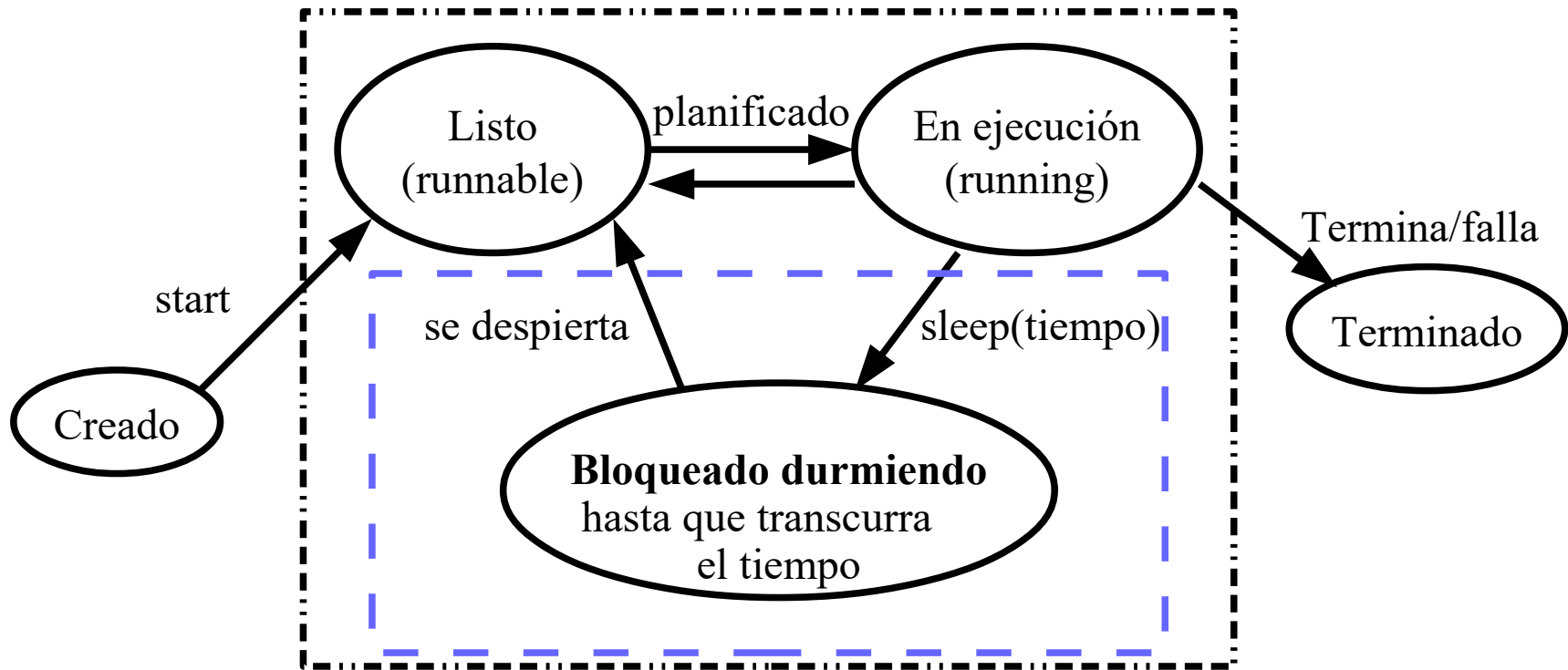
```
Clase UNO  
public synchronized void metodoUno() {  
    // codigo del metodo aca  
}
```

```
hilo 1 ejecuta:  
...  
    miDato.metodoDos(...)  
...
```

```
Clase DOS  
public synchronized void metodoDos(UNO miVar) {  
  
    ...  
    miVar.metodoUno()  
    ...  
}
```

Al ejecutar metodoDos *hilo 1* obtiene el lock de *miDato*.
Cuando desde metodoDos se invoca a metodoUno(), el *hilo 1*, necesita obtener el lock de *miVar*, pero NO LIBERA el lock de *miDato* que ya tiene en su poder

Recordemos ... estados de un hilo



Recordemos ... estados de un hilo

