

PROGRAMACION CONCURRENTE

Sincronización basada en memoria
compartida:

Sincronización y Semáforos
2016-17

Procesos concurrentes y memoria compartida

- Si los diferentes hilos de un programa concurrente tienen acceso a variables globales o secciones de memoria comunes, la transferencia de datos a través de ella es una vía habitual de comunicación y sincronización entre ellos.
- Las primitivas para programación concurrente basada en memoria compartida resuelven los problemas de sincronización entre procesos y de exclusión mutua utilizando la semántica de acceso a memoria compartida.
- En esta familias de primitivas, la semántica de las sentencias hace referencia a la exclusión mutua, y la implementación de la sincronización entre procesos resulta de forma indirecta.

Mecanismos basados en memoria compartida

- **Secciones críticas:** Son mecanismos de nivel medio de abstracción orientados a su implementación en el contexto de un lenguaje y que permiten la ejecución de un bloque de sentencias de forma segura.
- **Semáforos:** Son componentes pasivos de bajo nivel de abstracción que sirven para arbitrar el acceso a un recurso compartido.
- **Monitores:** Son módulos de alto nivel de abstracción orientados a la gestión de recursos que van a ser usados concurrentemente. (otra clase)

Recursos de Java para sincronizar threads

- Todos los objetos tienen un bloqueo asociado, lock o cerrojo, que puede ser adquirido y liberado mediante el uso de métodos y sentencias **synchronized**.
- La sincronización fuerza a que la ejecución de los dos hilos sea mutuamente exclusiva en el tiempo.
- **Mecanismos de bloqueo:**
 - **Métodos synchronized** (exclusión mutua).
 - **Bloques synchronized** (regiones críticas).
- Mecanismos de comunicación de los threads (variables de condición):
 - Wait(),
 - notify(),
 - notifyAll()...
- Cualquier otro mecanismo:
 - Dado que con monitores se pueden implementar los restantes mecanismos de sincronización (Semáforos, Comunicación síncrona, Invocación de procedimientos remotos, etc.) se pueden encapsular estos elementos en clases y basar la concurrencia en ellos.

Lock asociado a los componentes Java.

- Cada objeto derivado de la clase `Object` (esto es, prácticamente todos) tienen asociado un elemento de sincronización o lock intrínseco, que afecta a la ejecución de los métodos definidos como `synchronized` en el objeto:
 - Cuando un objeto ejecuta un **método `synchronized`**, toma el lock, y cuando termina de ejecutarlo lo libera.
 - Cuando un **thread tiene tomado el lock**, ningún otro thread puede ejecutar ningún otro método `synchronized` del mismo objeto.
 - El thread que **ejecuta un método `synchronized`** de un objeto cuyo lock se encuentra tomado, se suspende hasta que el objeto es liberado y se le concede el acceso.
- **Cada clase Java derivada de `Object`**, tiene también un mecanismo lock asociado a ella (que es independiente del asociado a los objetos de esa clase) y que afecta a los procedimientos estáticos declarados `synchronized`.

Bloques synchronized

- Es el mecanismo mediante el que se implementan en Java las **regiones críticas**.
- Un bloque de código puede ser definido como synchronized respecto de un objeto.
- En ese caso solo se ejecuta si se obtiene el lock asociado al objeto
synchronized (object)
{ Bloque atómico}
- Se suele utilizar cuando se necesita en un entorno concurrente y se usa un objeto diseñado para un entorno secuencial.

Ejemplo de utilización de bloque synchronized

- // Hace que todos los elementos del array sean no negativos

```
public static void abs(int[] valores){  
    synchronized (valores){  
        // Sección crítica  
        for (int i=0; i<valores.length; i++){  
            if (valores[i]<0) valores[i]= -valores[i];  
        }  
    }  
}
```

Métodos synchronized

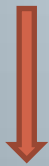
- Los **métodos** de una clase Java se pueden declarar como **synchronized**. Esto significa que **se garantiza que se ejecutará con régimen de exclusión mutua** respecto de otro método del mismo objeto que también sea synchronized.
- **Cuando un thread** invoca un método synchronized, trata de tomar **el lock del objeto** a que pertenezca.
 - Si está libre, lo toma y se ejecuta.
 - Si el lock está tomado por otro thread, se suspende el que invoca hasta que aquel finalice y libere el lock.
- Si el método **synchronized es estático** (static), el lock al que hace referencia es **de clase y no de objeto**, por lo que se hace en exclusión mutua con cualquier otro método estático synchronized de la misma clase.

Ejemplo de método synchronized

```
class CuentaBancaria{  
    private class Deposito{  
        protected double cantidad;  
        protected String moneda = "Euro"  
    }  
    Deposito elDeposito;  
    public CuentaBancaria(double initialDeposito,String moneda){  
        elDeposito.cantidad= initialDeposito;  
        elDeposito.moneda=moneda; }  
    public synchronized double saldo(){return elDeposito.cantidad;}  
    public synchronized void ingresa(double cantidad){  
        elDeposito.cantidad=elDeposito.cantidad + cantidad;  
    }  
}
```

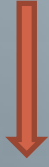
Exclusión mutua

Thread 2->lock tomado->Espera



```
public synchronized double saldo(){  
    return elDeposito.cantidad;  
}
```

Thread 1->Toma el lock->Ejecuta



```
public synchronized void ingresa(double cantidad){  
    elDeposito.cantidad=elDeposito.cantidad + cantidad;  
}
```

Consideraciones sobre métodos synchronized.

- El lock es tomado por el thread, por lo que mientras un thread tiene tomado el lock de un objeto puede acceder a otro método synchronized del mismo objeto.
- El lock es por cada instancia del objeto.
- Los métodos de clase (static) también pueden ser synchronized.
 - Por cada clase hay un lock y es relativo a todos los métodos synchronized de la clase.
 - Este lock no afecta a los accesos a los métodos synchronized de los objetos que son instancia de la clase.
- Cuando una clase se extiende y un método se sobrescribe, este se puede definir como synchronized o nó, con independencia de cómo era y como sigue siendo el método de la clase madre.

Definición de semáforo

- Un semáforo es un tipo de datos.
- Como cualquier tipo de datos, queda definido por:
 - Conjunto de valores que se le pueden asignar.
 - Conjunto de operaciones que se le pueden aplicar.
- Un semáforo tiene **asociada una lista/cola** de procesos, en la que se incluyen los procesos suspendidos a la espera de su cambio de estado.

Valores de un semáforo.

- En función del rango de valores que puede tomar, los semáforos se clasifican en: [?]
 - Semáforos **binarios**: Pueden tomar solo los valores 0 y 1.
var mutex: BinSemaphore; [?]
 - Semáforos **general**: Puede tomar cualquier valor Natural (entero no negativo).
var escribiendo: Semaphore;
- Un semáforo que ha tomado el valor 0 representa un semáforo cerrado, y si toma un valor no nulo representa un semáforo abierto.
- Es posible demostrar que un semáforo General se puede implementar utilizando semáforos Binarios.
- Los sistemas suelen ofrecer como componente primitivo semáforos generales, y su uso, lo convierte de hecho en semáforo binario.

Métodos de Object para sincronización.

- Un semáforo

var p: semaphore;

- admite dos operaciones seguras: `?`
- **wait(p):**
 - Si el semáforo no es nulo (abierto) decrementa en uno el valor del semáforo. Si el valor del semáforo es nulo (cerrado), el thread que lo ejecuta se suspende y se encola en la lista de procesos en espera del semáforo. `?`
- **signal(p);**
 - Si hay algún proceso en la lista de procesos del semáforo, activa uno de ellos para que ejecute la sentencia que sigue al wait que lo suspendió. Si no hay procesos en espera en la lista incrementa en 1 el valor del semáforo
- **Objetos como Semaforos**

Java: Métodos de Object para sincronización

- Todos son métodos de la clase object. Solo se pueden invocar por el thread propietario del lock (p.e. dentro de métodos synchronized).
- En caso contrario lanzan la excepción `IllegalMonitorStateException`

public final void wait() throws InterruptedException

- Espera indefinida hasta que reciba una notificación.

public final void wait(long timeout) throws InterruptedException

- El thread que ejecuta el método se suspende hasta que, o bien recibe una notificación, o bien, transcurre el timeout establecido en el argumento.
- `wait(0)` representa una espera indefinida hasta que llegue la notificación.

public final void wait(long timeout, int nanos)

- Wait en el que el tiempo de timeout es $1000000 * \text{timeout} + \text{nanos}$ nanosegundos

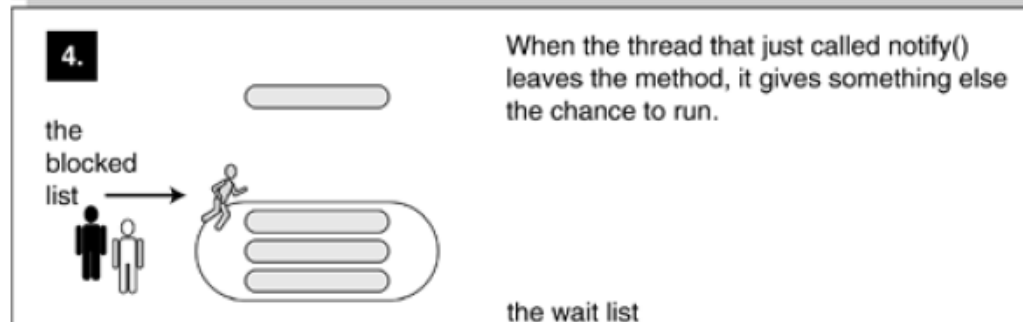
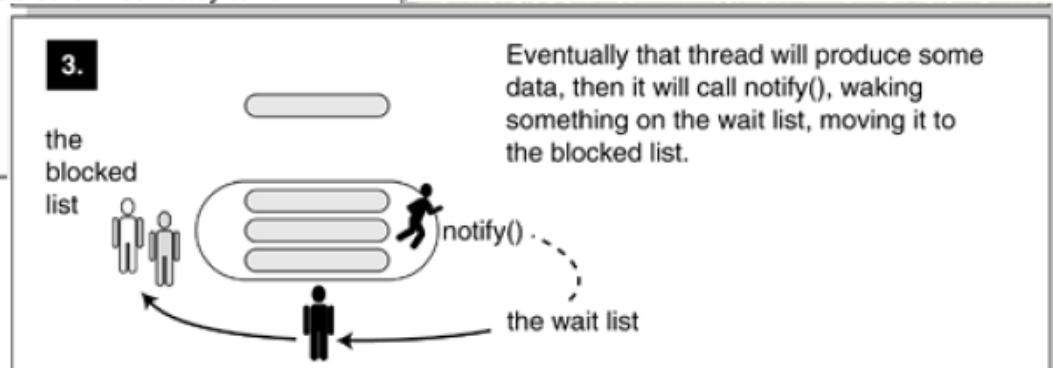
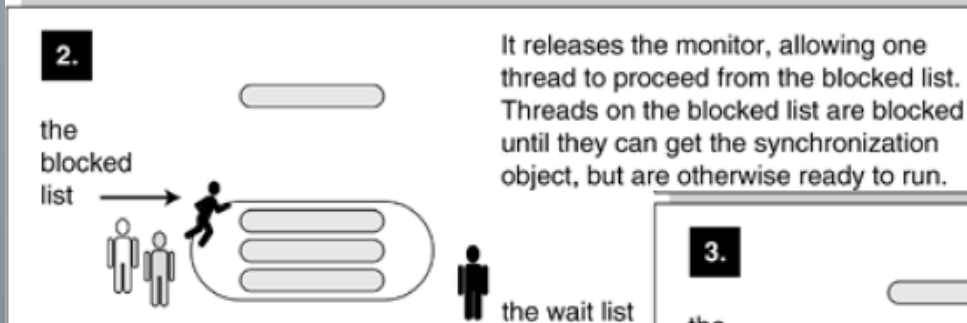
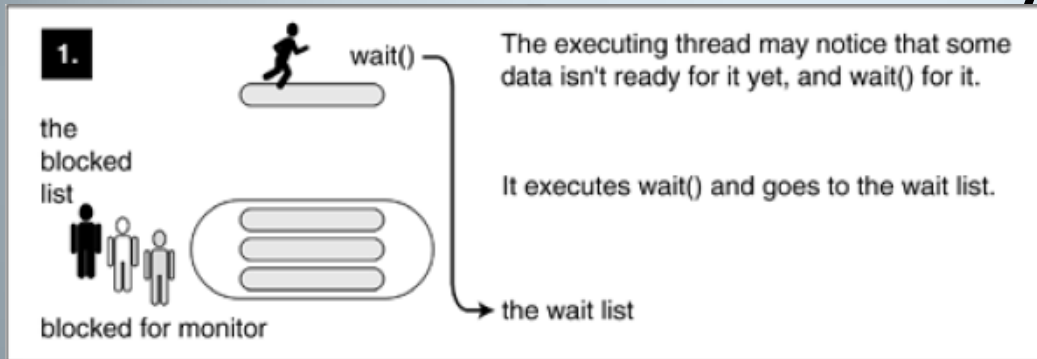
public final void notify()

- Notifica al objeto un cambio de estado, esta notificación es transferida a solo uno de los threads que esperan (han ejecutado un `wait`) sobre el objeto. No se puede especificar a cual de los objetos que esperan en el objeto será despertado.

public final void notifyAll()

- Notifica a todos los threads que esperan (han ejecutado un `wait`) sobre el objeto.

Protocolo Wait/Notify



Uso de los métodos wait().

- Debe utilizarse siempre dentro de un método synchronized y habitualmente dentro de un ciclo indefinido que verifica la condición:

```
synchronized void HazSiCondicion(){  
while (!Condicion) wait();  
..... // Hace lo que haya que hacer si la condición es cierta.  
}
```

- Cuando se suspende el thread en el wait, libera el lock que poseía sobre el objeto.
- La suspensión del thread y la liberación del lock son atómicos (nada puede ocurrir entre ellos).
- Cuando el thread es despertado como consecuencia de una notificación, la activación del thread y la toma del lock del objeto son también atómicos (Nada puede ocurrir entre ellos).

Uso de los métodos notify().

- Debe utilizarse siempre dentro de un método synchronized:

```
synchronized void changeCondition(){  
..... // Se cambia algo que puede hacer que la condición se satisfaga.  
notifyAll();  
}
```
- Muchos thread pueden estar esperando sobre el objeto:
 - Si se utiliza notify() solo un thread (no se sabe cual) es despertado.
 - Si se utiliza notifyAll() todos los thread son despertados y cada uno decide si la notificación le afecta, o si no, vuelve a ejecutar el wait() (dentro del while).
- El proceso suspendido debe esperar hasta que el procedimiento que invoca notify() o notifyAll ha liberado el lock del objeto.

Ejemplo de sincronización: Establecimiento de una variable.

- Thread 1

```
public synchronized guardedJoy() {  
    //This guard only loops once for  
    each  
    //special event, which may not  
    //be the event we're waiting for.  
    while(!joy) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    System.out.println("Joy and  
    efficiency have been achieved!");  
}
```

Thread 2

```
public synchronized  
notifyJoy() {  
    joy = true; notifyAll();  
}
```

Cuando sí/no sincronizar

- **Cuándo sincronizar**

“En cualquier momento en el que dos o más threads acceden al mismo objeto o campo.”

- **Cuándo no sincronizar**

“Sobresincronizar” causa retrasos innecesarios cuando dos o más threads tratan de ejecutar el mismo bloque de código .

- **Por ejemplo, no sincronizar el método entero, cuando sólo una parte necesita ser sincronizada.**

- **Poner sólo un bloque sincronizado en esa parte del código:**

```
public void myMethod() {  
    synchronized(this) {  
        // code that needs to be synchronized  
    }  
    // code that is already thread-safe  
}
```

- **No sincronizar un método que usa variables locales. Las variables locales son almacenadas en el stack, y cada thread tiene su propio stack, así que, no habrá problemas de concurrencia:**

- ```
public int square(int n) {
```
- ```
int s = n * n;
```
- ```
return s;
```
- ```
}
```

Operación no segura de un semáforo

- Un semáforo

var p: semaphore;

- admite una operación no segura: $\boxed{?}$

initial(p, Valor_inicial):

- Asigna al semáforo p el valor inicial que se pasa como argumento.
- Esta operación **es no segura** y por tanto debe ser ejecutada en una fase del programa en la que se tenga asegurada que se ejecuta sin posibilidad de concurrencia con otra operación sobre el mismo semáforo.

Operación wait.

- Pseudocódigo de la operación: wait(p);
 if $p > 0$
 then $p := p - 1$;
 else Suspende el proceso y lo encola en la lista del semáforo.
- Lo característico de esta operación es que está protegida contra interrupción entre el chequeo del valor del semáforo y la asignación del nuevo valor.
- El nombre de la operación wait es equívoco. En contra de su significado semántico natural, su ejecución a veces provoca una suspensión pero en otros caso no implica ninguna suspensión.

Operación signal

- Pseudocódigo de la operación: `signal(p);`
if Hay algún proceso en la lista del semáforo
then Activa uno de ellos
else `p:= p+1;`
- La ejecución de la operación `signal(p)` nunca provoca una suspensión del thread que lo ejecuta.
- Si hay varios procesos en la lista del semáforo, la operación `signal` solo activa uno de ellos.
- Este se elige de acuerdo con un criterio propio de la implementación (FIFO, LIFO, Prioridad, etc.).

Semaforos

- Java 7 Concurrency Cookbook Thread Synchronization Utilities.pptx
- Filminas 5 hasta 15

Sincronización basada en semáforos.



```
var datoDisponible: Semaphore:= 0;
```

```
process Productor;  
var dato: Tipo_Dato;  
begin  
  repeat  
    produceDato(var dato);  
    dejaDato(dato);  
    signal(datoDisponible);  
  forever;  
end;
```

```
process Consumidor;  
var dato: Tipo_Dato;  
begin  
  repeat  
    wait(datoDisponible);  
    tomaDato(var dato);  
    consume(dato);  
  forever;  
end;
```

Concurrencia en Java

- **Clases de utilidad para sincronización**
- La API de concurrencia tiene varias clases para la sincronización en el paquete `java.util.concurrent` como:
 - Semaphore,
 - CountdownLatch,
 - CyclicBarrier,
 - Phaser y Exchanger.

Semaphore

- Un semáforo tiene un contador que permite el acceso a un recurso compartido:
 - Cuando un thread **quiere adquirir** un semáforo, este verifica su contador y
 - si es **mayor** que **cero**, entonces el thread **obtiene** el semáforo y se reduce el contador.
 - Sin embargo, si el contador **es cero** el thread **espera** hasta que se incremente el contador.

Recurso

- Una clase (objeto) Recurso compartido:
 - el método **acquire()** trata de obtener el semáforo
 - Lo obtiene si el contador es mayor que cero,
 - No lo obtiene si es cero
 - espera hasta que se incremente el semáforo y reduce el contador,
 - El thread que obtiene el semáforo usa el recurso y finalmente, se ejecuta el método **release()** para incrementa el semaforo.
 - Si el semáforo inicializa su contador:
 - con un valor uno, se llama un semáforo binario y
 - se comporta como la interfaz `java.util.concurrent.locks.Lock`.

Recurso

```
package tareas;  
import java.util.concurrent.Semaphore;
```

```
public class Recurso {
```

```
    Semaphore semaphore = new Semaphore(1);
```

```
    //Indica que dos threads pueden acceder el recurso al mismo tiempo.
```

```
    public void lock() throws InterruptedException {
```

```
        semaphore.acquire();
```

```
    //Si el contador es cero el thread se duerme, de lo contrario se reduce y obtiene el acceso
```

```
        System.out.println("Comenzando el thread"+ Thread.currentThread().getName()+" a usar el  
recursotiene el lock");
```

```
        try {
```

```
            Thread.sleep(1000);
```

```
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
    // aqui van las acciones que se quieren realizar con el recurso
```

```
        System.out.println("Terminado el thread"+ Thread.currentThread().getName()+" de usar el  
recursoesta soltando el lock");
```

```
        semaphore.release();//Libera el semaphore e incrementa el contador
```

```
    }
```

```
}
```

Producer-Consumer

- P-Invariant equations**

$$M(P0) + M(P1) + M(P2) = 1$$

$$M(P3) + M(P4) + M(P5) = 1$$

$$M(P1) + M(P4) + M(P6) + M(P7) = 3$$

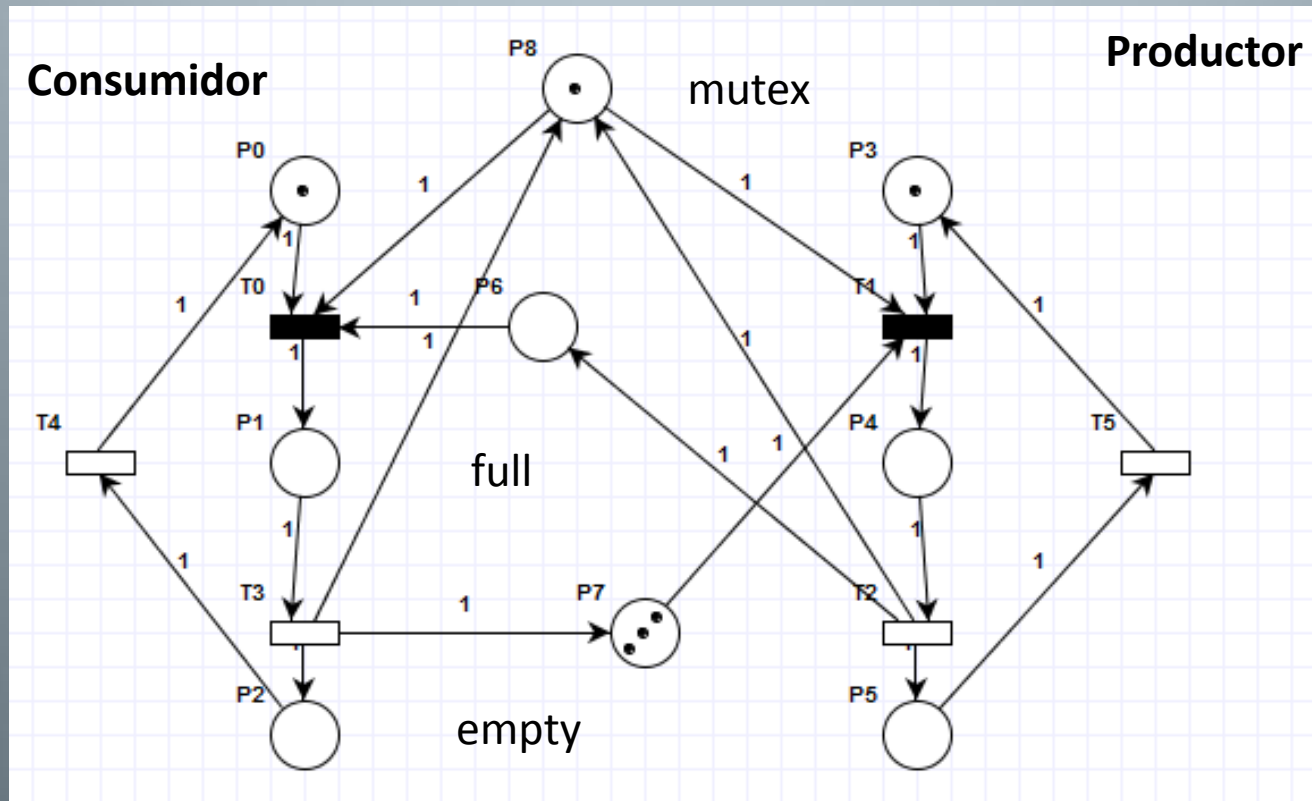
$$M(P1) + M(P4) + M(P8) = 1$$

- Petri net state space analysis results**

Bounded true

Safe false

Deadlock false



Producer-Consumidor

- **Clases**
 - **Solution**
 - Esta clase crea el búfer y los hilos del productor y del consumidor. (Autor: Silberschatz, Galvin, and Gagne)
 - **Producer** implements Runnable
 - Éste es el subproceso del productor para el problema del búfer limitado.
 - **Consumer** implements Runnable
 - Éste es el subproceso de consumidor para el problema del búfer limitado.
 - **interface Buffer**
 - Una interface para buffers
 - Metodos
 - `public abstract void insert(Object item);`
 - `public abstract Object remove();`
 - **BoundedBuffer** implements Buffer
 - This program implements the bounded buffer using shared memory.
 - **SleepUtilities**
 - Utilidades para hacer que un hilo duerma.

Productor-Consumidor

```
/**
```

```
 * Esta clase crea el búfer y los hilos del productor y del consumidor.
```

```
 * tomado de Original code by Silberschatz, Galvin, and Gagne
```

```
 */
```

```
public class Solution{
```

```
    public static void main(String args[]) {
```

```
        // Se crea el Buffer (create) que es compartido por el productor y el consumidor
```

```
        Buffer sharedBuffer = new BoundedBuffer();
```

```
        // se crea el producer y consumer (Son threads)
```

```
        Thread producerThread = new Thread(new Producer(sharedBuffer));
```

```
        Thread consumerThread = new Thread(new Consumer(sharedBuffer));
```

```
        // se arranca los hilos productor y consumidor start()
```

```
        // se reserva memoria para los nuevos thread en la JVM,
```

```
        // se llama al metodo run()
```

```
        producerThread.start();
```

```
        consumerThread.start();
```

```
    }
```

```
}
```


Productor-Consumidor

```
/**
 * Éste es el subproceso del productor para el problema del búfer limitado.
 */
import java.util.Date;
class Producer implements Runnable{

    private Buffer buffer;

    public Producer(Buffer b) {
        buffer = b;
    }

    public void run(){
        Date message;
        while (true) {
            System.out.println("Siesta del productor");
            SleepUtilities.nap();
            // Producir un ítem & introducirlo en el buffer
            message = new Date();
            System.out.println("Producer produced \"\" + message + "\"");
            buffer.insert(message);
        }
    }
}
```

Productor-Consumidor

```
/**
 * Éste es el subproceso de consumidor para el problema del búfer limitado.
 */
import java.util.Date;
class Consumer implements Runnable{

    private Buffer buffer;

    public Consumer(Buffer b) {
        buffer = b;
    }

    public void run(){
        Date message = null;
        while (true){
            System.out.println("Consumer napping");
            SleepUtilities.nap();
            // Consumir un elemento del buffer
            System.out.println("Consumer wants to consume");
            message = (Date)buffer.remove();
            System.out.println("Consumer consumed \"" + message + "\"");
        }
    }
}
```

Productor-Consumidor

```
/**
 * An interface for buffers
 *
 */

interface Buffer{
/**
 * insert, este metodo inserta un item en Buffer.
 * Note este metodo puede ser blocking
 * o non-blocking operacion.
 */
    public abstract void insert(Object item);

/**
 * remove, saca un item del Buffer.
 * Note este metodo puede ser blocking
 * o non-blocking operacion.
 */
    public abstract Object remove();
}
```

Productor-Consumidor

```
/**
 * This program implements the bounded buffer using shared memory.
 */
import java.util.Date;
import java.util.concurrent.Semaphore;
class BoundedBuffer implements Buffer{

    private static final int BUFFER_SIZE = 3; //maximo tamagno del buffer array
    private int count; //numero de items currientemente en el buffer
    private int in; // points to the next free position in the buffer
    private int out; // points to the first filled position in the buffer
    private Object[] buffer; //array of Objects
    private Semaphore mutex; //provides limited access to the buffer (mutual exclusion)
    private Semaphore empty; //keep track of the number of empty elements in the array
    private Semaphore full; //keep track of the number of filled elements in the array

    public BoundedBuffer(){
        // buffer is initially empty
        count = 0;
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];
        mutex = new Semaphore(1); //1 for mutual exclusion
        empty = new Semaphore(BUFFER_SIZE); //array begins with all empty elements
        full = new Semaphore(0); //array begins with no elements
    }
}
```

Productor-Consumidor

```
// producer calls this method
public void insert(Object item) {
    /*
        while (count == BUFFER_SIZE){
            // do nothing, if the buffer array cannot be used (because full)
        }
    */
    try{
        empty.acquire(); // seguimiento del número de elementos vacíos (valor--)
        // Esto proporciona la sincronización para el productor,
        // porque esto hace que el productor se suspenda cuando el búfer esté lleno
        mutex.acquire(); //mutual exclusion
    }
    catch (InterruptedException e) {
        System.out.println("ERROR in insert(): " + e);
    }

    // add un item en el buffer
    ++count;
    buffer[in] = item;
    // módulo (%) es el resto de una división
    // por ejemplo, 0% 3 = 0, 1% 3 = 1, 2% 3 = 2, 3% 3 = 0, 4% 3 = 1, 5% 3 = 2
    in = (in + 1) % BUFFER_SIZE;

    //Información del buffer
    if (count == BUFFER_SIZE){
        System.out.println("BUFFER FULL "
            + "Producer inserted \"" + item
            + "\" count=" + count + ", "
            + "in=" + in + ", out=" + out);
    }
    else{
        System.out.println("Producer inserted \"" + item
            + "\" count=" + count + ", "
            + "in=" + in + ", out=" + out);
    }

    mutex.release(); //mutual exclusion
    full.release(); //lleva la cuenta del number de elements (value++)
    //si el buffer esta vacío, Entonces esto despierta al Consumidor, Consumer
}
```

Productor-Consumidor

```
// consumer llama a este method
public Object remove() {
    Object item=null;
    /*
        while (count == 0){
            //Si no hay nada en el búfer, entonces no hacer nada
            // no se puede usar la matriz de búfer (porque está vacía)
        }
    */

    try{
        full.acquire(); //Realizar un seguimiento del número de elementos (valor--)
        // Esto proporciona la sincronización para el consumidor,
        // porque esto hace que el consumidor se suspenda cuando el buffer está vacío

        mutex.acquire(); //mutual exclusion
    }
    catch (InterruptedException e) {
        System.out.println("ERROR in try(): " + e);
    }

    // remove an item from the buffer
    --count;
    item = buffer[out];
    //Módulo (%) es el resto de una división
    // por ejemplo, 0% 3 = 0, 1% 3 = 1, 2% 3 = 2, 3% 3 = 0, 4% 3 = 1, 5% 3 = 2
    out = (out + 1) % BUFFER_SIZE;

    //buffer information feedback
    if (count == 0){
        System.out.println("BUFFER EMPTY "
            + "Consumer removed \"" + item
            + "\" count=" + count + ", "
            + "in=" + in + ", out=" + out);
    }
    else{
        System.out.println("Consumer removed \"" + item
            + "\" count=" + count + ", "
            + "in=" + in + ", out=" + out);
    }

    mutex.release(); //mutual exclusion
    empty.release(); //Realizar un seguimiento del número de elementos vacíos (valor ++)
    // si el búfer estaba lleno, entonces esto despierta al productor
    return item;
}

}
```

Productor-Consumidor

```
* Utilidades para hacer que un hilo duerma.  
* Nota, debemos manejar excepciones interrumpidas  
* Pero se elige no hacerlo para claridad de código.  
*  
*/
```

```
class SleepUtilities{
```

```
    private static final int NAP_TIME = 5; //Tiempo de siesta máximo en segundos
```

```
    /**
```

```
     * Siesta entre cero y NAP_TIME segundos.
```

```
     */
```

```
    public static void nap() {
```

```
        nap(NAP_TIME);
```

```
    }
```

```
    /**
```

```
     * Siesta entre cero y segundos de duración.
```

```
     */
```

```
    public static void nap(int duration) {
```

```
        int sleeptime = (int) (NAP_TIME * Math.random());
```

```
        System.out.println("Nap for " + sleeptime + " seconds");
```

```
        // Hace que el subproceso en ejecución se detenga (cese la ejecución)
```

```
        // para el número especificado de milisegundos,
```

```
        // sujeto a la precisión y exactitud de los temporizadores y programadores del sistema.
```

```
        try { Thread.sleep(sleeptime*1000); }
```

```
        catch (InterruptedException e) {
```

```
            // método sleep () throws InterruptedException - si algún hilo ha interrumpido el subproceso actual.
```

```
            System.out.println("ERROR in nap(): " + e);
```

```
        }
```

```
    }
```

```
}
```

Critica de los semáforos

- Ventajas de los semáforos:
 - Resuelven todos los problemas que presenta la concurrencia.
 - Estructuras pasivas muy simples.
 - Fáciles de comprender.
 - Tienen implementación muy eficientes.
- Peligros de los semáforos:
 - Son de muy bajo nivel y un simple olvido o cambio de orden conducen a bloqueos.
 - Requieren que la gestión de un semáforo se distribuya por todo el código.
 - La depuración de los errores en su gestión es muy difícil.
- Los semáforos son los componentes básicos que ofrecen todas las plataformas hardware y software como base para construir los mecanismos de sincronización. Las restantes primitivas se basan en su uso implícito.

Sincronización

- Java 7 Concurrency Cookbook Basic Thread Synchronization.pptx
- Filmina 2 hasta filmina 18