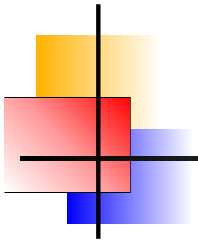


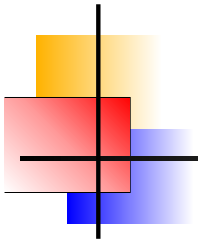
Concurrencia
Monitores

Guillermo Román Díez

groman@fi.upm.es

Universidad Politécnica de Madrid

Curso 2016-2017



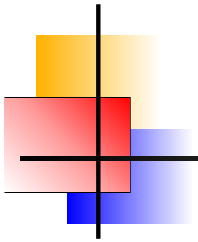
Recursos Compartidos

- La *especificación de recursos compartidos* nos permite:
  - Definir la interacción entre procesos de independientemente del lenguaje o técnica de programación
  - La comunicación entre procesos se realiza mediante las operaciones sobre el recurso compartido
  - La *exclusión mutua* se produce entre todas las operaciones del recurso
  - La *sincronización por condición* se define a través de las CPRE's de las operaciones del recurso compartido

Pregunta

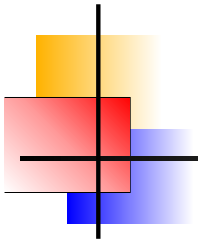
¿cómo implementamos un recurso compartido especificado?

- Métodos **synchronized** (no se verá en la asignatura)
- **Monitores**
- **Paso de Mensajes**



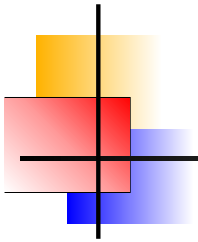
Monitores

- Son un mecanismo de **alto nivel** para programación concurrente
- Podríamos decir que un monitor *es una instancia de una clase que puede ser usada de forma segura por múltiples threads*
- Todos los **métodos** de un monitor deben ejecutarse en **exclusión mutua**
  - Habrá únicamente un proceso accediendo a memoria compartida
- Permite definir la **sincronización por condición** de cada uno de los métodos del monitor
  - Permite bloquear procesos hasta que se cumplan las condiciones para que puedan ejecutar
  - Permite notificar (señalizar) a un proceso bloqueado que puede continuar su ejecución cuando se cumplan las condiciones necesarias para ejecutar



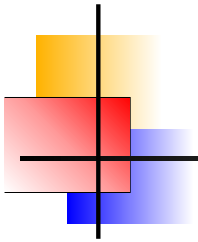
Ventajas del uso de Monitores

- Presenta ventajas con respecto a utilizar mecanismos de *bajo nivel* (espera activa, semáforos, . . .)
- Todos los accesos a memoria compartida se encuentran dentro de una misma clase
- No depende del número de procesos que accedan
- El "cliente" del monitor únicamente necesita conocer el interfaz del recurso
  - Los atributos (memoria compartida) del monitor únicamente serán accesibles desde dentro del monitor
  - La sincronización por condición también la realiza el monitor
  - Garantizar la exclusión mutua y la sincronización por condición ya no depende del thread que accede a la memoria compartida
- Permiten un desarrollo más "sistemático" que no depende de tener una "idea feliz" para solucionar el problema
- Se puede demostrar su corrección mucho más fácilmente que con otros mecanismos de bajo nivel



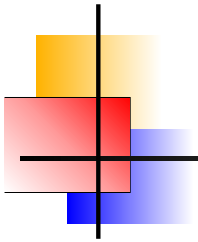
Monitores: Librería cclib

- Usaremos la clase `es.upm.babel.cclib.Monitor` que se encuentra en la librería `cclib.jar`
- En Java hay otros mecanismos para implementar recursos compartidos
  - Métodos `synchronized` + `notifyAll()`
  - Las clases `Lock`, `Condition` y `ReentrantLock`
- No entraremos en detalle de estas soluciones, pero las iremos comparando a lo largo del curso



Monitores: Mutex y SC

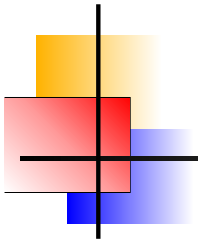
- El objetivo es implementar un **recurso compartido** implementádolo con un **monitor**
- Los monitores deben implementar las dos piezas clave:
  - **Exclusión Mutua**
    - Todos los métodos del recurso compartido deben ejecutarse en exclusión mutua
  - **Sincronización por Condición**
    - Los métodos del recurso compartido sólo deben ejecutarse cuando se cumpla su CPRE
    - Si no se cumple la CPRE el proceso debe esperar a que se cumpla
- Cada proceso es responsable de despertar a alguno de los procesos que estén esperando (si lo hay y se cumple su CPRE)



Monitores: Exclusión Mutua

- La clase que implemente un monitor debe tener un atributo de tipo `es.upm.babel.cclib.Monitor`

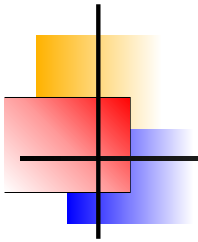
```
class Parking {
    private Monitor mutex;

    public Parking () {
        mutex = new Monitor ();
        ...
    }
    ...
}
```



Monitores: Exclusión Mutua

- Todos los métodos deben ejecutarse en exclusión mutua:
  - `mutex.enter()` al entrar
  - `mutex.leave()` al salir

```
class Parking {
   private Monitor mutex;

   public void entrar () {
     mutex.enter();
     ... // Ejecuta en Excl. Mutua
     mutex.leave();
   }

   public void salir () {
     mutex.enter();
     ... // Ejecuta en Excl. Mutua
     mutex.leave();
   }
}
```

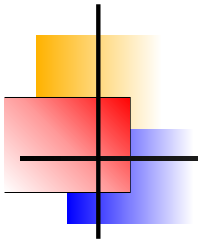


Monitores: Sincronización por condición

- Para la sincronización por condición usaremos la clase `es.upm.babel.cclib.Monitor.Cond`
- Modela una cola de espera de procesos (FIFO)
- Un *condition* se crea a través de un `Monitor`:

```
Cond cond = mutex.newCond();
```

- Dispone de los métodos:
  - `cond.await()`
    - Duerme un proceso en la cola de `cond` (wait-set)
    - Libera el *mutex* del monitor asociado a la condition
  - `cond.signal()`
    - Desbloquea un proceso de la cola de `cond`
  - `cond.waiting()`
    - Devuelve el número de procesos que haya bloqueados en `cond`



Monitores: Políticas Signal

- **Signal and Continue (SC)**: El proceso que señaliza mantiene el *mutex* y el proceso despertado debe competir por el *mutex* para ejecutar
- **Signal and Wait (SW)**: El proceso que señaliza es bloqueado y debe competir de nuevo por el *mutex* para continuar y el proceso despertado adquiere el *mutex* y cotinua su ejecución
- **Signal and Urgent Wait (SU)**: El proceso que señaliza es bloqueado pero será el primero en conseguir el *mutex* cuando lo libere el proceso despertado
- **Signal and Exit (SX)**: El proceso que señaliza sale del método y el proceso despertado coge directamente el *mutex* para ejecutar

NOTA!!

`cclib` usa SC y el proceso señalizado será el primero coger el *mutex* cuando acabe el proceso que señaliza

Monitores: Ejemplo Parking

```
class Parking {
    private Monitor mutex;          // Control mutex
    private Monitor.Cond cond;      // Cola espera
    private int nCoches;            // Memoria compartida

    public Parking () {
        mutex = new Monitor ();
        cond = mutex.newCond();
    }
public void entrar () {
    mutex.enter();
    if (nCoches>=CAP)
        cond.await();

    nCoches ++;

    if (nCoches < CAP &&
         cond.waiting>0)
        cond.signal();
    mutex.leave();
}
```

```
public void salir () {
    mutex.enter();
    // CPRE = cierto

    nCoches--;

    //nCoches < CAP
    if (cond.waiting>0)
        cond.signal();
    mutex.leave();
}
```

Monitores: De especificación a monitor

```
class Parking {
  private Monitor mutex;
  private Monitor.Cond cond;
  private int nCoches;  // Tipo

  public Parking () {
      mutex = new Monitor ();
      cond = mutex.newCond();
      nCoches = 0;       // Inicial
   }
  public void entrar () {
    mutex.enter();       // Ex. Mutua
    if (nCoches>=CAP)    // if (!CPRE)
      cond.await();

    nCoches ++;          // POST

    if (nCoches < CAP && cond.waiting>0)
      cond.signal();
    mutex.leave();       // Ex. Mutua
  }
  public void salir () {
    mutex.enter();       // Ex. Mutua
    // CPRE cierto

    nCoches--;           // POST

    // nCoches < CAP
    if (cond.waiting>0) // CPRE
      cond.signal();
    mutex.leave();       // Ex. Mutua
  }
}
```

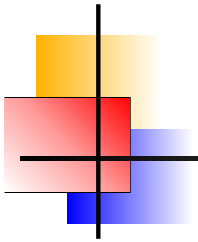
C-TAD: Parking

OPERACIONES:
ACCION: entrar
ACCION: salir

DOMINIO
TIPO: Parking $= \mathbb{N}$
DONDE: $CAP = \mathbb{N}$
INVARIANTE: $0 \leq self \leq CAP$
INICIAL: $\text{self} = 0$

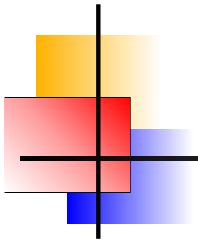
CPRE: $\text{self} < CAP$
entrar()
POST: $\text{self} = \text{self}^{pre} + 1$

CPRE: cierto
salir()
POST: $\text{self} = \text{self}^{pre} - 1$



Monitores: Conditions

- IMPORTANTE! Para lanzar un **signal** debemos garantizar que se **desbloquea un proceso** y que **cumple su CPRE**

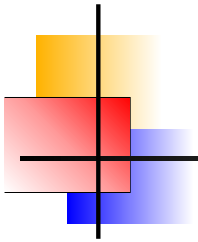


Monitores: Conditions

- IMPORTANTE! Para lanzar un **signal** debemos garantizar que se **desbloquea un proceso** y que **cumple su CPRE**

Pregunta

¿En el almacén de un dato podemos poner en la misma cola los productores y los consumidores?



Monitores: Conditions

- IMPORTANTE! Para lanzar un **signal** debemos garantizar que se **desbloquea un proceso** y que **cumple su CPRE**

Pregunta

¿En el almacén de un dato podemos poner en la misma cola los productores y los consumidores?

Solución

NO! Si productores y consumidores fueran bloqueados en la misma condition, al hacer el signal no sabríamos si se despierta un productor o un consumidor

- Cuando se completa el método `almacenar` sólo se puede despertar un proceso *consumidor* $\rightarrow$ una condition para consumidores
- Cuando se completa el método `extraer` sólo se puede despertar un proceso *productor* $\rightarrow$ una condition para productores

Monitores: Almacén de 1 Dato

```
class Almacen1Dato {
  private Monitor mutex;
  private Monitor.Cond cAlmacenar;
  private Monitor.Cond cExtraer;
  private Object almacenado;
  private boolean hayDato;

  public Almacen1Dato () {
      mutex = new Monitor ();
      cAlmacenar = mutex.newCond();
      cExtraer = mutex.newCond();
      hayDato = false;  // Inicial
   }

  public void almacenar (Object e) {
    mutex.enter();       // Ex. Mutua
    if (hayDato)        // if (!CPRE)
      cAlmacenar.await();

    almacenado = e;      // POST
    hayDato = true;      // POST

    if (cExtraer.waiting()>0) //hayDato
      cExtraer.signal();

    mutex.leave();       // Ex. Mutua
  }

  // [CONTINUA ...]
```

C-TAD: Almacen1Dato

OPERACIONES:
ACCION: almacenar:Tipo_Dato[e]
ACCION: extraer:Tipo_Dato[s]

DOMINIO
TIPO: $Almacen1Dato = (Dato{:}Tipo\_Dato \times HayDato : \mathbb{B})$

INVARIANTE: *cierto*
INICIAL: $\neg \text{self}.HayDato$

CPRE: $\neg \text{self}.HayDato$
almacenar(e)
POST: $\text{self}.Dato = e^{pre} \wedge \text{self}.HayDato$

CPRE: $\text{self}.HayDato$
extraer(s)
POST: $s = \text{self}^{pre}.Dato \wedge \neg \text{self}.HayDato$

Monitores: Almacén de 1 Dato

```
  ...

  public Object extraer () {
    mutex.enter();        // Ex. Mutua

    if (!hayDato)        // if (!CPRE)
      cExtraer.await();

    hayDato = false;     // POST
    Object dato = almacenado;

    if (cAlmacenar.waiting()>0) //!hayDato
      cAlmacenar.signal();

    mutex.leave();        // Ex. Mutua

    return dato;
  }
}
```

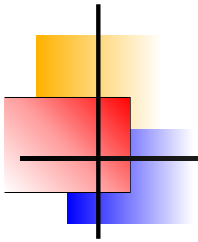
C-TAD: Almacen1Dato

OPERACIONES:
ACCION: almacenar:Tipo_Dato[e]
ACCION: extraer:Tipo_Dato[s]

DOMINIO
TIPO: $Almacen1Dato = (Dato : Tipo\_Dato \times HayDato : \mathbb{B})$
INVARIANTE: *cierto*
INICIAL: $\neg \text{self}.HayDato$

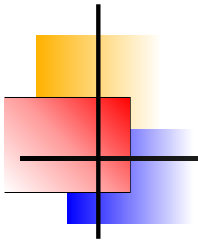
CPRE: $\neg \text{self}.HayDato$
almacenar(d)
POST: $\text{self}.Dato = e^{pre} \wedge \text{self}.HayDato$

CPRE: $\text{self}.HayDato$
extraer(s)
POST: $s = \text{self}^{pre}.Dato \wedge \neg \text{self}.HayDato$



Implementación de recursos con Monitores

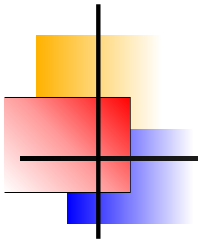
- A continuación vamos a ver como "sistematizar" la implementación de un recurso compartido con monitores
- Hay dos propiedades que deben cumplirse en la implementación de un recurso compartido:
  a) **Safety**: Un método sólo ejecuta si se cumple su CPRE
  b) **Progreso**: Al terminar de ejecutar un método, si se cumple la CPRE de algún proceso bloqueado, UNO debe ser despertado



Implementación de recursos con Monitores

- A continuación vamos a ver como "sistematizar" la implementación de un recurso compartido con monitores
- Hay dos propiedades que deben cumplirse en la implementación de un recurso compartido:
  a) **Safety**: Un método sólo ejecuta si se cumple su CPRE
  b) **Progreso**: Al terminar de ejecutar un método, si se cumple la CPRE de algún proceso bloqueado, UNO debe ser despertado

- A la hora de implementar un recurso compartido debemos contestas a las siguientes preguntas:
  Q1. ¿cuántos monitores necesito?
  Q2. ¿cómo garantizo la exclusión mutua?
  Q3. ¿cómo programo una CPRE?
  Q4. ¿cuántas conditions necesito?
  Q5. ¿cuándo hago un signal en una condition?



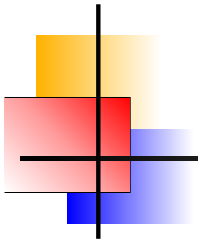
Implementación de recursos con Monitores

Q1. ¿cuántos monitores necesito?

- Normalmente, con un monitor es suficiente

Q2. ¿cómo garantizo la exclusión mutua?

```
public void metodo () {
  mutex.enter();
  ...
  mutex.leave();
}
```

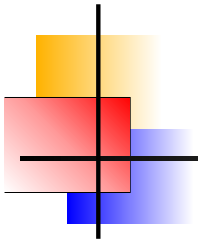


Implementación de recursos con Monitores

Q3. ¿cómo programo una CPRE?

```
public void metodo () {
   mutex.enter();
   if (!CPRE) {
      condition.await();
   }
   // Aqui se tiene que cumplir la CPRE!!
   ...
}
```

- La *condition* en la que bloquear un proceso depende de la respuesta de la pregunta Q4



Implementación de recursos con Monitores

Q4. ¿cuántas *conditions* necesito?

- Depende de cómo sean las CPRE's y del *arte* del programador
- A continuación veremos algunas ideas para identificar las *conditions* necesarias

Q5. ¿cuándo hago un signal en una condition?

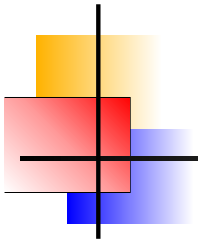
- Al terminar un método del recurso debemos *intentar* hacer UN signal (NUNCA más de uno)
- Para hacer un signal en una condition debemos comprobar que se cumple la CPRE de los procesos que estén bloqueados en dicha condition (fundamental la respuesta de Q4)
- La condition debe tener procesos bloqueados (`waiting()>0`)
  - Si hacemos el *signal* y la condition no tiene procesos bloqueados, *perdemos* el signal

Monitores: Estructura General

- La estructura *típica* del código de un monitor sería:

```
public class EjMonitor {

  // Decl. mutex                  // Q1
  private Monitor mutex;

  // Decl. conditions             // Q4
  private Monitor.Cond ...;

  // Declaracion del dominio
  private ...

  public EjMonitor () {
    mutex = new Monitor ();
    // Inic. conditions
    ... mutex.newCond();          //Q4
    // INICIAL monitor
   }
```

```
  public ... metodoi () {

    mutex.enter();   // Q2
    if (!CPRE) {     // Q3
      xxx.await();   // Q3,Q4
    }                // Q3

    // POST

    // SIGNALS!!     // Q5
    desbloquear();   // Q5

    mutex.leave();   // Q2

    return ...;
  }
}
```



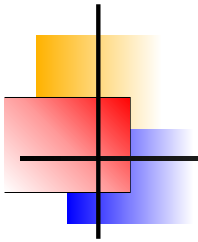
Monitores: ¿cuántas *conditions* necesito?

- Al intentar hacer un *signal* tenemos que evaluar las CPRE's de los procesos que se encuentran en la condition
- El número de *condititons* depende de las CPRE's de los métodos del recurso compartido
- Casos posibles:

1) **CPRE = cierto**

2) **CPRE $\neq$ cierto** y la **CPRE** sólo depende del **estado del recurso**

3) **CPRE $\neq$ cierto** y la **CPRE** depende de **parámetros de entrada** (y opcionalmente del estado del recurso)



Monitores: ¿cuántas *conditions* necesito?

1) *CPRE* = *cierto*

   Al ser *CPRE* = *cierto* no se bloquea ningún proceso y por tanto no es necesaria ninguna *condition*

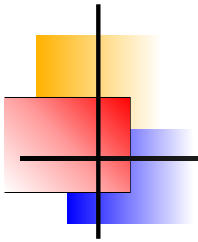
2) *CPRE* $\neq$ *cierto* y sólo depende del estado del recurso

   Con una *condition* por operación es suficiente (p.e. Almacén de un dato, Parking)

3) *CPRE* $\neq$ *cierto* y depende de parámetros de entrada (y opcionalmente del estado del recurso)

   El objetivo es poder evaluar la CPRE del proceso que se ha quedado bloqueado. Para poder hacer esto tenemos dos opciones:

   - *Indexación por parámetro*
   - *Indexación por cliente*



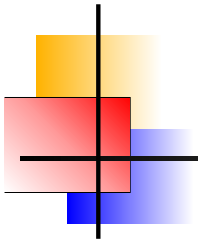
Monitores: Indexación por parámetro

- El objetivo es **poder evaluar la CPRE** de un proceso para hacer el signal correspondiente
- Para ello la indexación por parámetro **almacena en la misma condition** los procesos que hayan llamado **a la misma operación con los "mismos" valores de los parámetros**
- Tenemos dos posibilidades:
  - Se pueden encontrar **clases de equivalencia** para los valores de los parámetros. Entonces habría una condition por cada clase de equivalencia encontrada

Clase de Equivalencia

"Decimos que dos valores de entrada pertenecen a la misma clase de equivalencia cuando sus CPRE's son true para el mismo estado del recurso compartido "

  - Si no se encuentran clases de equivalencia, entonces tenemos una condition por cada posible valor de los parámetros



Monitores: Indexación por cliente

Pregunta

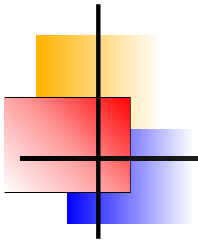
¿qué pasa si el conjunto de clases de equivalencia o de posibles valores de los parámetros de entrada es enorme (o infinito)?

- El número de *conditions* sería enorme (o infinito)

  Por ejemplo: Tenemos un conjunto S que contiene una serie de identificadores a bloquear (p.e. una lista negra)

  $$CPRE : id \in S$$
  $$operacion(id)$$
  $$POST : \ldots$$

  Tendríamos un número infinito de conditions (tantas como posibles *id* tengamos)
- SOLUCION: **Indexación por clientes**



Monitores: Indexación por cliente

- La indexación por clientes se basa en la idea de que cada "*cliente*" crea su propia *condition* para bloquearse
  - Una vez desbloqueado el cliente, esa condition es eliminada
- Los pasos en el **bloqueo** serían:
  1. Cuando no se cumple una CPRE, se crea una nueva condition
  2. La nueva condition, junto con la información para evaluar la CPRE, se almacena en una "colección" (lista, pila, cola, cola con prioridad, ...)
  3. Para esto (en Java) es necesario crear una clase que contenga la condition y almacene los valores de los parámetros
- Los pasos en el **desbloqueo** serían:
  1. Recorrer la colección de procesos bloqueados
  2. Para cada cliente, evaluar sus CPREs, desbloquear UNO de los clientes de los que se cumpla su CPRE