

PROGRAMACION CONCURRENTE

Monitores

la noción de un monitor fue presentada inicialmente por
dijkstra (1971),

posteriormente por
brinch hansen (1973)

y después fue refinada por
hoare (1974).

Introducción

- Los semáforos son primitivas con las cuales es difícil expresar una solución a grandes problemas de concurrencia y su presencia en programas concurrentes incrementa la ya existente dificultad para probar que los programas son correctos.
- Los semáforos tienen algunas debilidades:
 - La omisión de una de estas primitivas puede corromper la operación de un sistema concurrente.
 - El control de la concurrencia es responsabilidad del programador.
 - Las primitivas de control se encuentran esparcidas por todo el sistema.

Mas Inconveniente de los semáforos

- Nos puede llevar fácilmente a errores
 - Errores transitorios
 - La ejecución de cada sección crítica sobre un semáforo debe:
 - comenzar con `wait()` y
 - terminar con `signal()` No se puede restringir el tipo de operación sobre el recurso
 - Se debe incluir todas las sentencias críticas en la sección crítica
 - Tanto la sección crítica como la sincronización se implementan usando las mismas primitivas y luego son difíciles de identificar
 - Los programas que usan semáforos son difíciles de mantener,
 - El código de sincronización esta repartido entre diferentes procesos

Que buscamos?

- Compartir Recursos

- 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

- ¿como implementamos un recurso compartido especificado?
 - Métodos synchronized
 - Monitores
 - Paso de Mensajes

Definición

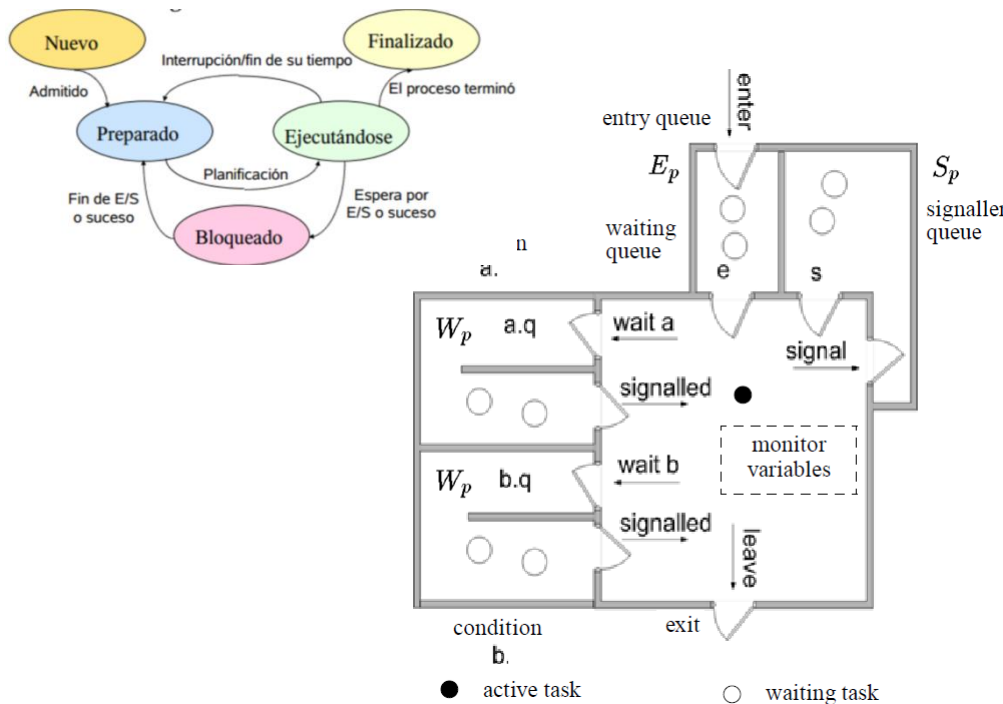
- Un monitor :
 - Es un mecanismo de software (de alto nivel) para control de concurrencia que contiene los datos y los procedimientos necesarios para realizar la asignación de un determinado recurso o grupo de recursos compartidos *reutilizables en serie*.
 - *Un monitor contiene*
 - *Variables que representan el estado del recurso*
 - *Procedimientos que implementan operaciones sobre el recurso*
- Se usa para manejar todas las funciones de concurrencia:
 - comunicación entre procesos
 - localización física de recursos en una región crítica
- Podríamos decir que un monitor es una instancia de una clase que puede ser usada de forma segura por múltiples threads

Como Hacerlo

- El monitor tiene varios procedimientos que manipulan datos internos y existe una parte de inicialización .
 - El monitor puede ser visto como una aduana en donde se permite o no el acceso a un recurso compartido o se espera por un tramite.
- El código del monitor consta de 2 partes lógicas:
 - El algoritmo para la manipulación del recurso y sincronización.
 - El mecanismo para la asignación del orden en el cual los procesos asociados pueden compartir el recurso y/o son sincronizados.
- Desde el punto de vista de los procesos activos que usan al monitor, este ofrece un conjunto de procedimientos para utilizar recursos compartidos y sincronizar actividades
- Se cumple que:
 - Todos los métodos de un monitor deben ejecutarse en exclusión mutua
 - Se define la sincronización por condición de cada uno de los métodos del monitor
 - bloquear procesos hasta que se cumplan las condiciones para que puedan ejecutar
 - notificar (señalizar) a un proceso bloqueado que puede continuar su ejecución cuando se cumplan las condiciones para ejecutarse

Estructura básica de un monitor

- Para asegurar que un proceso obtenga el recurso que espera, el monitor (*la lógica del monitor*) debe darle prioridad sobre los nuevos procesos que solicitan entrar al monitor.
- De otra manera, los nuevos procesos tomaran el recurso antes de que el proceso que espera lo haga, esto puede llevar al proceso que espera a la *postergación indefinida*.



Elementos que componen al Monitor

- Conjunto de **variables locales**, denominadas permanentes, almacenan el estado interno del recurso, y los procedimientos
- Código de inicialización (**constructor**)
- Conjunto de **procedimientos interno** que maneja las variables permanentes
- Declaración de los **procedimientos** que son **exportados** pueden ser accedidos por los procesos activos

```
Monitor nombre_monitor
    var variables_locales;
    export procedimientos_exportados;

    procedure proc1(parametros){
        var variables_locales;
        begin
        //Codigo del procedimiento
        end

    procedure proc2(parametros){
        var variables_locales;
        begin
        //Codigo del procedimiento
        end

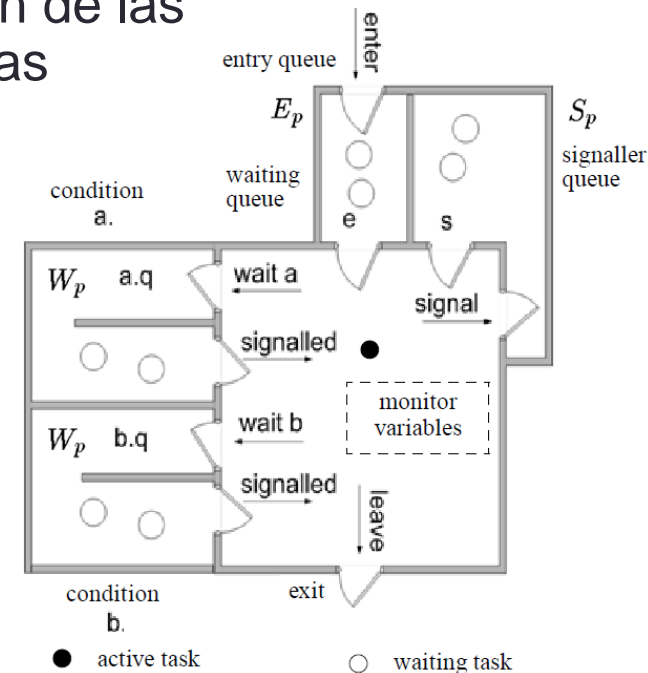
    begin
        //codigo de inicializacion
    end
```


Gestión del Acceso Exclusivo

- El control de la exclusión mutua esta basado en la **cola asociada del monitor**
- Gestión de la Cola
 - Un proceso activo esta ejecutando un procedimiento del monitor (el procedimiento esta en el monitor) y otro proceso activo trata de ejecutar el mismo u otro, el código de acceso al monitor bloquea la llamada y lo manda a la cola
 - Cuando un proceso activo abandona el monitor , el monitor toma el proceso que esta el frente de la cola y lo desbloquea
 - Si la cola del monitor esta vacía, este queda libre
- Lo anterior garantiza la exclusión mutua, el responsable de esto es el monitor

Clasificación de Monitores

- Los monitores se dividen en función de señal la operación:
 - señal explícita
 - Es un monitor con una declaración explícita de la variable señal
 - Señal implícita
 - Es un monitor de señal automática, que no tiene ninguna declaración
- A su vez, la clasificación esta basada en función de las prioridades relativas asociadas a estas tres colas
 - Prioridad relativa de la cola de entrada
 - Prioridad relativa de la cola de espera
 - Prioridad relativa de la cola de señalización.
- Las políticas que nos interesan son:
 - Wait and Notify $E_p = W_p < S_p$
 - Signal and Wait $E_p = S_p < W_p$
 - Signal and Continue $\bar{E}_p < \bar{W}_p < \bar{S}_p$
 - Signal and Urgent $\bar{E}_p < \bar{S}_p < \bar{W}_p$



Monitores: Políticas Signal

- **Signal and Continue (SC):** El proceso que señala mantiene el mutex y el proceso despertado debe competir por el mutex para ejecutar

$$E_p < W_p < S_p$$

- **Signal and Wait (SW):** El proceso que señala es bloqueado y debe competir de nuevo por el mutex para continuar y el proceso despertado adquiere el mutex y continua su ejecución

$$E_p = S_p < W_p$$

- **Signal and Urgent Wait (SU):** El proceso que señala es bloqueado pero será el primero en conseguir el mutex cuando lo libere el proceso despertado


$$E_p < S_p < W_p$$

- **Signal and Exit (SX):** El proceso que señala sale del metodo y el proceso despertado coge directamente el mutex para ejecutar

$$E_p = W_p < S_p$$

Monitor: Variable de Condición

- Una **variable de condición** son “contenedores” de subprocesos que esperan una determinada condición.
 - Los monitores proporcionan un mecanismo para que los subprocesos otorguen temporalmente acceso exclusivo para esperar a que se cumpla alguna condición, antes de recuperar el acceso exclusivo y reanudar su tarea.
 - Cada proceso puede requerir una sincronización distinta, por lo que hay que programar cada caso, para ello, se usarán las variables “condición”
 - se usan para hacer esperar a un proceso hasta que determinada condición sobre el estado del monitor se “cumpla”
 - también para despertar a un proceso que estaba esperando por su causa
- instrucción **wait**:
 - el proceso invocador de la acción que la contiene queda “dormido”
 - y pasa a la cola asociada a la variable, en espera de ser despertado
- instrucción **signal**:
 - si la **cola** de la señal está **vacía**: no pasa nada, y la acción que la ejecuta sigue con su ejecución
 - al terminar, el monitor está disponible para otro proceso
 - si la **cola no** está **vacía**:
 - el primer proceso de la cola se despierta (pero no avanza) se saca de la cola
 - el proceso que la ejecuta sigue con su ejecución, hasta terminar el procedimiento



Depende de la política
implementada-
Schedule

Políticas Signal: Signal and Continue (SC)

- Con las *variables de condición sin bloqueo* , "señal y continuar"
 - la señalización no hace que el hilo de señalización pierda la ocupación del monitor.
 - En cambio, los hilos señalizados se mueven a la cola de entrada. No hay necesidad de cola de cortesía.

notify all c :

move all threads waiting on *c.q* to *e*

mueve todos los subprocesos que esperan una variable de condición a la cola de entrada y la da mas prioridad que los que esperan para entrar.

enter the monitor:

```
enter the method
if the monitor is locked
    add this thread to e
    block this thread
else
    lock the monitor
```

$$E_p < W_p < S_p$$

enter

notified

notified

b.q

a.q

wait a

wait b

notify

leave

notify c :

```
if there is a thread waiting on c.q
    select and remove one thread t from c.q
    >(t is called "the notified thread")
    move t to e
```

schedule :

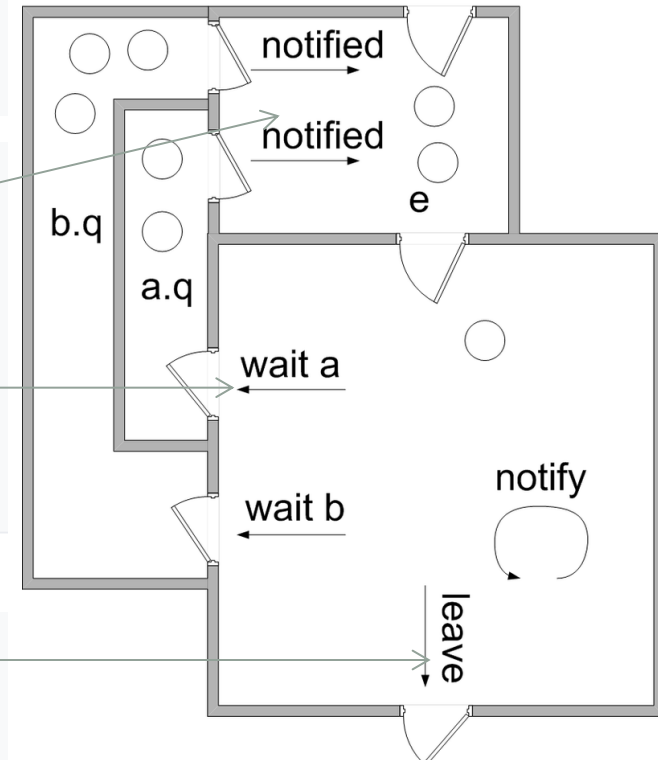
```
if there is a thread on e
    select and remove one thread from e and restart it
else
    unlock the monitor
```

wait c :

```
add this thread to c.q
schedule
block this thread
```

leave the monitor:

```
schedule
return from the method
```



Polticas Signal:

• Signal and Urgent Wait (SU)

$$E_n < S_n < W_p$$

La disciplina de señalización resultante se conoce como "*señal y espera urgente*", ya que el señalizador debe esperar, pero se le da prioridad sobre los hilos en la cola de entrada.

Una alternativa es "*señal y espera*", en la que no hay s-cola y el señalizador espera en la e-cola.

- hay dos colas de subprocesos asociados con cada objeto en el monitor
 - e es la cola de entrada
 - s es una cola de hilos que han señalado.
- para cada variable de condición c , hay una cola
 - c.q, que es una cola para subprocesos que esperan en la variable de condición c

```

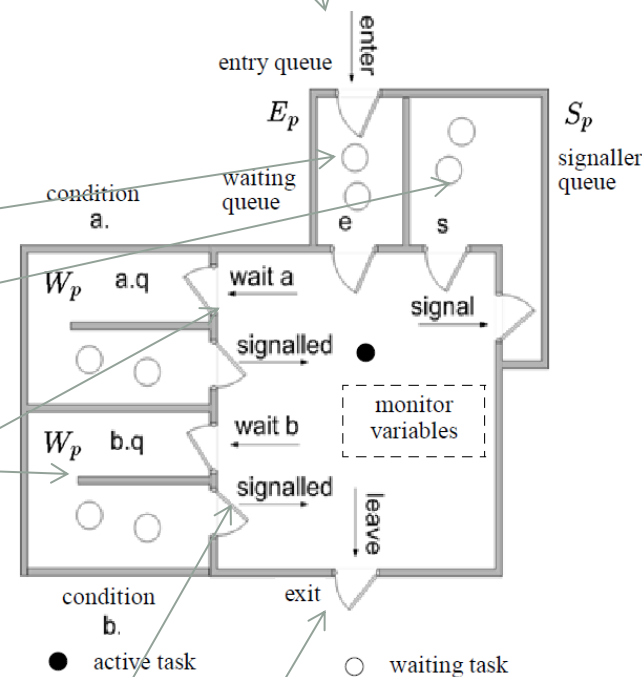
schedule :
  if there is a thread on s
    select and remove one thread from s and restart it
    (this thread will occupy the monitor next)
  else if there is a thread on e
    select and remove one thread from e and restart it
    (this thread will occupy the monitor next)
  else
    unlock the monitor
    (the monitor will become unoccupied)
    
```

```

wait c :
  add this thread to c.q
  schedule
  block this thread
    
```

```

enter the monitor:
  enter the method
  if the monitor is locked
    add this thread to e
    block this thread
  else
    lock the monitor
    
```



```

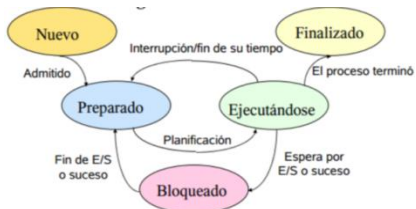
leave the monitor:
  schedule
  return from the method
    
```

```

signal c :
  if there is a thread waiting on c.q
    select and remove one such thread t from c.q
    (t is called "the signaled thread")
    add this thread to s
    restart t
    (so t will occupy the monitor next)
    block this thread
    
```

Como Wp se transfiere la urgencia a Wp

Proceso	Estado	Proceso	Estado
s	sig	eje	blo
s	wai	eje	pre
s	blo	a	eje
s	pre	a	eje



Políticas Signal:

• Signal and Wait (SW)

$$E_p = S_p < W_p$$

mueve los subprocesos que esperan una variable de condición al monitor como activo.

El proceso señalizador a la cola de espera, donde la prioridad es la de llegada.

signal c

```
if there is a thread waiting on c.q
  select and remove one thread t from c.q
  (t is called "the notified thread")
  move t to e
else
  schedule
  return from the method
```

• Signal and Exit (SX):

$$E_p = W_p < S_p$$

mueve los subprocesos que esperan una variable de condición a la cola de entrada, con la prioridad de llegada.

El proceso señalizador sigue en el monitor hasta terminar, al salir llama a schedule.

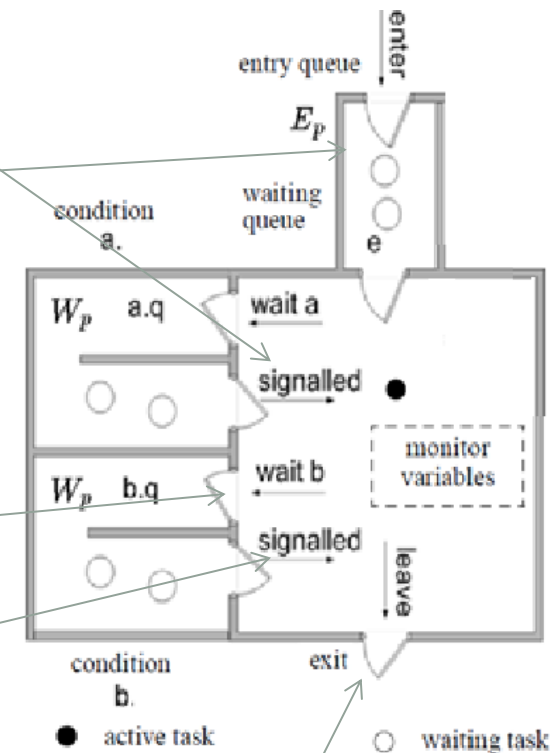
signal c and return :

```
if there is a thread waiting on c.q
  select and remove one such thread t from c.q
  (t is called "the signaled thread")
  restart t
  (so t will occupy the monitor next)
else
  schedule
  return from the method
```

schedule :

```
if there is a thread on e
  select and remove one thread from e and restart it
else
  unlock the monitor
```

```
enter the monitor:
  enter the method
  if the monitor is locked
    add this thread to e
    block this thread
  else
    lock the monitor
```



leave the monitor:

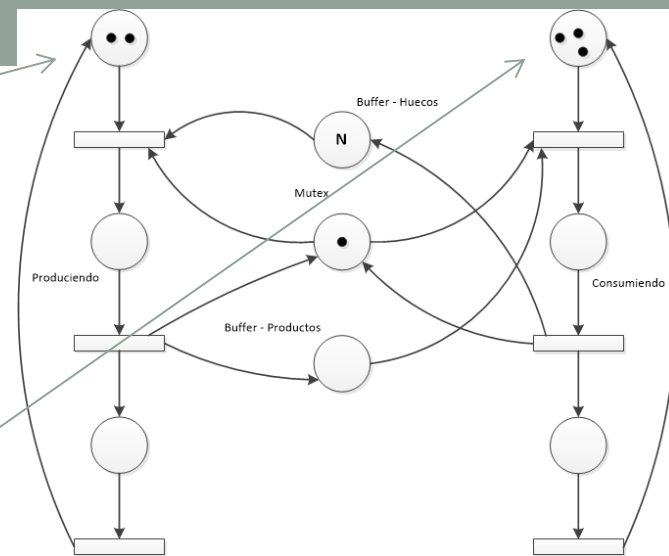
```
schedule
return from the method
```

Objetivo

- Nuestro objetivo solucionar un problemas de concurrencia que resultarían muy difíciles con semáforos.
- Hemos visto que las colas de monitores y condiciones interactúan de maneras no triviales.
- La solución que proponemos es un monitores que facilite identificar, separar y gestionar:
 - La lógica del Sistema
 - La política del Sistema
 - Que emerge de la relación entre las colas de entrada, condición y cortesía
 - De las colas de condición
 - Las acciones que se ejecutan
- Concretamente, tendremos que decidir:
 1. una estructura general para los métodos que implementan acciones de un recurso compartido (en el monitor),
 2. que condición en las colas se requiere para los bloqueos y
 3. una estrategia para efectuar los desbloqueos cuando hay varios hilos candidatos a ser desbloqueados.

Métodos en el monitor

```
4 public class Productor implements Runnable {
5     /**
6      * This is the producer thread for the bounded buffer problem.
7      */
8
9
10    private Buffer buffer;
11
12    public Productor(Buffer b) {
13        buffer = b;
14    }
15
16    public void run(){
17        Date message;
18        while (true) {
19            System.out.println("Producer napping");
20            SleepUtilities.nap();
21            // produce an item & enter it into the buffer
22            message = new Date();
23            System.out.println("Producer produced \"" + message + "\"");
24            buffer.insert(message);
25        }
26    }
27
28
29    private Buffer buffer;
30
31    public Consumidor(Buffer b) {
32        buffer = b;
33    }
34
35    public void run(){
36        Date message = null;
37        while (true){
38            System.out.println("Consumer napping");
39            SleepUtilities.nap();
40            // consume an item from the buffer
41            System.out.println("Consumer wants to consume");
42            message = (Date)buffer.remove();
43            System.out.println("Consumer consumed \"" + message + "\"");
44        }
45    }
46 }
```



- La estructura será

- ▾ productorConsumidor
 - > Buffer.java
 - > BufferL.java
 - > Consumidor.java
 - > Productor.java
 - > SleepUtilities.java
 - > ThreadApp2.java

Métodos en el monitor

```

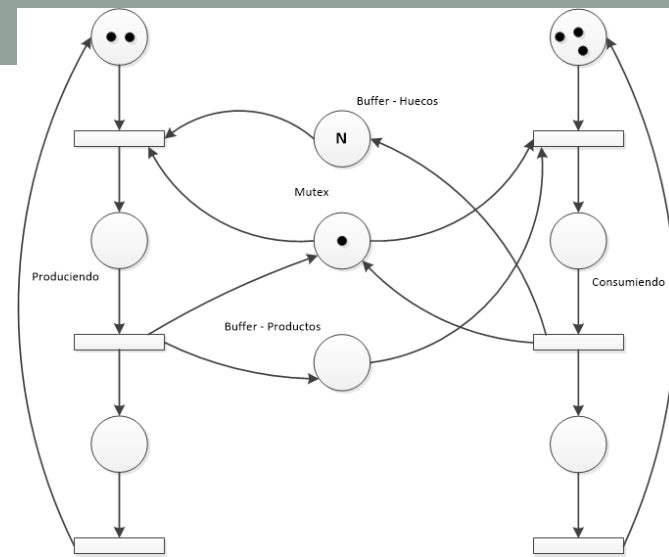
4 public class ThreadApp2 {
5
6     public static void main(String args[]) {
7         //instantiate (create) buffer shared by Producer & Consumer
8         Buffer sharedBuffer = new Buffer();
9         // create the producer and consumer threads
10        Thread producerThread = new Thread(new Productor(sharedBuffer));
11        Thread consumerThread = new Thread(new Consumidor(sharedBuffer));
12        //start() method allocates memory for a new thread in the JVM,
13        //and calls the run() method
14        producerThread.start();
15        consumerThread.start();
16    }
17 }
18

```

```

10 class SleepUtilities{
11
12     private static final int NAP_TIME = 3; //max nap time in seconds
13
14     /**
15      * Nap between zero and NAP_TIME seconds.
16      */
17     public static void nap() {
18         nap(NAP_TIME);
19     }
20
21     /**
22      * Nap between zero and duration seconds.
23      */
24     public static void nap(int duration) {
25         int sleeptime = (int) (NAP_TIME * Math.random() );
26         System.out.println("Nap for " + sleeptime + " seconds");
27         //Causes the currently executing thread to sleep (cease execution)
28         //for the specified number of milliseconds,
29         //subject to the precision and accuracy of system timers and schedulers.
30         try { Thread.sleep(sleeptime*1000); }
31         catch (InterruptedException e) {
32             //method sleep() throws InterruptedException - if any thread has interrupted the current thread.
33             System.out.println("ERROR in nap(): " + e);
34         }
35     }
36 }
37

```



- La estructura será

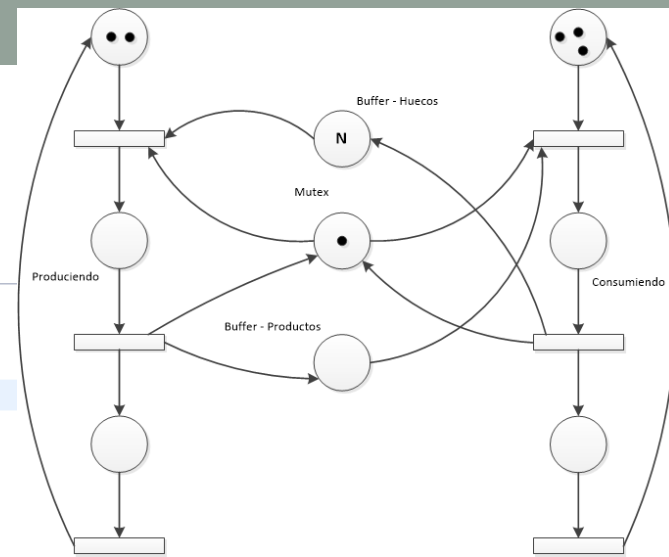
```

v [icon] productorConsumidor
  > [icon] Buffer.java
  > [icon] BufferI.java
  > [icon] Consumidor.java
  > [icon] Productor.java
  > [icon] SleepUtilities.java
  > [icon] ThreadApp2.java

```

Métodos en el monitor

```
1 package productorConsumidor;
2 import java.util.concurrent.*;
3
4 public class Buffer implements BufferI{
5     private static final int BUFFER_SIZE = 3; //max size of buffer array
6     private int count; //number of items currently in the buffer
7     private int in;    // points to the next free position in the buffer
8     private int out;   // points to the first filled position in the buffer
9     private Object[] buffer; //array of Objects
10    private Semaphore mutex; //provides limited access to the buffer (mutual exclusion)
11    private Semaphore empty; //keep track of the number of empty elements in the array
12    private Semaphore full; //keep track of the number of filled elements in the array
13    public Buffer(){
14        // buffer is initially empty
15        count = 0;
16        in = 0;
17        out = 0;
18        buffer = new Object[BUFFER_SIZE];
19        mutex = new Semaphore(1); //1 for mutual exclusion
20        empty = new Semaphore(BUFFER_SIZE); //array begins with all empty elements
21        full = new Semaphore(0); //array begins with no elements
22    }
23 }
```



- La estructura será

```
▼ [icon] productorConsumidor
  > [icon] Buffer.java
  > [icon] BufferI.java
  > [icon] Consumidor.java
  > [icon] Productor.java
  > [icon] SleepUtilities.java
  > [icon] ThreadApp2.java
```

```

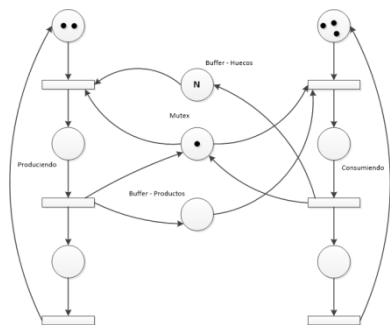
25 public void insert(Object item) {
26     /*while (count == BUFFER_SIZE){
27         // do nothing, if the buffer array cannot be used (because full)
28     } */
29     try{
30         empty.acquire(); //keep track of number of empty elements (value--)
31         //This provides synchronization for the producer,
32         //because this makes the producer stop running when buffer is full
33         mutex.acquire(); //mutual exclusion
34     }
35     catch (InterruptedException e) {
36         System.out.println("ERROR in insert(): " + e);
37     }
38
39     // add an item to the buffer
40     ++count;
41     buffer[in] = item;
42     //modulus (%) is the remainder of a division
43     //for example, 0%3=0, 1%3=1, 2%3=2, 3%3=0, 4%3=1, 5%3=2
44     in = (in + 1) % BUFFER_SIZE;
45
46     //buffer information feedback
47     if (count == BUFFER_SIZE){
48         System.out.println("BUFFER FULL "
49             + "Producer inserted \"" + item
50             + "\" count=" + count + ", "
51             + "in=" + in + ", out=" + out);
52     }
53     else{
54         System.out.println("Producer inserted \"" + item
55             + "\" count=" + count + ", "
56             + "in=" + in + ", out=" + out);
57     }
58
59     mutex.release(); //mutual exclusion
60     full.release(); //keep track of number of elements (value++)
61     //If buffer was empty, then this wakes up the Consumer
62 }

```

```

64 // consumer calls this method
65 public Object remove() {
66     Object item=null;
67     /* while (count == 0){
68         //if nothing in the buffer, then do nothing
69         //the buffer array cannot be used (because empty)
70     } */
71
72     try{
73         full.acquire(); //keep track of number of elements (value--)
74         //This provides synchronization for consumer,
75         //because this makes the Consumer stop running when buffer is empty
76         mutex.acquire(); //mutual exclusion
77     }
78     catch (InterruptedException e) {
79         System.out.println("ERROR in try(): " + e);
80     }
81
82     // remove an item from the buffer
83     --count;
84     item = buffer[out];
85     //modulus (%) is the remainder of a division
86     //for example, 0%3=0, 1%3=1, 2%3=2, 3%3=0, 4%3=1, 5%3=2
87     out = (out + 1) % BUFFER_SIZE;
88     //buffer information feedback
89     if (count == 0){
90         System.out.println("BUFFER EMPTY "
91             + "Consumer removed \"" + item
92             + "\" count=" + count + ", "
93             + "in=" + in + ", out=" + out);
94     }
95     else{
96         System.out.println("Consumer removed \"" + item
97             + "\" count=" + count + ", "
98             + "in=" + in + ", out=" + out);
99     }
100     mutex.release(); //mutual exclusion
101     empty.release(); //keep track of number of empty elements (value++)
102     //if buffer was full, then this wakes up the Producer
103     return item;
104 }

```

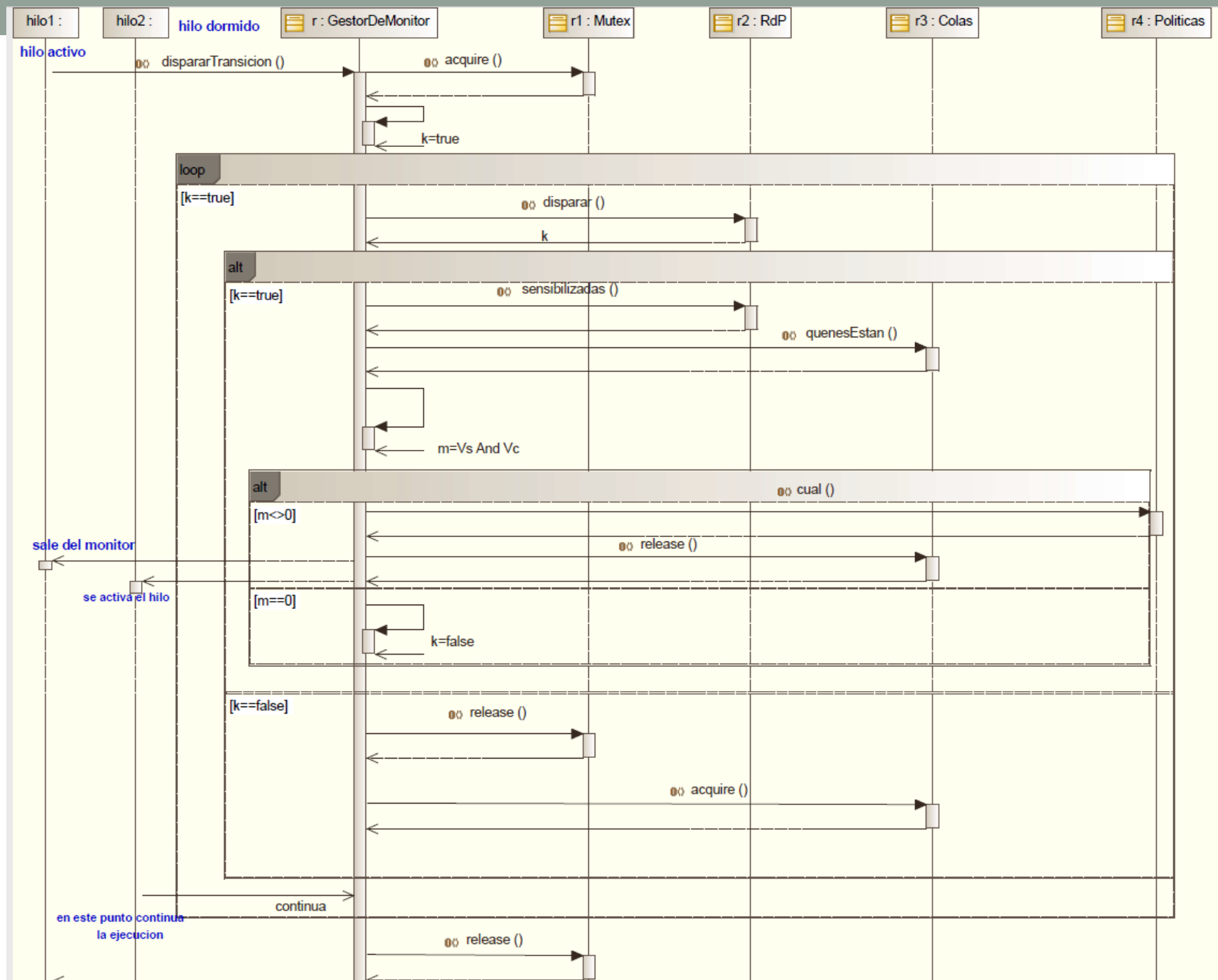


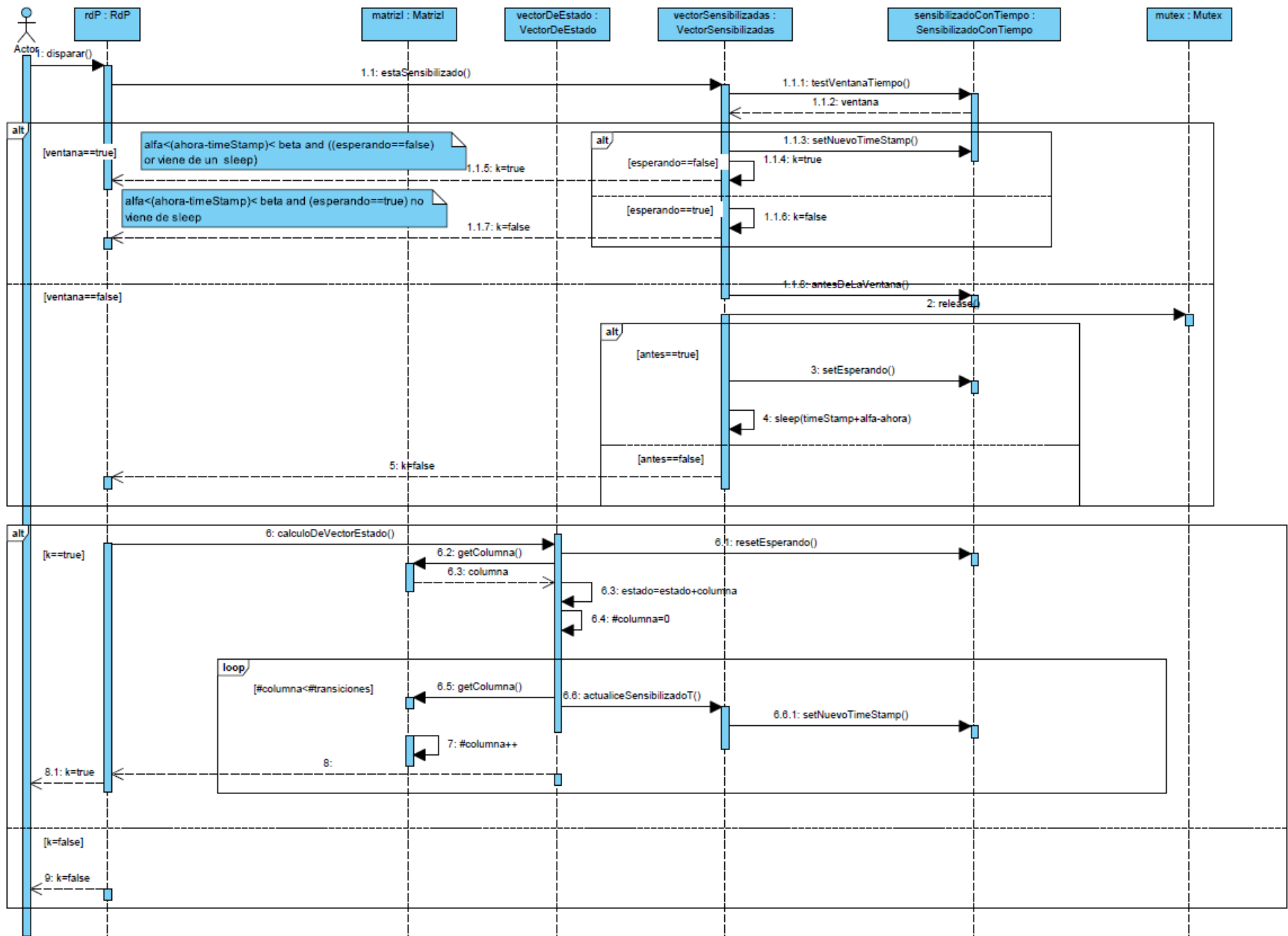
```

2 public interface BufferI {
3     /**
4     * insert an item into the Buffer.
5     * Note this may be either a blocking
6     * or non-blocking operation.
7     */
8     public abstract void insert(Object item);
9
10    /**
11    * remove an item from the Buffer.
12    * Note this may be either a blocking
13    * or non-blocking operation.
14    */
15    public abstract Object remove();
16 }

```

- productorConsumidor
 - Buffer.java
 - BufferI.java
 - Consumidor.java
 - Productor.java
 - SleepUtilities.java
 - ThreadApp2.java





Bibliografía

Notas de clase de Programación Concurrente

Comunicación y Sincronización con Monitores

Resumen del Tema

**Dpto de Lenguajes y
Ciencias de la Computación
Universidad de Málaga**

María del Mar Gallardo Melgarejo

Málaga, 2002