MONITORES

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

¿Cómo implementamos un recurso compartido especificado?
A través de métodos **synchronized**, **monitores** y/o **paso de mensajes**

Los **monitores** (`es.upm.babel.cclib.Monitor`) son los responsables de garantizar la ejecución en exclusión mutua de ciertos fragmentos de código (habrá únicamente un proceso accediendo a memoria compartida); mientras que las **conditions** (o *condition queues*) son las encargadas de manejar la sincronización por condición de cada uno de los métodos del monitor (permiten bloquear procesos hasta que se cumplan las condiciones para que puedan ejecutar - *CPREs* - y notificar a un proceso bloqueado que puede continuar su ejecución cuando se cumplan las condiciones necesarias para ejecutar).

Ventajas del uso de monitores:
(1) Todos los accesos a memoria compartida se encuentran dentro de una misma clase
(2) No depende del número de procesos que accedan
(3) El "cliente" del monitor únicamente necesita conocer el interfaz del usuario: los atributos (memoria compartida) del monitor únicamente serán accesibles desde dentro del monitor, la sincronización por condición también la realiza el monitor y garantizar la exclusión mutua y la sincronización por condición ya no depende del thread que accede a la memoria compartida
(4) Permiten un desarrollo más sistemático que no depende de tener una idea feliz para solucionar el problema
(5) Se puede demostrar su corrección mucho más fácilmente que con otros mecanismos de bajo nivel.

Los objetos de la clase monitor proporcionan un método para **solicitar permiso de ejecución** en exclusión mutua (`enter()`) y otro método para **liberar ese permiso** (`leave()`). Por ejemplo:

```
Monitor mutex = new Monitor();
mutex.enter();
<sección crítica>
mutex.leave();
```

A simple vista, un **monitor** se parece mucho a un *semáforo* inicializado a uno como los que usábamos para el protocolo de exclusión mutua, pero los monitores solo tienen dos estados: **libre** y **ocupado**. Cuando se crea, el monitor está libre. Al hacer `enter()` sobre un monitor libre pasa a estar ocupado. El proceso que hace `enter()` sobre un monitor ocupado se bloquea. Hacer `leave()` sobre un monitor ocupado lo libera si no hay procesos bloqueados en `enter()` – en caso contrario se desbloquea al más antiguo. No se puede hacer `leave()` sobre un monitor no adquirido previamente.

Ejemplo: contador compartido (garantizando exclusión mutua con Monitores):

```
class ContadorMon {
```

Sincronización por condición:

Con cada objeto de clase Monitor podemos asociar objetos (en número variable) de la clase Monitor.Cond que son *waitsets* con política FIFO. Estos son las *conditions queues*, que se construyen a partir de monitores usando el método `newCond()` de la clase **Monitor**; lo que garantiza la asociación de cada cola con un único monitor.

El bloqueo de hilos se realiza llamando al método `await()` de la clase **Monitor.Cond**. El proceso bloqueado se apunta al final de la cola interna de la *condition queue*. Para ejecutar `await()` es imprescindible haber adquirido el monitor asociado con la *condition* en cuestión, y el bloqueo conlleva la liberación automática de dicho monitor.

¿Cómo se desbloquean los hilos en una *condition queue*? Otro thread que haya ganado acceso exclusivo al monitor asociado puede ejecutar el método `signal()` de la clase **Monitor.Cond**. Esto saca al primer hilo de la cola y lo coloca al principio de la cola del monitor. En caso de no haber hilos bloqueados en una *condition* el `signal()` no tiene efecto alguno. Esta operación debe ser la última (excepto mutex.leave()) que se haga dentro de un método. Si la cola de la variable está vacía, equivale a una operación vacía.

Además, el método `waiting()` proporciona el número de procesos que hay bloqueados en una *condition queue*.

Ejemplo: semáforos (con Monitores):

Ejemplo: aparcamiento (añadimos sincronización por condición al contador compartido)

C-TAD: Parking
OPERACIONES:
ACCION: entrar
ACCION: salir

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

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

CPRE: cierto
salir()
POST: self = $self^{PRE} - 1$

```
public class Parking {
```

Almacén de un dato con Monitores:

¿En el almacén de un dato podemos poner en la misma cola los productores y los consumidores? No, ya que si ambos fueran bloqueados en la misma *condition (cond)*, 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 → *cond* para consumidores
Cuando se completa el método extraer sólo se puede despertar un proceso productor → *cond* para productores

C-TAD: Almacen1Dato
OPERACIONES:
ACCION: almacenar: Tipo_Dato[e]
ACCION: extraer: Tipo_Dato[s]

DOMINIO
TIPO: Almacen1Dato = (Dato: Tipo_Dato x HayDato: B)
INVARIANTE: cierto
INICIAL: ¬self.HayDato

CPRE: ¬self.HayDato
almacenar(e)
POST: self.Dato = e ∧ self.HayDato

CPRE: self.HayDato
extraer(s)
POST: s = $self^{pre}$.Dato ∧ ¬self.HayDato

¿Cómo sistematizamos la implementación de un recurso compartido con monitores?
Se deben cumplir dos propiedades: (1) **Safety** - un método SOLO ejecuta si se cumple su CPRE y (2) **Progreso** - al terminar de ejecutar un método, si se cumple la CPRE de algún proceso bloqueado, UNO debe ser desbloqueado.

Además, se debe contestar a las siguientes preguntas:

(1) ¿Cuántos monitores necesito?

Normalmente, con un monitor es suficiente

(2) ¿Cómo garantizo la exclusión mutua?

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

(3) ¿Cómo programo una CPRE?

```
public void metodo() {
  mutex.enter();
  if(!CPRE)
    condition.await();   // La condition sobre la que bloquear un proceso
                         // depende de la respuesta de la pregunta siguiente
  // Aquí se tiene que cumplir la CPRE!!
  …
  mutex.leave();
}
```

(4) ¿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 conditions depende de las CPRE's de los método del recurso compartido. Casos posibles:

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

(2) CPRE ≠ cierto y la CPRE sólo depende del estado del recurso compartido
Con una *condition* por operación es suficiente (p.e. Almacén de un dato, Parking)

(3) CPRE ≠ cierto y la CPRE depende de parámetros de entrada (y opcionalmente 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
Almacena en la misma *condition* los procesos que hayan llamado a la misma operación con los "mismos" valores de los parámetros. Hay dos posibilidades: (i) Encontrar clases de equivalencia para los valores de los parámetros, en cuyo caso habría una *condition* por cada clase de equivalencia (ii) Si no se encuentran clases de equivalencia, entonces tenemos una *condition* por cada posible valor de los parámetros Si el conjunto de clases de equivalencia es infinito, el número de *conditions* también lo sería (p.e. un conjunto S que contiene una serie de identificadores a bloquear, entonces tendríamos un número infinito de conditions - tantas como ids tengamos).	Para solucionar el problema anterior, nace la indexación por cliente, que se basa en la idea de que cada cliente crea su propia *condition* para bloquearse y una vez desbloqueada se elimina. Los pasos en el bloqueo serían: cuando no se cumple una CPRE, se crea una nueva *condition*; que junto con la información para evaluar la CPRE, se almacena en una colección (lista, pila, cola, cola con prioridad, …). 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: recorrer la colección de procesos bloqueados y, para cada cliente, evaluar sus CPREs, desbloquear UNO de los clientes de los que cumpla su CPRE.

(5) ¿Cuándo hago un `signal` en una *condition*?

Al terminar un método del recurso debemos intentar hacer UN `signal` (y 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* (depende de la respuesta de la pregunta anterior)
La *condition* debe tener procesos bloqueados (`waiting() > 0`). Si hacemos el `signal` y la condition no tiene procesos bloqueados, perdemos el `signal`

Estructura típica de un problema de Monitores:

```
public class EjMonitor {
  // Declaración mutex                    // Q1
  private Monitor mutex;

  // Declaración conditions               // Q4
  private Monitor.Cond …

  // Declaración del dominio
  private …

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

  public … metodoi() {
    mutex.enter();                  // Q2
    if(hayDato)                           // Q3
      xxx.await();                  // Q3, Q4

    // POST

    // SIGNALS                            // Q5
    desbloquear()                         // Q5
    mutex.leave();                  // Q2
    return …;
  }
}
```

EXÁMENES

1. Supóngase que una condición de sincronización (CPRE) de una operación Op de un recurso compartido depende del estado del recurso y de un parámetro de entrada (x) que puede tomar dos valores. Supóngase que dicho recurso va a ser implementado con monitores y que la operación va a ser llamada a lo sumo por un único proceso.
(a) No es posible implementar la sincronización condicional de Op con una única variable Cond.
(b) Es posible implementar la sincronización condicional de Op con dos variables Cond.

2. Supóngase el siguiente código:
```
Monitor m = new Monitor();
Cond c = m.newCond();
```
(a) Al ejecutar `c.wait()` se libera el monitor `m`.
(b) Es necesario ejecutar `m.leave()` inmediatamente después de ejecutar `c.await()` para liberar el monitor `m`.

3. Dado el siguiente CTAD:

ACCIÓN op1:
ACCIÓN op2:

TIPO: MiCTAD $= \mathbb{Z}$
INICIAL: $self = 0$

CPRE: $self <= 0$
op1()
POST: self $=$ self$^{pre} + 1$
CPRE: self $>= 0$
op2()
POST: self $= -1$

Se ha decidido implementarlo con monitores mediante el siguiente código:

```
public class MiCTAD{
  private Monitor mutex =
    new Monitor();
  private Monitor.Cond c1 =
    mutex.newCond();
  private Monitor.Cond c2 =
    mutex.newCond();
  private int self = 0;

  public void op1(){
    mutex.enter();
    if (self > 0) {c1.await();}
    self ++;
    c2.signal();
    mutex.leave();
}
```

```
  public void op2(){
    mutex.enter();
    if (self < 0) {c2.await();}
    self = -1;
    c1.signal();
    mutex.leave();
}
```

(a) Podrían ejecutarse operaciones cuya CPRE no se cumple.
(b) Podría darse el caso de que hubiese hilos esperando en c1 o en c2 que, pudiendo ejecutarse, no se desbloqueen.
(c) Se trata de una implementación correcta del recurso compartido

4. Dada esta otra implementación del CTAD del ejercicio anterior.

```
public class MiCTAD{
  private Monitor mutex =
    new Monitor();
  private Monitor.Cond c1 =
    mutex.newCond();
  private Monitor.Cond c2 =
    mutex.newCond();
  private int self = 0;

  public void op1(){
    mutex.enter();
    if (self > 0) {c1.await();}
    self ++;
    desbloquear();
    mutex.leave();
}
```

```
  public void op2(){
    mutex.enter();
    if (self < 0) {c2.await();}
    self = -1;
    desbloquear();
    mutex.leave();
}


private void desbloquear () {
  if (self <= 0) {
    c1.signal();
  }
  else if (self >= 0) {
    c2.signal();
  }
}
```

(a) Podrían ejecutarse procesos cuya CPRE no se cumple.
(b) Se trata de una implementación correcta del recurso compartido
(c) Podría darse el caso de que hubiese hilos esperando en c1 o en c2 que, pudiendo ejecutarse, no se desbloqueen.

5. Supóngase que una condición de sincronización (CPRE) de una operación Op de un recurso compartido depende del estado del recurso y de dos parámetros de entrada: x, que puede tomar 3 valores e y, que puede tomar 7 valores. Supóngase que dicho recurso va a ser implementado con monitores, siguiendo la metodología vista en clase, y que la operación va a ser invocada por un número indefinido (> 100) de procesos.
(a) Es posible implementar la sincronización condicional de Op con un solo objeto Cond.
(b) Es posible implementar la sincronización condicional de Op con 10 objetos Cond.
(c) Es posible implementar la sincronización condicional de Op con 21 objetos Cond.

6. La figura 1 muestra una implementación del conocido problema de lectores/escritores mediante monitores.
(a) La implementación del método finEscribir es correcta.
(b) La implementación del método finEscribir es incorrecta.

7. Seguimos con ese mismo código.
(a) La sentencia cLeer.signal(); en el método inicioLeer es necesaria.
(b) 2 La sentencia cLeer.signal(); en el método inicioLeer es innecesaria.

8. Con el mismo código, nos referimos ahora a la sentencia cLeer.signal(); del método finLeer.
(a) Es una sentencia correcta y necesaria.
(b) Es una sentencia correcta, pero innecesaria.
(c) Es una sentencia incorrecta, puede provocar que threads ejecuten inicioLeer indebidamente.

```
class LectoresEscritores1 {
    private int lectores;
    private int escritores;
    private Monitor mutex;
    private Monitor.Cond cLeer;
    private Monitor.Cond cEscribir;
    public LectoresEscritores1(){
        lectores = 0;
        escritores = 0;
        mutex = new Monitor();
        cLeer     = mutex.newCond();
        cEscribir = mutex.newCond();
    }
    public void inicioLeer() {
        mutex.enter();
        if (escritores > 0) {
            cLeer.await();
        }
        lectores++;
        cLeer.signal();
        mutex.leave();
    }
    public void finLeer() {
        mutex.enter();
        lectores--;
        if (lectores+escritores == 0 &&
            cEscribir.waiting()>0) {
            cEscribir.signal();
        } else {
            cLeer.signal();
        }
        mutex.leave();
    }

    public void inicioEscribir() {
        mutex.enter();
        if (lectores+escritores > 0) {
            cEscribir.await();
        }
        escritores++;
        mutex.leave();
    }
    public void finEscribir() {
        mutex.enter();
        escritores--;
        if (cLeer.waiting() > 0) {
            cLeer.signal();
        }
        if (cEscribir.waiting() > 0) {
            cEscribir.signal();
        }
        mutex.leave();
    }
}
```

TIPO: $LE = (l : \mathbb{Z} \times e : \mathbb{Z})$
INVARIANTE: $\forall r \in LE \bullet\ r.e \geq 0 \wedge r.l \geq 0 \wedge r.e \leq 1 \wedge$
$((r.e > 0 \Rightarrow r.l = 0) \wedge (r.l > 0 \Rightarrow r.e = 0))$
INICIAL: $\text{self} = (0, 0)$

CPRE: $\text{self}.e = 0$
inicioLeer()
POST: $\text{self} = (\text{self}^{pre}.l + 1, \text{self}^{pre}.e)$
CPRE: $\text{self}.e = 0 \wedge \text{self}.l = 0$
inicioEscribir()
POST: $\text{self} = (\text{self}^{pre}.l, \text{self}^{pre}.e + 1)$
CPRE: *cierto*
finLeer()
POST: $\text{self} = (\text{self}^{pre}.l - 1, \text{self}^{pre}.e)$
CPRE: *cierto*
finEscribir()
POST: $\text{self} = (\text{self}^{pre}.l, \text{self}^{pre}.e - 1)$

Figura 1: Lectores/Escritores con monitores (especificación a la derecha).

9. A continuación mostramos la especificación formal de un recurso para jugar al popular juego de mesa *Los nómadas que cantan*. Los desarrolladores han ideado un recurso compartido para representar las materias primas que tiene el jugador, siendo estas materias primes cereal, agua y madera. Se pide: Completar la implementación de este recurso mediante monitores siguiendo la metodología vista en clase. En cuanto al código de desbloqueos os recomendamos implementar en el método `desbloqueoSimple()` un código genérico para todas las operaciones.

C-TAD MateriasPrimas
OPERACIONES
ACCIÓN cargarCereal:
ACCIÓN cargarAgua:
ACCIÓN cargarMadera:
ACCIÓN avanzar:
ACCIÓN reparar:
SEMÁNTICA
DOMINIO:
TIPO: $MateriasPrimas = (cereal : \mathbb{N} \times agua : \mathbb{N} \times madera : \mathbb{N})$
INICIAL: $\text{self} = (0, 0, 0)$
INVARIANTE: $\text{self}.cereal + \text{self}.agua + \text{self}.madera < 10$

CPRE: $\text{self}.cereal + \text{self}.agua + \text{self}.madera + 1 < 10$
cargarCereal
POST: $\text{self}^{pre} = (c, a, m) \land \text{self} = (c + 1, a, m)$

CPRE: $\text{self}.cereal + \text{self}.agua + \text{self}.madera + 1 < 10$
cargarAgua
POST: $\text{self}^{pre} = (c, a, m) \land \text{self} = (c, a + 1, m)$

CPRE: $\text{self}.cereal + \text{self}.agua + \text{self}.madera + 1 < 10$
cargarMadera
POST: $\text{self}^{pre} = (c, a, m) \land \text{self} = (c, a, m + 1)$

CPRE: $\text{self}.cereal > 0 \land \text{self}.agua > 0$
avanzar
POST: $\text{self}^{pre} = (c, a, m) \land \text{self} = (c - 1, a - 1, m)$

CPRE: $\text{self}.agua > 0 \land \text{self}.madera > 0$
reparar
POST: $\text{self}^{pre} = (c, a, m) \land \text{self} = (c, a - 1, m - 1)$

```
class MateriasPrimas {
  // Estado del recurso


  // Monitores y colas conditions



  public void MateriasPrimas() {



  }

  public void cargarCereal() {
    // acceso a la sección crítica y código de bloqueo


    // codigo de la operacion


    // codigo de desbloqueo y salida de la seccion critica

  }

  public void cargarAgua() {
    // acceso a la sección crítica y código de bloqueo


    // codigo de la operacion


    // codigo de desbloqueo y salida de la seccion critica

  }

  public void cargarMadera() {
    // acceso a la sección crítica y código de bloqueo


    // codigo de la operacion


    // codigo de desbloqueo y salida de la seccion critica

  }
```

```
  public void avanzar() {
    // acceso a la sección crítica y código de bloqueo


    // codigo de la operacion


    // codigo de desbloqueo y salida de la seccion critica

  }

  public void reparar() {
    // acceso a la sección crítica y código de bloqueo


    // codigo de la operacion


    // codigo de desbloqueo y salida de la seccion critica

  }

  private void desbloqueoSimple() {









  }
}
```

10. Dada la siguiente especificación formal de un recurso compartido *Peligro*. Se pide: Completar la implementación de este recurso mediante monitores:

C-TAD *Peligro*
OPERACIONES
ACCIÓN *avisarPeligro*: $\mathbb{B}[e]$
ACCIÓN *entrar*:
ACCIÓN *salir*:

SEMÁNTICA
DOMINIO:
TIPO: $Peligro = (p : \mathbb{B} \times o : \mathbb{N})$
INICIAL: $\text{self} = (false, 0)$

INVARIANTE: $\text{self}.o \leq 5$

CPRE: Cierto
***avisarPeligro*(x)**
POST: $\text{self}.p = x \wedge \text{self}.o = \text{self}^{pre}.o$

CPRE: $\neg\text{self}.p \wedge \text{self}.o < 5$
***entrar*()**
POST: $\neg\text{self}.p \wedge \text{self}.o = \text{self}^{pre}.o + 1$

CPRE: $\text{self}.o > 0$
***salir*()**
POST: $\text{self}.p = \text{self}^{pre}.p \wedge \text{self}.o = \text{self}^{pre}.o - 1$

```
class Peligro {
  // Estado del recurso (inicialización incluída)


  // Monitores y conditions (inicialización incluída)



  public void Peligro() {  }

  public void avisarPeligro(boolean x) {







  }

  public void entrar() {







  }

  public void salir() {







  }

  private void desbloquear() {







  }
}
```

11. A continuación mostramos la especificación formal de un recurso gestor *Misil*. Se pide: Completar la implementación de este recurso mediante monitores. En cuanto al código de desbloqueos podéis optar tanto por un método de desbloqueo genérico como por tener código de desbloqueo especializado en los distintos métodos. Si optáis por la segunda opción dejan en blanco el cuerpo del método desbloqueo. NOTA: Podéis usar el método Math.abs(x) para calcular el valor absoluto de un número, |x|:

C-TAD Misil
OPERACIONES
ACCIÓN notificar: $\mathbb{Z}[e]$
ACCIÓN detectarDesviacion: $TUmbral[e] \times \mathbb{Z}[s]$

SEMÁNTICA
DOMINIO:
TIPO: $TUmbral = [0..100]$
TIPO: $Misil = \mathbb{Z}$

INICIAL: self $= 0$

CPRE: *Cierto*
notificar(desv)
POST: self $= desv$

CPRE: $|$ self $|> umbral$
detectarDesviacion(umbral,d)
POST: $self = \text{self}^{pre} \wedge d = \text{self}^{pre}$

```
class Misil {
  // Estado del recurso


  // Monitores y colas conditions



  public void Misil() {



  }

  public void notificar(int desv) {
    // acceso a la sección crítica y código de bloqueo


    // codigo de la operacion


    // codigo de desbloqueo y salida de la seccion critica

  }

  public void detectarDesviacion(int umbral) {
    // acceso a la sección crítica y código de bloqueo


    // codigo de la operacion


    // codigo de desbloqueo y salida de la seccion critica

  }

  private void desbloqueo() {







  }
}
```