



## Actividades UD 3.- Primitivas de sincronización



Concurrencia y Sistemas Distribuidos



## Guía de Actividades

---

- ▶ Características generales de monitores: Actividades 1 y 2
  - ▶ Monitores básicos en Java: Actividades 3, 4, 5 y 6
  - ▶ Variantes monitor: Act. 7, 8, 9, 10, 11 y 12
  - ▶ Invocaciones cruzadas: Act. 13
-



# ACTIVIDAD 1

Eventos	CPU	ACTIVOS fuera del monitor X	ACTIVOS dentro del monitor X	COLA del monitor X	Suspendidos
Inicialmente	T4	---	T4	---	T1,T2,T3
T1 se activa e invoca un método del monitor X					
T2 se activa y ejecuta código que está fuera del monitor					
El proceso en ejecución invoca una operación del monitor X					
T3 se activa y ejecuta código que está fuera del monitor					



# ACTIVIDAD 1 - Solución

Eventos	CPU	ACTIVOS fuera del monitor X	ACTIVOS dentro del monitor X	COLA del monitor X	Suspendidos
Inicialmente	T4	---	T4	---	T1,T2,T3
T1 se activa e invoca un método del monitor X	T4		T4	T1	T2,T3
T2 se activa y ejecuta código que está fuera del monitor	T2	T2	T4	T1	T3
El proceso en ejecución invoca una operación del monitor X	T4		T4	T1,T2	T3
T3 se activa y ejecuta código que está fuera del monitor	T3	T3	T4	T1,T2	



## Actividad 2

---

- |  |  |
|--|--|
| 1. Un monitor es un mecanismo de sincronización de alto nivel integrado en algunos lenguajes de programación concurrentes.   |  |
| 2. El concepto de monitor evita la necesidad de que distintos hilos compartan memoria.   |  |
| 3. Un monitor es una clase que además resuelve exclusión mutua y sincronización condicional.   |  |
| 4. Un monitor dispone siempre de una cola de entrada donde esperan aquellos hilos que desean utilizar el monitor cuando lo está utilizando otro hilo.                                      |  |
| 5. Dentro de un monitor se pueden producir condiciones de carrera. Cuando esto sucede, el hilo que ejecuta código dentro del monitor debe ejecutar <code>c.wait()</code> para abandonarlo. |  |



## Actividad 2 - Solución

1. Un monitor es un mecanismo de sincronización de alto nivel integrado en algunos lenguajes de programación concurrentes.	V
2. El concepto de monitor evita la necesidad de que distintos hilos compartan memoria.	F
3. Un monitor es una clase que además resuelve exclusión mutua y sincronización condicional.	V
4. Un monitor dispone siempre de una cola de entrada donde esperan aquellos hilos que desean utilizar el monitor cuando lo está utilizando otro hilo.	V
5. Dentro de un monitor se pueden producir condiciones de carrera. Cuando esto sucede, el hilo que ejecuta código dentro del monitor debe ejecutar <code>c.wait()</code> para abandonarlo.	F



## ACTIVIDAD 3

---

- ▶ A continuación se muestra la última de las soluciones propuestas en la actividad del problema del *productor-Consumidor* visto en la Unidad 2.
- ▶ Modifíquela apropiadamente, utilizando la sintaxis de Java y el modelo de monitores Java, para que la clase resultante sea un monitor correcto.

```

public class Consumidor extends Thread
{
    private Caja caja;
    private int cnumber;
    public Consumidor(Caja c, int number)
    {
        caja = c;
        cnumber = number;
    }
    public void run()
    {
        int value = 0;
        for (int i = 1; i < 11; i++)
        {
            value = caja.obtener();
            System.out.println("Consumidor #" + cnumber+ " obtiene: " + value);
            try
            {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}

```

```

public class Productor extends Thread
{
    private Caja caja;
    private int prodnumber;
    public Productor(Caja c, int number)
    {
        caja = c;
        prodnumber = number;
    }
    public void run()
    {
        for (int i = 1; i < 11; i++)
        {
            caja.poner(i);
            System.out.println("Productor #" + prodnumber+ " pone: " + i);
            try
            {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}

```

```

public class CondicionDeCarrera
{
    public static void main(String[] args)
    {
        Caja c = new Caja();
        Consumidor c1 = new Consumidor(c, 1);
        Productor p1 = new Productor(c, 1);

        c1.start();
        p1.start();

    }
}

```

```

public class Caja
{
    private int contenido = 0;
    private boolean llena = false;

    public synchronized int obtener()
    {
        while (!llena) Thread.yield();
        int valor = contenido;
        contenido = 0;
        llena = false;
        return valor;
    }

    public synchronized void poner(int valor)
    {
        while (llena) Thread.yield();
        llena = true;
        contenido = valor;
    }
}

```



## ACTIVIDAD 3 - Solución

```
public class Caja
{
    private int contenido= 0;
    private boolean llena = false;

    public synchronized int obtener()
    {
        if (!llena) try {wait();
            }catch (InterruptedException e) {};
        int valor = contenido;
        contenido = 0;
        llena = false;
        notify();
        return valor;
    }

    public synchronized void poner(int valor)
    {

        if (llena) try {wait();
            }catch (InterruptedException e) {};
        llena = true;
        contenido = valor;
        notify();
    }
}
```

```
public class Caja
{
    private int contenido= 0;
    private boolean llena = false;

    public synchronized int obtener()
    {

        while (!llena) try {
            wait();
            }catch (InterruptedException e) {};
        int valor = contenido;
        contenido = 0;
        llena = false;
        notifyAll();
        return valor;
    }

    public synchronized void poner(int valor)
    {

        while (llena) try {
            wait();
            }catch (InterruptedException e) {};
        llena = true;
        contenido = valor;
        notifyAll();
    }
}
```



# ACTIVIDAD 4

```
public class BoundedBuffer {  
    private int first, last;  
    private int numItems, capacity;  
    private long items[];  
    public BoundedBuffer(int size){  
        capacity = size;  
        items = new long[size];  
        numItems = first = last = 0;  
    }  
  
    public synchronized void put(long item){  
        if (numItems == capacity)  
            try{wait();} catch(Exception e){};  
        items[last] = item;  
        last = (last + 1) % capacity;  
        numItems++;  
        notify();  
    }  
  
    public synchronized long get() {  
        long valor;  
        if (numItems == 0)  
            try{wait();} catch(Exception e){};  
        valor = items[first];  
        first = (first + 1) % capacity;  
        numItems--;  
        notify();  
        return valor;  
    }  
}
```

```
class ProdCons {  
    static BoundedBuffer buf=  
        new BoundedBuffer(1);  
  
    public static void main(String[] args) {  
        new Thread(new Runnable() {  
            public void run() {  
                for (int i=0; i<10; i++) {  
                    buf.put(i);  
                } }, "producer").start();  
  
        new Thread(new Runnable() {  
            public void run() {  
                for (int i=0; i<10; i++) {  
                    System.out.println(buf.get());  
                } }, "consumer").start();  
    }  
}
```



## ACTIVIDAD 4 – Solución

1.	Al ejecutar este programa se crearán 10 hilos productores y 10 hilos consumidores.	F
2.	Este código no es correcto pues en Java debe utilizarse siempre <i>notifyAll()</i> en lugar de <i>notify()</i> .	F
3.	Este código no funciona, pues al llamar a <i>wait()</i> no se dejaría abierto el monitor y nadie más podría acceder a él.	F
4.	Este programa no ejecuta ninguno de los hilos generados, pues no se invoca el método <i>run()</i> de éstos.	F
5.	Si los hilos invocaran <i>Thread.currentThread().getName()</i> en su método <i>run()</i> , obtendrían el nombre interno que les asigna Java por defecto.	F



# ACTIVIDAD 5

```
public class gestorAparcamiento {  
    private int n = 100;  
    private int nCoches = 0;  
    private int nEspNorte = 0;  
    private int nEspSur = 0;  
    private boolean turnS = true;  
    private boolean turnN = true;  
  
    public synchronized void entrarSur(){  
        nEspSur++;  
        while(nCoches == n || !turnS) wait();  
        nEspSur--;  
        nCoches ++;  
        if (nCoches == n) {turnS = false; turnN = true;}  
    }  
  
    public synchronized void entrarNorte(){  
        nEspNorte++;  
        while(nCoches == n || !turnN) wait();  
        nEspNorte--;  
        nCoches ++;  
        if (nCoches == n) {turnS = true; turnN = false;}  
    }  
  
    public synchronized void salir(){  
        if (nCoches < n ||  
            (nCoches == n && (nEspNorte == 0 || nEspSur == 0 ))){  
            turnN = true;  
            turnS = true;  
        }  
        nCoches --;  
        notifyAll();  
    }  
}
```



## ACTIVIDAD 5 - Solución

1.	El monitor limita el grado de concurrencia al aparcamiento a 100 coches.	V
2.	No pueden usar el aparcamiento simultáneamente coches que entren por el norte y coches que entren por el sur.	F
3.	El primer coche que entre en el aparcamiento habrá utilizado la entrada norte.	F
4.	El atributo n cuenta el número de coches que están en el aparcamiento.	F
5.	Cuando el aparcamiento está lleno y hay coches esperando en las dos entradas, se regula el acceso para que cuando hayan plazas libres, se vayan alternando los coches de ambas entradas.	V
6.	En la cola asociada a wait, no pueden haber esperando, simultáneamente, coches que quieren entrar por el norte y coches que quieren entrar por el sur.	F



# ACTIVIDAD 6

```
public class GestorDePerlas {  
  
    final static private int NMaxBlancas = 50;  
    final static private int NMaxAzules = 50;  
  
    private int NBlancas = 0;  
    private int NAzules = 0;  
  
    private boolean PedidoEnCurso = false;  
  
    public synchronized void AñadirBlanca() {  
        NBlancas = NBlancas ++;  
        notifyAll();  
        while (NBlancas == NMaxBlancas)  
            try {wait();} catch (InterruptedException e){};  
    }  
  
    public synchronized void AñadirAzul() {  
        NAzules = NAzules ++;  
        notifyAll();  
        while (NAzules == NMaxAzules)  
            try {wait();} catch (InterruptedException e){};  
    }  
  
    public synchronized void SolicitarPedido(int SolBlancas,  
                                             int SolAzules) {  
  
        while (PedidoEnCurso)  
            try {wait();}  
            catch (InterruptedException e){};  
  
        PedidoEnCurso = true;  
  
        while (SolBlancas > NBlancas || SolAzules > NAzules)  
            try {wait();} catch (InterruptedException e){};  
  
        NBlancas = NBlancas - SolBlancas;  
        NAzules = NAzules - SolAzules;  
        PedidoEnCurso = false;  
        notifyAll();  
    }  
}
```



## ACTIVIDAD 6 - Solución

1.	En esta solución se puede sobrepasar el máximo número de piezas blancas o azules en los cestos, puesto que se incrementan los contadores antes de comprobar si caben.	F
2.	El atributo PedidoEnCurso es necesario para proporcionar exclusión mútua en el acceso al método SolicitarPedido .	F
3.	La solución no es correcta porque la invocación al método notifyAll en AñadirBlanca y AñadirAzul, debería ser la última instrucción en ambos métodos.	F
4.	El calificativo synchronized en los métodos AñadirBlanca y AñadirAzul, no es necesario ponerlo, ya que sólo hay un hilo que añade piezas blancas y un hilo que añade piezas azules.	F
5.	El atributo PedidoEnCurso se utiliza para conseguir que cuando un pedido P1 está esperando a que se completen las piezas solicitadas, los nuevos pedidos no se atenderán hasta que se complete P1.	V
6.	La solución propuesta para el monitor es correcta, y sincroniza adecuadamente según el enunciado, los proveedores de perlas y la gestión de los pedidos que realizan los montadores.	V



## Actividad 7 - Solución

1. El monitor tipo Hoare garantiza que tras una operación <i>notify()</i> el hilo reactivado encuentra el estado del monitor exactamente igual que estaba cuando se ejecutó dicho <i>notify</i> .	V
2. El monitor tipo Lampson-Redell utiliza una cola especial prioritaria sobre la entrada, donde esperan aquellos que han ejecutado <i>notify</i> .	F
3. Un monitor que siga el modelo de Hoare suspende al hilo que invoca a <i>notify()</i> , quedándose dicho hilo suspendido en una cola especial.	V
4. Un monitor que siga el modelo de Lampson/Redell jamás suspende a un hilo en la invocación a <i>notify()</i> .	V
5. El lenguaje Java proporciona por defecto monitores de tipo “Lampson-Redell”	V



## ACTIVIDAD 8

---

- ▶ En una calzada por la que circulan **coches** se tiene un paso de **peatones** cuya condición de corrección es que coches y peatones no pueden cruzarlo simultáneamente.
  - ▶ El paso de peatones está gobernado por el monitor *Crosswalk*, implantado siguiendo la variante de Lampson/Redell
  - ▶ Los métodos **enterX()** son invocados por los hilos de tipo X (C=Coche o P=Peatón) al llegar al paso de peatones. Los métodos **leaveX()** son invocados por los hilos de tipo X al abandonar el paso de peatones.
    - a) Describa la evolución del estado de cada uno de los atributos del monitor si se produce la siguiente secuencia ordenada de invocaciones a métodos del monitor
    - b) Observe que este monitor otorga prioridad a un tipo de hilos. ¿A cuál de las dos? ¿Por qué?
-



# ACTIVIDAD 8

```
monitor Crosswalk {  
    condition OKcars, OKpedestrians;  
    int c, c_waiting, p, p_waiting;  
  
    public Crosswalk() {  
        c = c_waiting = p = p_waiting = 0;  
    }  
  
    entry void enterC() {  
        c_waiting++;  
        while (COND-1) OKcars.wait();  
        c_waiting--;  
        c++;  
        OKcars.notify();  
    }  
  
    entry void leaveC() {  
        c--;  
        OKcars.notify();  
        OKpedestrians.notify();  
    }  
}
```

```
entry void enterP() {  
    p_waiting++;  
    while (COND-2) OKpedestrians.wait();  
    p_waiting--;  
    p++;  
    OKpedestrians.notify();  
}  
  
entry void leaveP() {  
    p--;  
    OKcars.notify();  
    OKpedestrians.notify();  
}
```

COND-1 ≡ (p > 0) y COND-2 ≡ (c > 0) || (c\_waiting > 0)



## ACTIVIDAD 8a

- a) Describa la evolución del estado de cada uno de los atributos del monitor si se produce la siguiente secuencia ordenada de invocaciones a métodos del monitor

	Método invocado	c_waiting	c	Cola OKcars	p_waiting	p	Cola OKpedestrians	Cola entrada al monitor
0)	(inicial)	0	0	vacía	0	0	vacía	vacía
1)	P1:M.enterP();					1		
2)	P2:M.enterP();					2		
3)	C1:M.enterC();	1		C1				
4)	P3:M.enterP();				1		P3	
5)	C2:M.enterC();	2		C1,C2				
6)	P1:M.leaveP();			C2,C1		1	P3	<del>C1,P3</del>
7)	C3:M.enterC();	3		C2,C1,C3				
8)	P2:M.leaveP();	0	3	-	1	0	P3	<del>C2,P3,C1,C3</del>



## ACTIVIDAD 8c

- c) Se requiere otorgar ahora prioridad al otro tipo de hilo. Rellene esta tabla explicativa de la sincronización condicional que se necesita. Indique por qué motivos deberían esperar los hilos en cada método, qué estados se modifican si los métodos se ejecutan por completo, y a quiénes se tendría que notificar y por qué motivo.

Crosswalk	E s p e r a cuando..	M o d i f i c a estado...	Notifica a.. (indicar por qué)
enterC()	Hay peatones pasando o esperando pasar	incrementa coches pasando	coches esperando pasar, para que puedan pasar todos juntos
leaveC()		decrementa coches pasando	peatones y coches
enterP()	Hay coches pasando	Peatones esperando e incrementa peatones pasando	peatones esperando pasar, para que puedan pasar todos juntos
leaveP()		decrementa peatones pasando	peatones y coches



## ACTIVIDAD 8d

```
monitor Crosswalk {  
    condition OKcars, OKpedestrians;  
    int c, c_waiting, p, p_waiting;  
  
    public Crosswalk() {  
        c = c_waiting = p = p_waiting = 0;  
    }  
  
    entry void enterC() {  
        c_waiting++;  
        while (COND-1) OKcars.wait();  
        c_waiting--;  
        c++;  
        OKcars.notify();  
    }  
  
    entry void leaveC() {  
        c--;  
        OKpedestrians.notify();  
        OKcars.notify();  
    }  
}
```

```
entry void enterP() {  
    p_waiting++;  
    while (COND-2) OKpedestrians.wait();  
    p_waiting--;  
    p++;  
    OKpedestrians.notify();  
}  
  
entry void leaveP() {  
    p--;  
    OKpedestrians.notify();  
    OKcars.notify();  
}  
}
```

COND-1 ≡ (p >0) || (p\_waiting >0) y COND-2 ≡ (c>0)



## ACTIVIDAD 8e

	Método invocado	c_waiting	c	Cola OKcars	p_waiting	p	Cola OKpedestrians	Cola entrada al monitor
0)	(inicial)	0	0	vacía	0	0	vacía	vacía
1)	P1:M.enterP();					1		
2)	P2:M.enterP();					2		
3)	C1:M.enterC();	1		C1				
4)	P3:M.enterP();					3		
5)	C2:M.enterC();	2		C1,C2				
6)	P1:M.leaveP();			C2,C1		2		C1
7)	C3:M.enterC();	3		C2,C1,C3				
8)	P4:M.enterP();					3		
9)	P2:M.leaveP();	3		C1,C3,C2		2		C2



## ACTIVIDAD 8f

	Método invocado	c_waiting	c	Cola OKcars	p_waiting	p	Cola OKpedestrians	Cola entrada al monitor
9)	P2:M.leaveP();	3		C1,C3,C2		2		C2
10)	P3:M.leaveP();			C3,C2,C1		1		C1
11)	P4:M.leaveP();	0	3			0		C3,C2,C1
12)	C2:M.leaveC();			2				
13)	C1:M.leaveC();			1				
14)	C3:M.leaveC();			0				

- Los pasos 10 y 11 indistinto P3 ó P4
- Los pasos 12, 13 y 14 en cualquier orden



## ACTIVIDAD 9

```
monitor SynchronousLink {
    condition OKsender, OKreceiver;
    int senders_waiting, receivers_waiting;
    Message msg;

    public SynchronousLink() {
        senders_waiting = receivers_waiting = 0;
        msg = null;
    }

    entry void send(Message m) {
        if (receivers_waiting > 0) {
            msg = m;
            OKreceiver.notify();
        } else {
            senders_waiting++;
            OKsender.wait();
        }
    }

    senders_waiting--;
    msg = m;
}

entry Message receive() {
    if (senders_waiting > 0) {
        OKsender.notify();
    } else {
        receivers_waiting++;
        OKreceiver.wait();
        receivers_waiting--;
    }
    return msg;
}
```



## ACTIVIDAD 9c

```
public class SynchronousLink {  
    condition OKsender, OKreceiver,  
    int senders_waiting, receivers_waiting;  
    Message msg;  
  
    public SynchronousLink() {  
        senders_waiting = receivers_waiting = 0;  
        msg = null;  
    }  
  
    public synchronized void send(Message m) {  
        if (receivers_waiting > 0) {  
            msg = m;  
            OKreceiver.notify();  
        } else {  
            msg=m;  
            senders_waiting++;  
        }  
        try {  
            OKsender.wait();  
        } catch (InterruptedException e) {}  
    }  
    senders_waiting--;  
    msg = m;  
}  
}  
  
public synchronized Message receive() {  
    if (senders_waiting > 0) {  
        OKsender.notify();  
    } else {  
        receivers_waiting++;  
        try {  
            OKreceiver.wait();  
        } catch (InterruptedException e) {}  
        receivers_waiting--;  
    }  
    return msg;  
}
```



# ACTIVIDAD 10 - Hormigas

```
// Opción A
monitor Territorio {
    boolean[N][N] ocupada;
    condition[N][N] libre;
    entry void desplaza(int x,y,x',y') {
        if (ocupada[x',y'])
            libre[x',y'].wait();
        //actualiza matriz
        ocupada[x',y']=true;
        ocupada[x,y]=false;
        //para avisar a quien quiere ir a x,y
        libre[x,y].notify();
    }
}
```

**OK Hoare**

```
// Opción B
monitor Territorio {
    boolean[N][N] ocupada;
    condition libre;
    entry void desplaza(int x,y,x',y') {
        while (ocupada[x',y']){
            libre.notify();
            libre.wait();
        };
        //actualiza matriz
        ocupada[x',y']=true;
        ocupada[x,y]=false;
        //para avisar a quien quiere ir a x,y
        libre.notify();
    }
}
```

**OK Lampson-Redell**

- Analizar ambas soluciones al problema de la colonia de hormigas e indicar en qué tipo de monitor funcionarían correctamente.



## ACTIVIDAD II

- a) ¿Qué ocurre con monitor Hoare?  
b) ¿Qué ocurre con monitor Lampson Redell?

```
monitor Rio {  
    condition entrar,salir;  
    int dentro=0;  
  
    entry void lobo_entrar()  
    {  
        // lobo tiene hambre y sed  
        if ( dentro == 1) comer_cordero();  
    }  
  
    entry void lobo_salir()  
    { // lobo deja de beber }  
  
    comer_cordero(){  
        //ñam  
    }  
}
```

```
entry void cordero_entrar()  
{ // cordero tiene sed  
    entrar.notify();  
    if ( dentro == 0) entrar.wait();  
    dentro++;  
    salir.notify();  
}  
  
entry void cordero_salir()  
{ salir.notify();  
    if ( dentro == 2) salir.wait();  
    dentro--;  
    // cordero deja de beber  
}  
}// fin monitor
```



## ACTIVIDAD II - Solución

- a) Si se utiliza la variante de monitores de Hoare, ¿los lobos y los corderos acceden al río en exclusión mutua? Es decir, ¿si hay lobos bebiendo no puede haber corderos bebiendo y viceversa?

No. Lobos y corderos pueden beber simultáneamente del río.

¿Se podrán comer los lobos a los corderos?

No, pues nunca llegan a quedarse solos (dentro !=1)

- b) Si la variante de monitor utilizada es la de Lampson Redell, ¿se podrán comer los lobos a los corderos? Justifique su respuesta con una traza.

Sí, pues tanto al entrar como al salir corderos dentro puede llegar a valer 1.

C1.entrar, C2.entrar deja a C2 bloqueado y dentro==1, L.entrar (**se come al cordero**). C3.entrar deja dentro==3 y pasan C2 y C3, C3.salir, C2.salir deja a C2 bloqueado, C1.salir deja a C1 bloqueado y desbloquea a C2 quien deja dentro==1. L2.entrar se come a un cordero.

El código resuelve la siguiente estrategia de supervivencia: nunca ha de quedarse un cordero sólo en el río, en presencia de uno o varios lobos. Si hay varios corderos los lobos no se los pueden comer.



## ACTIVIDAD 12

```
monitor Baño {  
    condition lleno, sexo_opuesto;  
    int ocupantes=0, capacidad=3;  
    boolean mujeres=false;  
    entry entra_adulto(boolean esMujer){  
        if (ocupantes+1 > capacidad) lleno.wait();  
        if (ocupantes>0 && mujeres!= esMujer){  
            lleno.notify();  
            sexo_opuesto.wait();  
            sexo_opuesto.notify();  
        }  
        ocupantes++;  
        mujeres = esMujer;  
    }  
  
    entry sale_adulto(){  
        ocupantes--;  
        if (ocupantes+1 == capacidad)  
            lleno.notify();  
        else if (ocupantes==0)  
            sexo_opuesto.notify();  
    }  
}
```

- a) ¿Qué ocurre con variante Hoare?  
b) ¿Qué ocurre con variante Lampson Redell?

- En una empresa se utiliza un baño mixto en el que pueden entrar tanto hombres como mujeres, pero con la condición de que simultáneamente sólo pueda haber personas de un único sexo.
- Además, el baño tiene una capacidad limitada de 3 personas.



## ACTIVIDAD 12 - Solución

---

- a) Con la variante de Hoare, ¿podrían entrar adultos de distinto sexo en el baño?
- ▶ No  
¿se podría sobrepasar la capacidad del baño?.
  - ▶ Sí. Cuando pasan los del sexo\_opuesto lo hacen todos juntos sin comprobar la capacidad
- b) Con la variante de Hoare, ¿habría diferencia en la ejecución del monitor si en vez de las sentencias "if" del método entra\_adulto tuviéramos sentencias "while"?
- ▶ No en cuanto al comportamiento = al de la opción a)



## ACTIVIDAD 12 - solución

- c) Realice el mismo análisis para la variante de Lampson-Redell.
- ▶ Pueden entrar adultos del mismo sexo en el baño. Cuando el último ocupante (H) sale, notifica al sexo\_opuesto (M) quien va a parar a la cola de entrada, compitiendo con otros hilos por el acceso al monitor. Si accediera un H, encontraría ocupantes a 0 y entraría al baño. Posteriormente, la M continuaría y pasaría al baño, reactivando a su vez a otras M.
- d) ¿Se hace uso en este monitor de la "reactivación en cascada"? Si es así, ¿cómo se habría podido implementar en Java?
- ▶ Sí, para que pasen juntos los del mismo sexo que esperan en sexo\_opuesto. En Java se usaría un `notifyAll()`.



## ACTIVIDAD 13

► Dado el siguiente programa:

```
public class TestIt {  
    public class Test {  
        public synchronized void test (Test t) {  
            t.SayHola ();  
        }  
        public synchronized void SayHola () {  
            System.out.println ("Hola");  
        }  
    }  
    public TestIt () {  
        Test t1 = new Test();  
        Test t2 = new Test();  
        new Thread (new Runnable () {public void run() {  
            t1.test(t2);  
        }}).start();  
        new Thread (new Runnable () {public void run() {  
            t2.test(t1);  
        }}).start();  
    }  
    public static void main (String args[]) {  
        new TestIt();  
    }  
}
```



# ACTIVIDAD 13

► Analizar el siguiente programa:

```
public class TestIt {  
    public class Test {  
        public synchronized void test (Test t) {  
            t.SayHola ();  
        }  
        public synchronized void SayHola () {  
            System.out.println ("Hola");  
        }  
    }  
    public TestIt () {  
        Test t1 = new Test();  
        Test t2 = new Test();  
        new Thread (new Runnable () {public void run() {  
            t1.test(t2);  
        }}).start();  
        new Thread (new Runnable () {public void run() {  
            t2.test(t1);  
        }}).start();  
    }  
    public static void main (String args[]) {  
        new TestIt();  
    }  
}
```

Monitor

Crea y lanza hiloA

Crea y lanza hiloB

Crea objeto clase TestIt

Crea monitores t1 y t2

hiloA	hiloB
t1.test(t2); // cierra lock t1; (*)	t2.test(t1); // cierra lock t2; (*)
t2.SayHola(); // cierra lock t2; escribe Hola; // abre ambos locks;	t1.SayHola(); // cierra lock t1; escribe Hola; // abre ambos locks;

Puede haber  
interbloqueo si se  
interrumpe en (\*)



# ACTIVIDAD 13 - Solución

```
public class TestIt {  
    public class Test {  
        public synchronized void test (Test t) {  
            t.SayHola ();  
        }  
        public synchronized void SayHola () {  
            System.out.println ("Hola");  
        }  
    }  
    public TestIt () {  
        Test t1 = new Test();  
        Test t2 = new Test();  
        new Thread (new Runnable () {public void run() {  
            t1.test(t2);  
        }}).start();  
        new Thread (new Runnable () {public void run() {  
            t2.test(t1);  
        }}).start();  
    }  
}
```

```
    public static void main (String args[]) {  
        new TestIt();  
    }  
}
```

hiloA	hiloB
t1.test(t2); <i>// cierra lock t1;</i> <b>(*)</b>	t2.test(t1); <i>// cierra lock t2;</i> <b>(*)</b>
t2.SayHola(); <i>// cierra lock t2;</i> escribe Hola; <i>// abre ambos locks;</i>	t1.SayHola(); <i>// cierra lock t1;</i> escribe Hola; <i>// abre ambos locks;</i>

Si por la pantalla vemos una vez "Hola", seguro que veremos "Hola" dos veces.

V

Se trata de un programa correcto, libre de condiciones de carrera y libre de interbloqueos, pues los métodos son "synchronized".

F

En toda ejecución, al menos veremos por la pantalla "Hola", una vez.

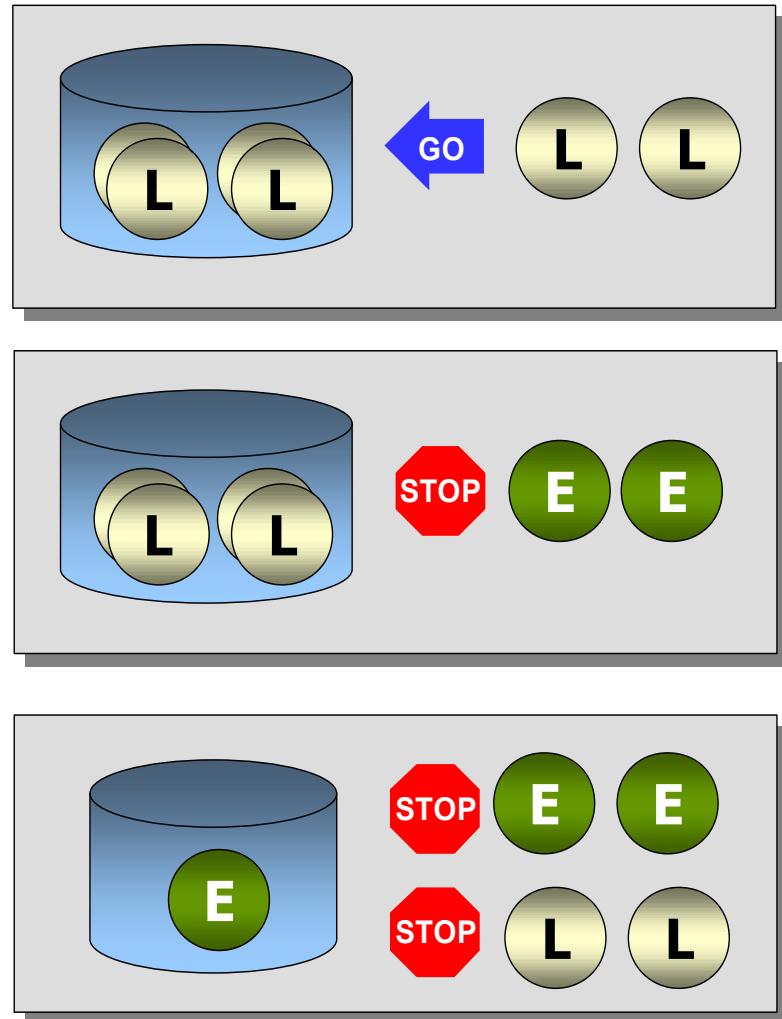
F

Puede presentar condiciones de carrera en el acceso a las variables t1 y t2

F

## Ejemplos Académicos.- Lectores/Escritores

- ▶ Existe un recurso común (ej fichero), y distintos hilos que acceden a él
  - ▶ Algunos de ellos se limitan a leer su contenido (sin modificar el estado). Se denominan lectores
  - ▶ Otros modifican el contenido (modifican el estado). Se denominan escritores
- ▶ Para garantizar un acceso correcto al recurso, se fijan las siguientes normas
  - ▶ Varios lectores pueden acceder a la vez
  - ▶ Un escritor excluye la presencia de otros hilos (ya sean lectores o escritores)





# Lectores-escritores con monitores

```
Monitor control_LE {  
    int lectores, escritores, le_waiting;  
    condition leer, escribir;  
  
    public control_LE() {  
        lectores=escritores= le_waiting=0;  
    }  
  
    entry void pre-leer() {  
        le_waiting ++;  
        if (escritores > 0) leer.wait();  
        le_waiting --;  
        lectores = lectores+1;  
        leer.notify(); // despertar al siguiente  
    }  
  
    entry void post-leer() {  
        lectores = lectores-1;  
        if (lectores == 0) escribir.notify();  
    }  
}
```

- Analizar soluciones según tipo de monitor y propiedades de vivacidad

OK Hoare

```
entry void pre-escribir() {  
    if (escritores > 0 || lectores > 0 )  
        escribir.wait();  
    escritores = escritores+1;  
}  
  
entry void post-escribir() {  
    escritores = escritores-1;  
    if (le_waiting > 0)  
        leer.notify();  
    else escribir.notify();  
}  
}
```



# Lectores-escritores con monitores

```
Monitor control_LE {  
    int lectores, escritores, es_waiting;  
    condition leer, escribir;  
    public control_LE() {  
        lectores=escritores= es_waiting=0;  
    }  
  
    entry void pre-leer() {  
        if (escritores > 0 || es_waiting >0)  
            leer.wait();  
        lectores = lectores+1;  
        leer.notify(); // despertar al siguiente  
    }  
  
    entry void post-leer() {  
        lectores = lectores-1;  
        if (lectores == 0) escribir.notify();  
    }  
}
```

OK Hoare

```
entry void pre-escribir() {  
    es_waiting ++;  
    if (escritores > 0 || lectores > 0 )  
        escribir.wait();  
    es_waiting --;  
    escritores = escritores+1;  
}  
  
entry void post-escribir() {  
    escritores = escritores-1;  
    if (es_waiting > 0)  
        escribir.notify() ;  
    else leer.notify();  
}  
}
```



# Lectores-escritores con monitores

```
Monitor control_LE {  
    int lectores, escritores, le_waiting,es_waiting;  
    condition leer, escribir;  
  
    public control_LE() {  
        lectores=escritores=le_waiting=es_waiting=0;  
    }  
  
    entry void pre-leer() {  
        le_waiting ++;  
        if (escritores > 0 || es_waiting >0)  
            leer.wait();  
        le_waiting ++;  
        lectores = lectores+1;  
        leer.notify(); // despertar al siguiente  
    }  
  
    entry void post-leer() {  
        lectores = lectores-1;  
        if (lectores == 0) escribir.notify();  
    }  
}
```

OK Hoare

```
entry void pre-escribir() {  
    es_waiting ++;  
    if (escritores > 0 || lectores > 0 )  
        escribir.wait();  
    es_waiting --;  
    escritores = escritores+1;  
}  
  
entry void post-escribir() {  
    escritores = escritores-1;  
    if (le_waiting > 0)  
        leer.notify();  
    else escribir.notify();  
}  
}
```



# Lectores-escritores en Java

```
class control_LE {  
    int lectores=0;  
    boolean escritores= false;  
  
    synchronized void pre-leer() {  
        while (escritores)  
            try {  
                wait();  
            } catch (InterruptedException e) {};  
        lectores = lectores+1;  
        notifyAll();  
    }  
  
    synchronized void post-leer() {  
        lectores = lectores-1;  
        if (lectores == 0) notifyAll();  
    }  
}
```

No podemos controlar a la salida a quien le damos el turno

```
synchronized void pre-escribir() {  
    while (escritores || (lectores > 0) )  
        try {  
            wait();  
        } catch (InterruptedException e) {};  
    escritores = true;  
}  
  
synchronized void post-escribir() {  
    escritores= false;  
    notifyAll();  
}  
}
```